Historical development of operating systems: Different decades brought different generations of operating systems.

- In the ’40s, there were no operating systems.
- In the ’50s, batch operating systems were developed to smooth the transition between jobs.
- In the ’60s, multistream batch systems and timesharing systems were developed.
  - Interactive systems
  - Turnaround time vs. response time.
  - Virtual storage.
- In the ’70s, timesharing came of age.
  - Controlled sharing and protection: Multics.
  - Object-oriented systems.
  - The emergence of UNIX™.
- In the ’80s, the major influences were the personal computer and parallel processing.
  - Computer networks.
  - Graphical interfaces.
  - Concurrency and distribution of responsibility.
- The ’90s have seen networking come to the fore.
  - Fileservers.
  - Client-server model.
  - Programs loaded across the Web.

Structure of operating systems: An OS manages a variety of hardware and software resources.

- Processors, memories, I/O devices.
- Programs and data files.

“Beautification principle—an operating system is a set of algorithms that hide the details of the hardware and provide a more pleasant environment.”—Finkel.

How might an operating system be organized?
Monolithic structure: Many modules, e.g.,

- processor scheduling,
- memory management,
- device management,

but only one of these can execute at a time.

- A single process.
- Control transferred by procedure calls and branches.

User programs may be viewed as “subroutines” that are invoked by the OS.

How does the OS regain control?

- Program terminates (like a “return”).
- Program runs out of time.
- Program requests system service, e.g., I/O.
- An interrupt occurs, indicating that, e.g., a device requires attention of the OS.

A monolithic operating system:

As they grow large, monolithic systems become unreliable.

The kernel approach: The most vital low-level functions are collected in a “kernel,” or “nucleus.”
Other parts of the operating system call kernel functions to perform these low-level tasks.

The kernel is protected against other parts of the system.

There may be more than one process at a time in a kernel-based OS.

(A process can be thought of as a program in execution.)

- The kernel is responsible for dividing CPU time among all the processes.
- Some processes are “system processes” and others are “user processes.”
- There must be some way for the processes to interact.

Processes can communicate via *send* and *receive* primitives.

- Each primitive specifies a buffer that can hold the message.
- It also specifies where the message should be taken from or put.

Why do we need both of these?
• Also, the receiver (or sender) must be specified.

E.g., Receive a message from YourProcess, taking the message out of OurBuffer and putting it into variable Msg.

Process hierarchy: In a kernel-based OS, everything is either in the kernel or not in the kernel.

If the OS is large, the kernel may grow fairly large itself, raising the same problems as before.

Therefore, the operating system may be structured in more than two levels.

An early example of this was Dijkstra’s THE operating system.

<table>
<thead>
<tr>
<th>User₁</th>
<th>User₂</th>
<th>…</th>
<th>Userₙ</th>
<th>L4: Indep. user processes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I/O device processes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L3: Virtual I/O devices</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Command Interpreter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L2: Virtual Operator Consoles</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Segment Interpreter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>L1: Virtual Segmented Memory</td>
</tr>
<tr>
<td>CPU alloc., synchroniz’t’n.</td>
<td></td>
<td></td>
<td></td>
<td>L0: Virtual CPUs</td>
</tr>
<tr>
<td>CPU</td>
<td></td>
<td></td>
<td></td>
<td>Actual hardware</td>
</tr>
</tbody>
</table>

Each layer defines a successively more abstract “virtual machine.”

Each layer performs a resource-management task.

Processes further up the hierarchy can use resources handled at lower levels.

• Processors above level 0 need not be concerned with the number of available physical processors.
• All processes above level 2 have the illusion of having their own “console.”
**Functional hierarchy:** It is difficult to structure an OS strictly in terms of a functional hierarchy.

Consider process management and memory management.

- To create a new process, the *process manager* calls the *memory manager* to allocate space for the new process.

- When memory is full, the *memory manager* calls the *process manager* to deactivate a process, thereby freeing memory.

\[ \therefore \] It is not always possible to organize processes into a strict hierarchy. So—place functions, not processes into a hierarchy.

- Each level consists of a set of functions.
- The levels \( L_0, L_1, \ldots, L_n \) are ordered such that—
  - functions defined at level \( L_i \) are also known to \( L_{i+1} \), and
  - at the discretion of \( L_{i+1} \), also to levels \( L_{i+2} \), etc.
  - \( L_0 \) corresponds to the hardware instructions of the machine.

- Each level provides a new "computer"—a virtual machine—to the next higher level.

**Example:** Memory and process management.
• The functions of process creation and destruction need segment creation and destruction, and thus are placed above the latter.

• Similarly, process scheduling is placed above segment management.

• Why are segment creation and destruction are above process scheduling?

Object-oriented structure: A process can be considered to be an object.

But a process is only one kind of object.

Other kinds of objects include files, procedures, and pages of memory.

Basic idea: A object-oriented system can be characterized as—

• a collection of abstract entities
• with certain basic properties and
• precisely defined interactions.

Each part of the system only needs to know—

• the properties of an object, and
• the operations that may be performed on it

in order to be able to use it properly.

When an OS is structured according to the object model, each function (whether inside or outside the “kernel”) has access to only the data that it needs to manipulate.

Definition: An operating system (computer) is said to implement the object model if each action performed by the operating system (computer) is the application of some operation to one or more objects.
This is analogous to execution of ordinary instructions, where each instruction operates on a set of operands.

The object model is supported by various

- Languages, such as Smalltalk, C++, and Java.
- Operating systems, such as CAP, Hydra, STAROS, Monads, iMAX432.
- Distributed systems frameworks, such as CORBA, OLE, and DSOM.

Examples of objects:

(a) From real life—an elevator.
   - Properties: weight, volume, location (floor).
   - Operations: call, go up, go down, open door, close door.

(b) From application programs—a stack.
   - Properties: size (number of items); contents (the items themselves).
   - Operations: push, pop, inspect top element, initialize.

(c) From operating systems—a file system (Peterson & Silberschatz, §3.2).
   - Properties
     - Directory structure.
     - Information about existing files (name, type, location, size, current position, protection, usage count, time, etc.).
   - Operations:
     - Create file.
       - Find space for the new file.
       - Enter the file in a directory.
     - Write file.
       - Search directory to find file.
       - From end-of-file pointer, determine location of next block.
       - Write information to next block.
       - Update end-of-file pointer.
• Processes.

Definition: A process is an executing program. More precisely: A process is an entity representing a program in execution.

• Operations:

◊ Create process.
◊ Delete process.
◊ Suspend process (remove it from the processor; temporarily stop it).
◊ Resume process.

etc.

Processes and process states: A process is an executing program (an entity representing a program in execution).

• A process is a sequence of actions and is therefore dynamic.
• A program is a sequence of instructions and is therefore static.

A process may be in one of three states:

• Running. Currently assigned to the (a) processor, which is executing it.

Such a process is called the current process.

• Ready. The process could run, if (all) the processor(s) were not busy running other processes. A ready process is found on the ready queue.
• **Blocked.** The process cannot run until something happens.

For example, it may be awaiting the completion of input, or communication from another process.

Such a process will be *queued* on a particular *device* or *event*.

---

**Transition diagram:**

- **Running**
- **Ready**
- **Blocked**

Diagram arrows indicate transitions:
- Assign
- Preempt
- Block
- Awaken

---

**Representation of a process** (S&G, §4.1.3):

According to the object model, each object has a *representation* and a set of operations defined on it.

A process is represented by a *Process Control Block* (PCB).

The contents of the PCB vary from one operating system to another. In general, though, the following set of items is representative:

- **Process #**
- **Proc. status**
- **Program ctr.**
- **Register save area**
- **Memory-management information**
- **Accounting information**
- **I/O status information**
- **Scheduling information**
• The process number (also called process name, PID). Uniquely identifies this process.
• The process status (running, ready, or blocked).
• Program counter. Address of the next instruction that this process will execute.
• Register save area. A copy of the information in the CPU registers (programmable and non-programmable) when the process was last blocked or preempted.
• Memory-management information. E.g., base/length registers, page table.
• Accounting information. Amount of time used, time limit, user ID, etc.
• I/O status information. Outstanding I/O requests, open files, devices allocated, if any.
• Scheduling information. Pointer to ready queue or device queue, priority, etc.

**Suspending a process** (S&G, §4.1.1): Sometimes it is desirable to suspend a process, i.e., not to consider it for execution for awhile. Usually because—

- the system is too heavily loaded, and there is not enough room for all processes in main memory, or
- the system is unstable and may crash soon. If the process is running when that happens, it may be difficult or impossible to continue execution after the system comes back up.

A process in any state may be suspended.
Here is a more complete list of operations on processes:

- Create Process
- Destroy Process
- Suspend Process
- Resume Process
- Change Process’s Priority
- Block Process
- Awaken Process
- Assign Process

**Process scheduling** (S&G, §4.2): Why schedule the processor? (Why not just let each process run until it’s finished?) Well, ...

A typical process runs as follows:

```
Execute
Wait for I/O to complete
Execute
: 
: 
```

Consider two processes \( A \) and \( B \) that each

```
60 times \[
\begin{align*}
\text{Execute} & \quad (1 \text{ second}) \\
\text{Wait} & \quad (1 \text{ second}) \\
: & \\
: & 
\end{align*}
\]
```

If the processor runs \( A \) to completion and then \( B \), it takes

- 4 minutes to complete both processes
- 2 minutes (total) of processor time used (2 minutes wasted)
- 50% processor utilization.

If we interleave \( A \) and \( B \), it takes

- 2 minutes (~1 second) to finish \( A \)
- 2 minutes to finish \( B \)

≈100% processor utilization.

While one process waits for I/O, another process can be executed.

How is the processor switched between processes?
Process $A$ executes

Save state

*Handle interrupt.*

**Decide which process to run next.**

Restore state

Process $B$ executes

Save state

*Handle interrupt.*

**Decide which process to run next.**

Restore state

Process $A$ executes

When is the processor switched between processes? Examples:

- When the executing process performs a system call.
- When a higher-priority event needs attention.
- When a higher-priority process needs to be run.
- When a timer interrupt occurs.

*Scheduling queues* (S&G, §4.2.1): There are several queues within an operating system. Most of these should be familiar to you from earlier courses.
• Ready queue:

• Device queue:

• Job queue:

Queues are often maintained using what data structure?

Schedulers (S&G, §4.2.2): Process scheduling is performed by programs known in various OS's as the scheduler or the multiplexer.

Purpose: To allocate the processor among the processes.

Often there are two such programs.

• One (let us call it the multiplexer) is responsible for short-term scheduling (preemption, activation, awakening) of processes in main memory.

• The other (let us call it the scheduler) is responsible for long-term scheduling (changing priorities, suspension, resumption).

Sometimes it is called the job scheduler.
Often (in a non-virtual memory operating system),

- the scheduler determines which processes should be brought into main memory, while
- the multiplexer determines which process in main memory should be executed next.

The scheduler is invoked infrequently (seconds, minutes). It may be slow.

The multiplexer is invoked frequently (e.g., at least once every 100 ms.). It must be fast.

The scheduler controls the degree of multiprogramming.

If the degree of multiprogramming is stable, the scheduler may have to be invoked only when

On the next page is a diagram of the progress of a job that is never suspended-ready and only blocks awaiting I/O.

The multiplexer is entered——

- whenever

- whenever
We will assume that the multiplexer is entered after all

Operation of the multiplexer:

1. Is the current process still the most suitable to run?
   
   If so, return control to it.
   If not, then …

2. Save the volatile state (PC and other processor registers) in the register save area in the PCB.

3. Retrieve the volatile environment of the “most suitable” process from its PCB.

4. Transfer control to that process (at the location indicated by the restored PC).

Steps 2-4 are known as a process switch,

Which process is “most suitable” depends on the scheduling discipline, which we will soon consider.

Operations on processes (S&G, §4.3): The OS must provide a way to create and terminate processes.

Process creation: The creator process is known as the parent, and the created process is called the child.

Regarding the resources of the child process, there are several options:

- Child obtains resources directly from OS.
- Child obtains resources from
• Child _________ resources with parent.

The parent may also pass resources, such as open files, to its children.

Regarding concurrency of the child, there are two options.

•

•

Regarding the address space of the child, there are also two options.

•

•

In Unix, each process is identified by a *pid*.

• A new process is created by the *fork* system call.
  • It consists of a copy of the parent’s address space. (Parent and child can thus communicate easily.)
  • Both processes continue execution after the *fork*.
    ◦ In the child, the return code for the fork is 0.
    ◦ In the parent, the return code for the fork is

• After creation, a process’s address space is normally replaced by an *execve* system call.
  It loads an object file into memory, on top of itself, and starts its execution.

• The parent can create more children, or suspend itself, via a *wait* system call.
By contrast,

- VMS creates a new process, and loads a specified program into its address space.
- Windows/NT supports both models.

*Process termination:* When a process wants to terminate, it issues an `exit` call.

It may return data to its parent process.

All its resources are deallocated by the OS.

Generally, a parent is allowed to abort its child. Why might it do this?

- 

- 

- 

In Unix, a process issues `exit` to terminate.

A parent can `wait` for this event. `wait` returns the pid of the exiting child.

If the parent terminates, all children are terminated too.

*Threads* (S&G, §4.5): Traditionally, a process has consisted of

- a single address space, and
- a single thread of control.

Each process has its own PCB, and its own memory.

This makes it expensive to create and maintain processes.
With the advent of multiprocessors, it may be too expensive to create all the process data structures each time it is desired to execute code in parallel.

Multiprocessors come in two varieties—

- **Shared-memory** multiprocessors can have more than one processor sharing a region of memory. Processes can therefore communicate via shared memory.

- **Distributed-memory** multiprocessors usually have to perform interprocess communication by passing messages from one processor to another.

On a shared-memory multiprocessor, it is necessary to create and switch between tasks rapidly to take advantage of the available parallelism.

For example, a file server may service requests for files from many different users.

The clearest way to program a file system may well be to devote a separate “process” to each active request.

But all of these processes would have the same code, and much of the same data. Does it make sense to duplicate it?

For this reason, “lightweight processes,” or *threads*, were developed.

A thread consists of—

- a program counter,
- a register set,
- a stack of activation records, and (not mentioned in the text)

We can consider each “heavyweight” process as made up of one or more threads.
S&G use

• “task” to mean a heavyweight process, and
• “thread” to mean a lightweight process.

What is faster about using threads than using tasks?

Like processes, a thread can—

• be scheduled onto a processor,
• block while waiting for system calls, and
• create child threads.

However, unlike processes, threads are not protected from other threads.

One thread can read and write the other’s memory, even its stack.

Why doesn’t this compromise security?

Consider the file server again.

• If a file server had a single thread, what would happen when it made an I/O request on behalf of a single user?

• If the task had multiple threads, while one thread was _______, the other could ______.

Without threads, a system designer could create parallelism through multiple (heavyweight) processes. But,
There is one important decision regarding the level at which threads are implemented. They are implemented either at the

- User-level threads: A run-time library package provides the routines necessary for thread-management operations.

The libraries multiplex a potentially large number of user-defined threads on top of

What are some advantages of this approach?

- Low-cost context-switching. Why?

- Existing kernel need not be modified.

- Flexibility. Different libraries can be developed for different applications.

  One application does not incur overhead from operations it does not need.

  E.g., one library could provide preemptive priority scheduling, while another might use FIFO.

What are some limitations of user-level threads?

- A kernel call by any thread causes

  - Scheduling can be unfair. The threads in a process with many threads don’t get as much CPU attention as those in a process with few threads.
**Kernel-level threads**: Kernel-level threads have their own advantages.

- Easy to coordinate between the synchronization and scheduling of threads, since the kernel has information concerning the status of all threads.

- They are suitable for applications such as server processes, where interactions with the kernel are frequent.

However,

- Thread-management operations are more expensive than with user-level threads.
- Since the kernel implements threads, it has to provide any feature needed by any reasonable application. This may mean greater overhead, even for applications that do not use all the features.

**Example: Solaris 2**. Has a library for user-level threads.

Kernel “knows nothing” of these threads.

Solaris 2 implements—

- user-level threads,
- lightweight processes, and
- kernel-level threads.

Each task contains at least one LWP.

User-level threads are multiplexed onto LWPs, and can only do work when they are being run on an LWP.

All operations within the kernel are executed by kernel-level threads.

- Each LWP has a kernel-level thread.
- Some threads (e.g., a thread to service disk requests) are not associated with an LWP.
Some kernel-level threads (e.g., device drivers) are tied to a specific processor, and some are not.

A system call can “pin” any other thread to a specific processor, too.

Kernel intervention is not required to switch between user-level threads.

User-level threads can execute concurrently until they need kernel services. Then they must obtain an LWP, and will be blocked if they cannot.

However, a task need not be blocked waiting for I/O to complete. If it has multiple LWPs, others can continue while one is blocked.

Even if all the task’s LWPs are blocked, user-level threads can continue to run.

**Interprocess communication** (S&G §4.6): There are two ways in which processes may communicate.

- Shared memory.
- Message passing.

Message-passing is a higher-level interprocess-communication mechanism because the operating system is responsible for enforcing synchronization.

Two basic operations:

- *Send* a message *to* a process.
- *Receive* a message *from* a process.

There are several “degrees of freedom” in designing message systems:

- How are links established?
- Are links unidirectional or bidirectional?
- Can a link be accessible to more than two processes?
- How much (if any) buffer space is associated with a link?
- Are messages fixed size or variable size?
One option is for processes to send messages “directly” to each other.

For example, in the *producer/consumer problem*, the processes might communicate like this:

```plaintext
producer:
    while ... do begin
        ... produce an item in nextp;
        ... send(consumer, nextp);
    end;

c consumer:
    while ... do begin
        receive(producer, nextc);
        ... consume the item in nextc;
    end;
```

S&G calls this “direct communication.” What are some shortcomings of this scheme?

- 

- 

In most systems, message operations instead name *message buffers*, usually known as *mailboxes* or *ports*.

- `send(A, message)` — send a *message* to mailbox A.
- `receive(A, message)` — receive a *message* from mailbox A.
S&G calls this “indirect communication.”

A process that creates a mailbox (e.g., via a call to the memory manager) is the owner of the mailbox. Generally, it is the owner’s responsibility to allow other processes to use the mailbox.

However, the operating system may also create mailboxes; these mailboxes can outlive individual processes.

In some systems, only two processes can share a mailbox—one sender and one receiver.

Some message systems allow more than two processes to share a mailbox.

- Regardless of how many processes share a mailbox, only one may update it (by inserting or removing a message) at a time.
- Usually messages are buffered in FIFO order. A process that does a receive retrieves the oldest message in the mailbox.

Similarly, two processes may share more than one mailbox, if they wish.

Mailbox capacities (S&G §4.6.3): Messages which have been sent but not yet received are buffered in the mailbox.

The capacity determines the number of messages that may be buffered at a time.

- Unbounded capacity. Mailbox size is potentially “infinite” -- can grow to occupy all of primary and secondary storage, for example.
  
  Thus, mailbox is never “full”; it is always possible for the sender to place another message in it.

- Bounded capacity. A mailbox has a finite size; can get full. If a send is performed to a full mailbox, special action is necessary.

- Zero capacity. The mailbox can’t hold any messages. Special action (described below) is always necessary. This kind of interaction is called a rendezvous.
In the zero-capacity case, a sending process knows that the receiver got the message.

In the other cases, a sending process can’t be sure whether its message has been received.

Thus, if the sending process \( P \) wants to be sure, the receiving process \( Q \) needs to send an acknowledgment message.

Thus, process \( P \) executes

\[
\text{send}(mbox, \text{message});
\text{receive}(\text{ackbox}, \text{reply});
\]

Process \( Q \) executes

\[
\text{receive}(mbox, \text{message});
\text{send}(\text{ackbox}, \text{reply});
\]

Why are two mailboxes needed?

In the Thoth operating system, there was a \texttt{reply} primitive as well as \texttt{send} and \texttt{receive} operations.

- A process that did a \texttt{send} was delayed until the \texttt{reply} was received.
- The \texttt{reply} overwrote the message that had been sent.
- The only difference between \texttt{send} and \texttt{reply} was that the latter did not block the sender.

This is actually a special case of \textit{remote procedure call} (RPC). In a remote procedure call,

- the caller sends a message to the callee, passing the parameters in the message body;
- the receiver performs some procedure, and then sends a message back, returning the result (if there is one) in the message.

The caller waits for the RPC to be completed. This is just what happens in an ordinary procedure call, where the caller does not continue until the callee is finished.
Synchronization: Special action is necessary

- when a process attempts to receive from an empty mailbox, or
- when a process attempts to send to a full mailbox.

Some systems allow up to three kinds of special actions:

- the receiving (sending) process is blocked until a message arrives (is removed).
- a special error code is returned that means “try again later.”

- the receiving (sending) process is placed on a queue to be notified when a message (space) becomes available, although it is not blocked in the meantime.

Example: Assume the “special action” is to block a process.

Mailbox empty
Ernie sends
Now one message in mailbox.
John receives
The message sent by Ernie.
Joe receives
No message there \( \Rightarrow \) Joe is blocked.
Mike sends
Joe receives message and is unblocked.

Exception conditions (S&G §4.6.3): Process terminated. Suppose that a process \( P \) sends or receives a message from a process \( Q \) that has terminated. What happens …

- if \( P \) tries to receive a message from a process that has terminated?

- if \( P \) sends a message to a process that has terminated? It depends on whether the mailbox is
  - If so,
° If not,

Here too, the system should either terminate $P$ or notify $P$ that $Q$ has terminated.

Lost messages. If it is important to detect lost messages, either the OS or the sending process may take on this responsibility.

One way of detecting lost messages is via timeouts: Require a reply message, and if it is not received “soon enough,” assume that it is lost.

What problem could this cause if the message is not really lost?

Message passing by value vs. message passing by reference:
Similar to parameter passing in Pascal.

• pass by value: copy the entire message (which may be very long).
• pass by reference: pass a pointer to the message.

If pass-by-value is used,

• mailbox has to be bigger, and probably has to contain a length indication for each message
• a lot of work may be wasted if the recipient only refers to a few bytes of the message

If pass-by-reference is used,
The Mach message system (S&G §4.6.4): Mach is a distributed extension of Unix developed at Carnegie Mellon University.

In Mach, all inter-task communication is done via messages, which are sent to ports. System calls are also performed through messages.

Two ports are created for each task.

- The *kernel* mailbox is used by the task to communicate with the kernel.
- The *notify* mailbox is where the kernel sends notification of event occurrences.

There are three message operations: `msg_send`, `msg_receive`, and `msg_rpc`.

A message has a fixed-length header and a variable-length body.

The body is a list of typed data items. Different types of data items that may be sent include—

- access rights,
- task states, and
- memory segments.

Sending a “segment of memory” is a way to avoid copying a message twice:

- This double copying is a major reason why many message systems are slow.

If a mailbox is full, the sending thread has four options.

- Wait indefinitely until there is room in the mailbox.
- Wait at most $n$ milliseconds.
• Do not wait at all, but return immediately.
• Temporarily cache a message “under a rock.” The “rock” can hide only one message.

The last option allows a time-critical process to continue even if the target port is full.

This allows, e.g., a line-printer driver to inform a task that its request has been performed, even if the task’s port is full.

Why is it important to do this?

Scheduling disciplines: (different disciplines may be used by the low-level and the high-level schedulers.)

Criteria for scheduling disciplines: Scheduling disciplines can be evaluated by according to how well they meet the following goals.

1. Maximize processor utilization. The processor should be kept as busy as possible.

2. Maximize throughput. Throughput is the number of processes that complete per time unit.

3. Minimize turnaround time. Turnaround time = time between
   • a request for process creation, and
   • process termination.

A good scheduling strategy attempts to minimize average turnaround time (over a set of processes).

4. Minimize response time. Response time = time between
   • a request for process creation, and
   • when the first response is produced.

This is important in
Scheduling disciplines may be either preemptive or non-preemptive.

- With a non-preemptive scheduling discipline, once a processor is given to a process (sometimes called a “job”), the job continues to use the processor until it finishes using the processor.

- With a preemptive scheduling discipline, a process may have the processor taken away from it by a “more deserving” process.

Let us now consider several specific scheduling disciplines.

**FCFS scheduling** (First-come, first-served. Also known as FIFO.)

The simplest strategy to implement. Occasionally the most appropriate.

- Jobs are scheduled according to time of arrival.
- All jobs run to completion (no preemption).

This policy

- requires small jobs to wait for large jobs,
- but gives waiting times with a low variance.

**Example**:  

<table>
<thead>
<tr>
<th>Process</th>
<th>Processing time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>24</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
</tr>
</tbody>
</table>

Suppose the processes arrive in the order P1, P2, P3.

Here is a *Gantt chart* of the execution of these processes.

![Gantt chart](chart.png)

Turnaround times:

- Average turnaround time:
**LCFS scheduling** (last-come, first-served).

- Every arrival changes the choice.
- Short jobs get very good service, but long jobs are postponed indefinitely.

How would LCFS work on our example from above (assuming all 3 jobs arrived by time 0)?

![Turnaround times: Average turnaround time:](image)

Why did this schedule perform so much better than the previous one?

This experience suggests the following discipline:

**SJN scheduling** (shortest job next).

- Non-preemptive.
- Chooses waiting process with smallest estimated "run-time-to-completion."

Reduces average waiting time over FCFS, but waiting times have larger variance.

Favors short jobs at the expense of long jobs.

Less useful in a time-sharing environment.

**SRT scheduling** (shortest remaining time)

Preemptive counterpart of SJN

A running process may be preempted by a new process with a shorter estimated time.
SRT minimizes average waiting times over a set of jobs.

**RR scheduling** (round-robin).

- The most common low-level scheduling discipline.
- Effective for time-sharing.
- Must choose a *quantum*, or time slice. The process is preempted when the quantum expires, and placed at the end of the ready queue.

![Diagram of RR scheduling]

*Note:* A quantum is *not* a fixed time slot. Each job gets a full quantum, regardless of whether the previous job used up its quantum.

The processor is effectively shared among the jobs.

A short job is not forced to wait for completion of longer jobs ahead of it.

**Choice of quantum size**:

- Short quanta ⇒
- Long quanta ⇒

“Rules of thumb”:

- Quantum = \(100 \times (\text{preemption overhead})\)
- 80% of CPU bursts should be shorter than quantum.

(What is a CPU burst?)

Usually, a quantum \(q\) is between 10–100 msec.
Notice that if there are $N$ processes in the ready queue, then each process gets $1/N$ th of the processor time in chunks of at most $q$ time units at once.

No process waits more than ____________ time units before its next chunk.

**PS scheduling** (processor-sharing). Cannot be implemented, but a useful way of modeling RR.

PS is defined as

$$\lim_{\text{Quantum} \to 0} \text{RR} \quad (\text{RR with zero overhead})$$

(What is $\lim_{\text{Quantum} \to \infty} \text{RR}$?)

This is a reasonable approximation of RR when the quantum is

- Large with respect to overhead.
- Small with respect to mean CPU burst.

Special definition of waiting time for PS discipline:

Waiting time = Completion time –

**Priority disciplines** are scheduling disciplines where each process is assigned a priority number (e.g., an integer).

The processor is allocated to the process with the highest priority.

SJN is a priority discipline where the priority is the (predicted) processing time.

**Problem**: Starvation—low-priority processes may never execute.

**Solution**: Aging—increase the priority as time progresses.
Example:

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival time</th>
<th>Processing time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>P3</td>
<td>4</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

(High priority number = high priority.)

Let's draw a Gantt chart:

0 4 8 12 16 20 24 28

Multilevel priority queues: Several queues; highest is RR, and all the rest are FCFS.

- A job arrives initially at the level-1 queue.
- Each time a job is selected for processing, it is taken from the lowest non-empty queue.
- A job may move between the queues.
- The quantum usually increases as the job moves to higher levels.
Example:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Time quantum</th>
<th>msec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>RR—quantum</td>
<td>24</td>
</tr>
</tbody>
</table>

» A new arrival enters queue 1, which is served FCFS.
» When it arrives at the processor, it gets 8 msec.
» If it does not finish in 8 msec., it is moved to queue 2.
» Again it is served FCFS and gets 16 more msec.
» If it does not finish, it moves to queue 3 where it is served RR.

- Favors very short jobs even more than RR.
- Sometimes disciplines other than FCFS and RR may be used at different levels.
- To use this discipline, several parameters need to be defined:
  - The number of queues.
  - The scheduling discipline within each queue.
  - When do jobs move up to another queue?
  - When (if ever) do they move down?
  - Which queue is entered by a newly arriving job?

Multithreaded scheduling: Scheduling is more complex when multiple processors are available.

- When can a particular process be executed by any processor in the system? When—
  - 
  - 
  - 
• The possibility of load-sharing or load-balancing arises.
  ◦ The processors can use a common ready queue.
  ◦ The OS must be programmed so that only one processor at a time updates the queue.

• Two kinds of control.
  ◦ Asymmetric multiprocessing.
  ◦ Symmetric multiprocessing.

Real-time scheduling (S&G, §5.5): Real-time systems are divided into two types.

• Hard real-time systems are required to complete a critical task within a guaranteed amount of time.

• Soft real-time systems require that critical processes receive priority over other processes.

Hard real-time systems may employ deadline scheduling.

This guarantees different job classes (e.g. real-time processes, interactive users, batch jobs) a certain fraction of processing time.

• Each job class is guaranteed a certain fraction $f_i$ of processor time within each duty cycle.

• Each job in the class may be allocated some portion of this $f_i$.

• If $\sum f_i \leq 1$, then each process is guaranteed to receive its quota of processing time in every cycle of ready time.

What characteristics of a system might make it impossible to honor these guarantees?
Soft real-time systems

- have priority scheduling, with
- real-time processes having the highest priority.

The dispatch latency must be small.

How to lower the dispatch latency?

Problem is that many OSs have to wait for a system call to complete or an I/O block to occur before doing a context switch.

Therefore, system calls must be made preemptible by—

- inserting *preemption points* in long-duration system calls. At each preemption point, the system checks whether a high-priority process needs to execute. Sometimes it is difficult to find enough places to add preemption points.

- making the entire kernel preemptible. Then all kernel data structures must be protected by synchronization mechanisms.

What happens if the higher-priority process needs to access data structures that are currently being accessed by a lower-priority process?

This is called *priority inversion*.

In fact, there could be a chain of processes preventing access to the data structure.

This problem can be attacked with the *priority-inheritance protocol*. What do you think it does?
Let us list the components of the response interval of a real-time process.

- 
- 
- 
- 

Other considerations in scheduling: A job which uses much more processor time than I/O is said to be *CPU bound*.

A job which does much more I/O than processing is called *I/O bound*.

A scheduler should try to activate some CPU-bound and some I/O-bound jobs at the same time.
**Process synchronization:** A computer system may have *concurrent* processes because of—

- multiprogramming (multitasking)—one process may be started before another one finishes, and
- multiprocessing—there may be more than one (central) processor and I/O devices.

Until now, we have assumed that a program in execution is a single process.

In almost any program, some activities can be carried out concurrently or in arbitrary order.

Thus we may have concurrent processes within a single program.

Consider this short program:

\[
\begin{align*}
\text{read} (a); \\
\text{short program:} \\
\text{b} & := \sin(2.0); \\
\text{c} & := a + b; \\
\text{write} (c); \\
\end{align*}
\]

It doesn’t matter which order the first two statements are executed in. They could even be executed concurrently.

A *precedence graph* shows the order in which activities can execute.

A precedence graph is a directed *acyclic* graph whose nodes correspond to individual statements or program fragments.
We must unfold loops to create a precedence graph.

In a precedence graph, an edge from $P_i$ to $P_j$ means that $P_i$ must execute before $P_j$.

Here is a more complicated precedence graph:

$P_2$ and $P_3$ can be executed after $P_1$ completes.

$P_4$ can be executed after $P_2$ completes.

$P_7$ can only execute after $P_5$, $P_6$, and $P_3$ complete.

$P_3$ can be executed concurrently with $P_2$, $P_4$, $P_5$, and $P_6$.

**Precedence relations**: A precedence graph illustrates a precedence relation.

Let $P = \{P_1, P_2, \ldots, P_n\}$ be a set of cooperating processes.

The execution order of $P$ is defined by a relation $\Rightarrow$, a set of ordered pairs from $P \times P$:

$$\Rightarrow = \{(P_i, P_j) \mid P_i \text{ must complete before } P_j \text{ can start}\}$$

$P_i \Rightarrow P_j$ means $(P_i, P_j) \in \Rightarrow$.

"$P_i$ is a direct predecessor of $P_j$;"  

If $\exists$ a sequence $P_i \Rightarrow P_j \Rightarrow \ldots \Rightarrow P_k$, we say

"$P_i$ is a predecessor of $P_k$;"  

"$P_k$ is a successor of $P_i$;"

Two processes are independent if neither is a predecessor of the other.

A redundant edge in a precedence graph is an edge $(P_i, P_j)$ such that there is a longer path between $P_i$ and $P_j$;
Precedence relations may include redundant edges; precedence graphs should not.

What is a possible precedence relation for the precedence graph on the previous page?

\[ P = \{ P_1, P_2, P_3, P_4, P_5, P_6, P_7 \} \]

\[ \Rightarrow = \]

*Mutually noninterfering systems:* A system is called *determinate* if the sequence of values written in each memory cell is the same for any order of execution allowed by \( P \).

How can we decide if a system of processes is determinate? Let us define—

- **Read set** \( \equiv \) set of variables read from memory. More precisely, the read set \( R(P_i) \) is the set of variables referenced during the execution of process (statement) \( P_i \) that were not written by \( P_i \).

- **Write set** \( \equiv \) set of variables written to memory. The write set \( W(P_i) \) is the set of variables referenced during the execution of process \( P_i \) that were written by \( P_i \).

For the program at the start of the section, what is—

\[
\begin{align*}
R(\text{read} \ (a)) \\
R(c := a + b) \\
W(\text{read} \ (a)) \\
W(c := a + b)
\end{align*}
\]

Let \( P_i \) and \( P_j \) be two processes (statements). \( P_i \) and \( P_j \) can be executed concurrently if and only if—

\[
\begin{align*}
R(P_i) \cap W(P_j) &= \emptyset \\
W(P_i) \cap R(P_j) &= \emptyset \\
W(P_i) \cap W(P_j) &= \emptyset
\end{align*}
\]
These are called *Bernstein's conditions*.

Two processes are mutually noninterfering—

- if

- if

A mutually noninterfering system of processes is determinate.

*Maximally parallel systems*: It is important for systems to be determinate. Why?

But, it is not good to over-specify precedence constraints either. Why?

A system is called *maximally parallel* if the removal of any pair \((P_i, P_j)\) from \(\Rightarrow\) makes \(P_i\) and \(P_j\) interfering.

Given a system of processes, how can we construct a maximally parallel system?

The maximally parallel system equivalent* to a determinate system \(P\) is formed by replacing the original precedence relation \(\Rightarrow\) with a new precedence relation \(\Rightarrow'\) defined by—

\((P_i, P_j)\) is in \(\Rightarrow'\) iff

- \(P_i\) is a *predecessor* of \(P_j\) in \(\Rightarrow\), and
- \(P_i\) and \(P_j\) do not satisfy Bernstein's conditions.

*What does “equivalent” mean?
Specification of concurrency:

**fork and join constructs**: The earliest (and lowest-level) language construct for specifying concurrency.

*Syntax*: **fork** $L$;

- Starts an independent process at the label $L$. The computation that executed the **fork**, however, continues execution at the instruction that follows the **fork**.

- Splits an instruction stream into two streams that can be executed simultaneously by independent processors.

- Like a **goto** that is taken by one processor and not taken by the other.

*Syntax*: **join** $n$;

- Inverse of **fork**; brings independent instruction streams together. Causes $n$ independent streams to merge.

- The execution of instructions following a **join** $n$ instruction will not take place until $n$ independent processes have executed the **join** $n$ instruction.

- A process blocks until enough others have executed the **join**. Then all but the last process terminate. The execution of **join** $n$ has the following effect:

  $$n := n - 1;$$
  $$\text{if } n \neq 0 \text{ then quit;}$$

- The place where a **join** occurs is known as a *synchronization point*.

Here are precedence graphs for **fork** and **join**:

![Precedence graphs for fork and join](image-url)
Let’s use fork and join to code our examples:

```plaintext
count := 2
fork L1 ;
read (a);
goto L2 ;
L1 : b := sin(2.0);
L2 : join count ;
c := a + b ;
write (c);

fork L1 ;
count := 3;
fork L1 ;
goto L2 ;
P1 ;
count := 3;
L2 : b := sin(2.0);
P1 ;
count := 3;
P2 ;
c := a + b ;
P4 ;
write (c);
L2 :
goto L3 ;
P5 ;
L1 :
L3 : join count ;
```

The fork and join constructs are inconvenient to use because they force the programmer to specify control flow at a very low, goto-like level.

*The parbegin ... paren* construct:

```plaintext
parbegin P1 ; P2 ; ... ; Pn paren;
```

All statements enclosed between parbegin and paren can be executed concurrently.

This statement implements the precedence graph—
We can now redo our examples using this statement. Note that \texttt{parbegin \ldots parend} may be nested.

\begin{verbatim}
parbegin
  read (a);
  b := sin(2.0);
parend;
c := a+b;
write (c);
\end{verbatim}

\begin{verbatim}
parbegin
parend;
parend;
parend;
\end{verbatim}

There are some precedence graphs that the \texttt{parbegin \ldots parend} construct cannot implement. Can you cite an example?

On the other hand, any

\begin{verbatim}
parbegin P_1; P_2; \ldots ; P_n parend
\end{verbatim}

construct can be simulated using \texttt{fork/join}:

\begin{verbatim}
count := n;
fork L2;
fork L3;
  : 
fork Ln;
P_1: goto LL;
L2: P_2: goto LL;
L3: P_3: goto LL;
  : 
L_n: P_n;
LL: join count;
\end{verbatim}
Concurrent tasks (§6.1 of S&G): A task is a sequence of actions performed in sequential order. There are several advantages to defining concurrent tasks in a program:

- Concurrent execution on multiple processors.
- Theoretical performance improvement on a single processor.
- Simpler solutions.

How do concurrent tasks communicate and synchronize with each other? There are two basic ways:

- Via shared memory:

  ![Shared Variables Diagram]

- Via message passing:

  ![Message Passing Diagram]

Race conditions and their avoidance: It is rarely possible to decompose programs into tasks which share no information.

Bernstein’s conditions ⇒ must strictly control order in which shared variables are read and updated.

Example:

```
j := 10;
parbegin
  write(j);
  j := 1000;
parend;
```

A race condition occurs when the outcome of a computation depends on the relative speeds of its tasks.

One of the simplest examples of a race condition occurs when two tasks are attempting to update the same variable:

```
p1: count := count +1;  p2: count := count –1;
(1a)  R1 := count;  (2a)  R2 := count;
(1b)  R1 := R1 +1;  (2b)  R2 := R2 –1;
(1c)  count := R1;  (2c)  count := R2;
```
Which order of execution of these sequences will give an incorrect result?

To prevent accidents like this, we must assure that only one process at a time accesses shared variables. This is called the *mutual exclusion* problem.

A region of code that can be executed by only one process at a time is called a *critical section*.

In general, we need to implement

```
while ... do
begin
  ... ; { Non-critical section }
  mutexbegin
    ... ; { Critical section }
  mutexend;
  ... ; { Non-critical section }
end;
```

*mutexbegin* is called the *entry* code and *mutexend* the *exit* code.

To provide a general solution, our code must satisfy three constraints:

1. *Mutual exclusion*: If one process is executing a critical section, then no other process can be executing that critical section.

2. *No mutual blocking*: When a process is not in its critical section, it may not prevent other processes from entering their critical sections.

When a decision must be made to admit one of the competing processes to its critical section, the selection cannot be postponed indefinitely.

3. *Fairness*: If process \( p_i \) wants to enter its critical section, other processes must not be allowed to enter their critical section arbitrarily often before \( p_i \) is allowed to enter.
How do we implement `mutexbegin` and `mutexend`?

**mutexbegin**: Check whether there is any other process in critical section. If so, entering process must wait.

When no other process is in a critical section, the process proceeds past `mutexbegin`, setting an indicator so that other processes reaching a `mutexbegin` will wait.

**mutexend**: Must allow a waiting process (if there is one) to enter its critical section.

Seems easy! Just use a flag called `occupied` (initialized to false) to indicate that no process is in a critical section. The code for each process looks like this:

```pascal
while ... do
begin
  ... ; { Non-critical section }
  while occupied do begin end
  occupied := true;
  ... ; { Critical section }
  occupied := false;
  ... ; { Non-critical section }
end;
```

What’s wrong with this solution?

A correct solution is surprisingly difficult to achieve. (See the development in S&G, pp. 159–162.)

The correct solution uses—

- A shared variable `turn` that keeps track of which process’s turn it is to enter the critical section.

- A shared array `flag` that keeps track of which process is in the critical section:

```pascal
var flag : array [0 .. 1] of boolean;
```

Initially, elements of `flag` are `false` and `turn` is either 0 or 1.
Peterson’s algorithm:

```plaintext
while ... do
begin {Non-critical section }
  flag [i] := true ;
  j := (i +1) mod 2;
  turn := j;
  while flag[j] and turn = j
    do begin end;
  ... ; { Critical section }
  flag[i] := false ;
  ... ; {Non-critical section }
end;
```

We want to prove three things about this algorithm:

(a) Mutual exclusion.

(b) No mutual blocking.

(c) Fairness.

Proof:

(a) Mutual exclusion is preserved: If it were not, and both processes ended up in their critical section, we would have flag [0] = flag [1] = true.

Both processes could not have passed their inner while loops at the same time, because turn would have been favorable to only one of the processes.

This implies that after one process (say p₀) entered, turn was later set to a favorable value for the other process (p₁). But this can’t happen. Why?

Hence p₁ cannot enter its critical section until p₀ leaves, so mutual exclusion is preserved.
(b) Mutual blocking cannot occur: Consider $p_1$. It has only one wait loop (the inner \textbf{while}) loop. Assume it can be forced to wait there forever. After a finite amount of time, $p_0$ will be doing one of three general things:

(i) not trying to enter,
(ii) waiting at its inner while loop, or
(iii) repeatedly cycling through its critical & non-critical sections.

In case (i), $p_1$ can enter because $\text{flag}[0]$ is false. Case (ii) is impossible, because $\text{turn}$ must be either 0 or 1, so both processes can’t be waiting at once. In case (iii), $p_0$ will set $\text{turn}$ to 1 and never change it back to 0, allowing $p_1$ to enter.

(c) Fairness. After it indicates a desire to enter the critical section by setting $\text{flag}[i]$ to one, neither process can be forced to wait for more than one critical-section execution by the other before it enters.

Assume that $p_0$ is in its critical section and $p_1$ wishes to enter. Before $p_0$ can enter a second time, it sets $\text{turn}$ to 1, which prevents it from entering before $p_1$ enters and completes its critical section.

More than two processes: the bakery algorithm: Based on each process “taking a number” when it wants to enter its critical section.

The process with the lowest number is “served” next.

However, two processes may receive the same number. In this case, the tie is broken in favor of the process with the lowest pid.

The common data structures are

\begin{verbatim}
  type  process = 0..n-1;
  var   choosing: array [process ] of boolean;
        number: array [process ] of integer;
\end{verbatim}

Define $(a, b) < (c, d)$ if $a < c$

or if $a = c$ and $b < d$. 
Here is the code:

```plaintext
while ... do
begin
  choosing[i] := true;
  number[i] := max(number[0], number[1], …, number[n−1]) + 1;
  choosing[i] := false;
  for j := 0 to n−1 do
    begin
      while choosing[j] do begin end;
      while number[j] ≠ 0
        and (number[j], j) < (number[i], i)
      do begin end;
    end;
... ; { Critical section }
number[i] := 0;
... ; { Non-critical section }
end;
```

To show that the algorithm is correct, we need to show that if a process arrives while another process \( P_i \) is in its critical section, the new process gets a number—actually, a (number, pid) pair—that is higher than process \( P_i \)'s (number, pid) pair.

Once we have shown that, the proof follows easily.

Let \( P_k \) be the new process that arrives while \( P_i \) is in its critical section. When it executes the second `while` loop for \( j = i \), it finds

\[
number[i] ≠ 0 \quad \text{and} \quad (number[i], i) < (number[k], k).
\]

Thus, it continues looping until \( P_i \) leaves its critical section.

Since the processes enter their critical section FCFS, progress and bounded waiting are guaranteed.

**Hardware assistance with synchronization:** Purely software synchronization is

- somewhat complicated
- expensive

Yet hardware assistance can be provided relatively easily.
Some instruction sets include a \textit{swap} instruction, which provides the following indivisible action:

\begin{verbatim}
procedure swap (var a, b : boolean );
var temp : boolean ;
begin
  temp := a ;
  a := b ;
  b := temp ;
end;
\end{verbatim}

Here is a solution to the critical-section problem using \textit{swap}:
A global boolean variable \textit{lock} is initialized to \textit{false}. \textit{key} is a local variable in each process.

\begin{verbatim}
while ... do
begin
  ... ; \{ Non-critical section \}
  key := true ;
  repeat
    swap (lock , key )
  until
  ... ; \{ Critical section \}
  lock := false ;
  ... ; \{ Non-critical section \}
end;
\end{verbatim}

Many other instruction sets include a \textit{test_and_set} instruction, which performs the following \textit{indivisible} action:

\begin{verbatim}
function test_and_set (var target : boolean ): boolean ;
begin
  test_and_set := target ;
  target := true ;
end;
\end{verbatim}

In other words, it returns the previous value of \textit{target}, and regardless of what that value was, leaves \textit{target} true.

With \textit{test_and_set}, it’s trivial to implement \textit{mutexbegin} and \textit{mutexend}. Let’s rewrite our sample critical-section program.

\begin{verbatim}
while ... do
begin
  ... ; \{ Non-critical section \}
\end{verbatim}
This solution allows starvation. Why?

A complete solution is given on the next page.

To provide a starvation-free critical-section implementation using \texttt{test\_and\_set}, we declare these global variables:

\begin{verbatim}
var waiting : array [0..n-1] of boolean ;  
lock : boolean ;
\end{verbatim}

These variables are initialized to \texttt{false}. Each process has the following local variables:

\begin{verbatim}
var j : 0..n-1;  
key : boolean ;
\end{verbatim}

Here is the code:

\begin{verbatim}
while ... do 
begin 
\hspace{1em} ... ; \{ Non-critical section \} 
\hspace{1em} waiting [i] := true ;  
\hspace{1em} key := true ;  
\hspace{1em} while waiting [i] and key do 
\hspace{2em} key := test\_and\_set (lock );  
\hspace{2em} waiting [i] := false ;  
\hspace{2em} ... ; \{ Critical section \} 
\hspace{1em} j := (i +1) \mod n ;  
\hspace{1em} while (j \neq i \ and \ not waiting [j]) do 
\hspace{2em} j := (j+1) \mod n ;  
\hspace{2em} if j = i then lock := false 
\hspace{2em} else waiting [j] := false ;  
\hspace{2em} ... ; \{ Non-critical section \} 
end;
\end{verbatim}

A proof that this solution satisfies the three requirements is given in Peterson & Silberschatz (pp. 161–162).
• In general, processes are allowed inside the critical section in cyclic order (0, 1, …, n–1, 0, …) as long as any process wants to enter.

• If no process wants to enter, lock is set to false, allowing the first process that decides to enter to do so.

On a uniprocessor, instead of using test_and_set, one can merely disable interrupts.
Semaphores (§6.4 of S&G): For simple problems, test_and_set is fine, but it doesn’t work well for more complicated synchronization problems.

A semaphore is a more flexible synchronization mechanism.

There are two kinds of semaphores:

- **Binary semaphores**: Can only take on values 0 and 1.
- **Counting semaphores**: Can assume integer values.

Unless otherwise stated, we will be considering counting semaphores.

Two indivisible operations are defined on a semaphore \( s \) (of either kind):

- The \( P \) operation, \( P(s) \):

  \[
  s := s - 1; \\
  \text{if } s < 0 \text{ then wait till some process performs a } V(s); \\
  \]

  [Alternate definition for binary semaphore

  \[
  \text{if } s > 0 \text{ then } s := s - 1 \\
  \text{else } \text{ wait till some process performs a } V(s); \\n  \]

- The \( V \) operation, \( V(s) \):

  \[
  s := s + 1; \\
  \text{if process(es) are waiting on } s \text{ then let one of the waiting processes proceed; } \\
  \]

If several processes attempt a \( P(s) \) on a binary semaphore simultaneously, only one will be able to proceed.

For waiting processes, any queueing discipline may be used (e.g., FCFS).
Note: If \( s < 0 \), its value is the negative of the number of processes that are waiting.

Critical-section implementation is (even more) trivial:

\[
\begin{align*}
\textbf{while} & \quad \ldots \quad \textbf{do} \\
\textbf{begin} & \quad \ldots \quad \{ \text{Non-critical section} \} \\
& \quad \ldots \quad \{ \text{Critical section} \} \\
& \quad \ldots \quad \{ \text{Non-critical section} \} \\
\textbf{end;} 
\end{align*}
\]

What should be the initial value of \( \text{mutex} \)?

By executing a \( P \) operation on a semaphore \( \text{sleeper\_priv\_sem} \), a process can block itself to await a certain event (a \( V \) operation on \( \text{sleeper\_priv\_sem} \)):

\[ \text{sleeper process:} \quad \text{awakener process:} \]
\[ \{ \text{Compute} \} \quad \{ \text{Compute} \} \]
\[ P(\text{sleeper\_priv\_sem}); \quad V(\text{sleeper\_priv\_sem}); \]
\[ \{ \text{Compute} \} \quad \{ \text{Compute} \} \]

What should be the initial value of \( \text{sleeper\_priv\_sem} \)?

Semaphores can be implemented—

- in software, using busy-waiting.
- in software, using queues
- with hardware assistance, using queues.

◊ Here is the busy-waiting implementation:

\[
\begin{align*}
\textbf{procedure} & \quad P (s: \text{semaphore}) ; \\
\textbf{var} & \quad \text{blocked} : \text{boolean} ; \\
\textbf{begin} & \quad \text{blocked} := \text{true} ; \\
& \quad \textbf{while} \quad \text{blocked} \quad \textbf{do} \\
& \quad \textbf{begin} 
\end{align*}
\]
mutexbegin;
  if s.val > 0 then
  begin
    s.val := s.val – 1;
    blocked := false
  end
mutexend;
end;
end;

procedure V(s: semaphore);
begin
  mutexbegin;
  s.val := s.val + 1;
  mutexend;
end;

Notice that this results in a haphazard queueing discipline.

◊ If the queue implementation is used, a semaphore is declared as

```plaintext
type semaphore = record
  val: integer;
  q: queue;
end;
```

In overview, the code (for binary semaphores) is

```plaintext
procedure P(s: semaphore);
begin
  mutexbegin;
  if s.val > 0 then
    s.val := s.val – 1
  else
    add this process to semaphore queue and suspend it
  mutexend;
end;
```
procedure \( V(s: \text{semaphore}) \); begin
  \[ \text{mutexbegin}; \]
  if \( s.q = \text{nil} \) \{ Empty queue \} then \( s.val := s.val + 1 \)
  else remove a process from \( s.q \) and resume it
  \[ \text{mutexend}; \] end;

Busy-waiting is still needed, but it is limited to the implementation of \text{mutexbegin}.

◊ With hardware assistance, \( P \) and \( V \) would \textit{indivisibly}

- perform a decrement (or increment), and
- test the semaphore value and if necessary, perform queue manipulations to add or delete a process from the semaphore queue, then exit to the scheduler.

Regardless of method used, \( P \) and \( V \) operations have to be part of the kernel because—

- They must be available to \textit{all} processes (without a process switch), and hence must be implemented at a low level.
- A \( P \) operation may result in a process being blocked \( \Rightarrow \) needs authority to invoke the scheduler.
- The \( V \) operation must be accessible to the interrupt routines.

**Deadlocks and starvation:** If two processes are waiting on semaphores, it is possible for each to be waiting for an event that can only be generated by the other.

Thus, both processes will

They are considered to be \textit{deadlocked}.

Consider a system with two processes, \( P_0 \) and \( P_1 \), each accessing two semaphores that are initialized to 1:
How does the deadlock occur? What order must the events occur in?

We say that a set of processes is deadlocked when each process in the set is waiting for an event that can be caused only by another process in the set.

A similar problem is starvation, where processes wait indefinitely just to acquire the semaphore.

It can occur, for example, if queueing for the semaphore is LIFO (or haphazard).

**Classical process coordination problems** (§6.5 of S&G).

*Resource allocation:* A counting semaphore can be used to designate how much of a resource is available.

```plaintext
var priv_sem: array [0 . . n–1] of semaphore;
mutex: semaphore;
blocked: boolean;
```
procedure request(i: process; j: ResType);
begin
  blocked := false;
P(mutex);
  if avail[j] = 0 then
  begin
    enqueue(i, j);  { Enqueue process i to wait for resource j }
    blocked := true;
  end
  else avail[j] := avail[j] – 1;
  V(mutex);
  if blocked then P(priv_sem[i]);
end;

procedure release(j: ResType);
begin
  P(mutex);
  avail[j] := avail[j] + 1;
  if k := dequeue(j)  {I. e., if a process \( k \) is waiting for resource \( j \) } then
  begin
    avail[j] := avail[j] – 1;
    V(priv_sem[k]);
  end;
  V(mutex);
end;

The bounded-buffer problem : Assume a buffer pool has \( n \) buffers, each capable of holding one item.

Items are deposited into buffers by process producer and removed from buffers by process consumer. Mutual exclusion must be provided for the buffer pool.

Semaphores are used to—

- provide mutual exclusion for the buffer (binary semaphore mutex);
- keep track of how many buffers are empty (counting semaphore);
- and keep track of how many buffers are full (counting semaphore).
type item = ...;
var buffer: ...;
    full, empty, mutex: semaphore;
    nextp, nextc: item;
begin
    full := 0; { 0 of buffer slots are full }
    empty := n; { n of buffer slots are empty }
    mutex := 1;
parbegin
    producer:
        while ... do
        begin
            ...
            produce an item in nextp;
            ...
            P (empty);
            P (mutex);
            add nextp to buffer;
            V (mutex);
            V (full);
        end;

    consumer:
        while ... do
        begin
            P (full);
            P (mutex)
            remove an item from buffer to nextc;
            V (mutex);
            V (empty);
            ...
            consume the item in nextc;
            ...
        end;
parend;
end;

Readers and writers: Some processes need to read [a file of] data, while other processes need to alter [write] it.

Rules:
- Any number of processes may read a file at a time.
- Only one process may write the file at a time.
How many semaphores do we need to code a solution?

```
var __________: semaphore __________ ;
nreaders: integer := 0;

parbegin
reader:
  while ... do
  begin
    { Readers enter one at a time. }
    P(rmutex);

    { 1st reader waits for readers’
    turn, then inhibits writers}
    if nreaders = 0 then
      P(wmutex);
    nreaders := nreaders + 1;

    { Allow other readers
    to enter or leave. }
  begin
    { Each writer works alone.}
    ...
    Read the file
    ...
    { Readers exit one at
    a time. }
    P(rmutex);
    nreaders := nreaders – 1;

    { Last reader unblocks writers. }
    if nreaders = 0 then

      { Allow reader entry/exit. }
      V(rmutex);
  end;

Dining philosophers: a “classical” synchronization problem.

- Five philosophers spend their lives eating and thinking.
- They sit at a table with 5 chopsticks, one between each pair of philosophers.
• A bowl of spaghetti is in the center of the table.
• When a philosopher is hungry, he attempts to pick up the two chopsticks that are next to him.
• When he gets his chopsticks, he holds them until he is finished eating, then goes back to thinking.

This problem can be programmed by representing each chopstick by a semaphore.

• Grab chopstick \( \Rightarrow P \) semaphore
• Release chopstick \( \Rightarrow V \) semaphore

A philosopher is represented as a process.

Here is the code.

\[
\textbf{while} \; \text{true} \; \{ \text{i.e., forever} \} \; \textbf{do} \; \begin{align*}
\textbf{begin} & \quad P(\text{chopstick}[i]); \\
& \quad P(\text{chopstick}[i + 1 \; \text{mod} \; 5]); \\
& \quad \text{eat} ; \\
& \quad V(\text{chopstick}[i]); \\
& \quad V(\text{chopstick}[i + 1 \; \text{mod} \; 5]); \\
& \quad \text{think} ; \\
\textbf{end};
\end{align*}
\]

Deadlock is possible. How?

\textit{Solutions:}

•
•
•
•

Starvation is also possible. (literally!)
**Eventcounts and sequencers:** The computer version of “For Better Service, Take a Number.”

- Sequencer—a numbered tag.
- Eventcount—the value of the “Now Serving” sign (starts at 0).

Customers take a **ticket** and then **await** service.

Operations on sequencers:

- **ticket**(S); Returns the value of the next numbered tag (starting with 0).

Operations on eventcounts:

- **await**(E, v); Wait until eventcount E reaches v.
- **advance**(E); Increment eventcount E.
- **read**(E); Return current value of eventcount E.

Implementation of **await**(E, v):

```
if E < v then
  begin
    place the current process on the queue associated with E;
    resume next process;
  end;
```

Implementation of **advance**(E):

```
E := E + 1;
resume the process(es) waiting for E's value to reach current value of E;
```
How would we implement a critical section?

... Critical section ...


writer:
while ... do
begin
  { Each writer works alone. }
  await(Win, ticket(Wticket));
  Perform write operations.
  ...
  { Allow other writers (or a reader) to lock out. }
  advance(Win);
end;

• Would the program work if the first \texttt{advance(Rin)} were placed directly after the first \texttt{await(Rin, ticket(Rticket))}? \\

• Why is the second \texttt{await(Rin, ticket(Rticket))} needed? \\

• Does the program make the readers exit in the same order that they entered? \\

\textbf{And and or synchronization:} Thus far, we have discussed synchronization that occurs when a process waits for some single event to occur. \\

But in many synchronization problems, a process may be waiting for— \\

• the occurrence of \textit{any one} of a set of events before it takes some action (or synchronization), or \\
• the occurrence of \textit{all} the events in a set before it proceeds to take action (and synchronization).
Or *synchronization*: Suppose a device-controller process is waiting to handle the next device interrupt.

It is therefore waiting for the occurrence of a single event which can come from any device in a set of devices.

It does not know, however, which device will request service next.

Once an event (interrupt) has occurred, the process needs to figure out which event it was.

(We could use a separate process to wait for each interrupt, but this would be wasteful.)

Instead, we can list all the events and a “discrimination variable” as arguments to the *wait*:

\[
\text{wait-or}(e_1, e_2, \ldots, e_n, A);
\]

where \(e_i\) denotes a semaphore (with associated event) and the \(A\) returns which event(s) occurred.

The device-controller process would then use the *wait-or* in the following way:

*device-controller:*

```
begin
  while ... do
  begin
    wait \(e_1, e_2, \ldots, e_n, A\);
    case \(A\) of
      \(e_1\): Perform action 1;
      \(e_2\): Perform action 2;
      \(\vdots\)
      \(e_n\): Perform action \(n\);
    endcase;
  endloop;
end;
```
A naïve implementation would—

- place the waiting process on the event queue of each semaphore,

- then remove it from all the queues when the process is awakened.

A better implementation combines

- all the event variables, and
- the discrimination parameter $A$

into a single event variable. The new variable $e'$ stands for $e_1$, $e_2$, …, $e_n$, and $A$.

The process is then placed on a single queue associated with all the events.

Then, when an event occurs, the process is removed from this common queue.

It then tests the variable $e'$ to determine what event has occurred, and takes the appropriate action.
Some event in \( e_1, e_2, \ldots, e_n \) occurs …

Here is the code:

```
device-controller
begin
  while … do
  begin
    wait \( e' \);
    case \( e' \) of
      \( e_1 \): Perform action 1;
      \( e_2 \): Perform action 2;
      \vdots
      \( e_n \): Perform action \( n \);
    endcase;
    endloop;
  end
end
```

(Of course, this may require checking more than one event queue when an event occurs.)

And synchronization: Suppose a process needs two files at the same time, or three I/O devices.

It must wait until all are available.

The naïve implementation is prone to deadlock:

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P(Dmutex) ); ( P(Emutex) );</td>
<td>( P(Emutex) ); ( P(Dmutex) );</td>
</tr>
</tbody>
</table>
What happens if both processes execute each statement at the same time ("in lock step")?

As we will see later, several strategies can be used to avoid deadlock.

One simple strategy appropriate for semaphores is to

The *simultaneous P*, or *SP*, operation is specified as follows:

\[
\text{SP}(s_1, s_2, \ldots, s_n):
\begin{aligned}
\text{mutexbegin} \\
\quad \text{if } s_1 > 0 \text{ and } s_2 > 0 \text{ and } \ldots \text{ and } s_n > 0 \text{ then} \\
\qquad \text{for } i := 1 \text{ to } n \text{ do} \\
\qquad \quad s_i := s_i - 1; \\
\quad \text{else} \\
\qquad \text{begin} \\
\qquad \quad \text{Place the process in the queue of the first } s_j \text{ that is } \leq 0; \\
\qquad \quad \text{Set program counter to re-execute entire SP operation when process is reawakened.} \\
\qquad \text{end}; \\
\text{mutexend;}
\end{aligned}
\]

The *SV* operation is defined similarly:

\[
\text{SV}(s_1, s_2, \ldots, s_n):
\begin{aligned}
\quad \text{for } i := 1 \text{ to } n \text{ do} \\
\qquad \text{begin} \\
\qquad \quad \text{mutexbegin} \\
\qquad \qquad \text{mutexend}; \\
\qquad \quad \text{if process(es) are waiting on } s_j \text{ then} \\
\qquad \qquad \quad \text{let one of the waiting processes proceed.} \\
\qquad \text{end};
\end{aligned}
\]

Using *SP* and *SV*, it is easy to implement dining philosophers:
while true {i.e., forever} do
begin

    eat ;

    think ;
end;

Language constructs (S&G, §6.6): Although semaphores solve
the critical-section problem concisely and efficiently, they are not
a panacea. It is easy to make programming mistakes using
semaphores.

What are some of the mistakes cited by S&G?

•

•

•

Perhaps higher-level mechanisms built into programming
languages could help avoid these problems.

Critical regions: Suppose our program has a shared variable \( v \) that
can only be manipulated by one process at a time.

Then if a language provides critical regions, the compiler will
allow a program to access \( v \) only within a critical region.

The variable \( v \) can be declared

\[
\text{var } v: \text{shared } T;
\]

A variable declared like this can be used only within a region
statement:

\[
\text{region } v \text{ do } S;
\]
This means that while statement $S$ is being executed, no other process can access the variable $v$.

[Critical regions can implemented easily by a compiler. For each declaration like the one above, the compiler—

- generates a semaphore $v$-mutex initialized to ___,
- generates this code for each region statement:
  
  $P(v$-mutex);  $S$;  $V(v$-mutex);  ]

We can think of a process $P_1$ that is executing in a region $v$ as “holding” variable $v$ so that no one else can access it.

Suppose it tries to access another shared variable $w$ while it is in that region ($v$’s region).

In order to access this shared variable $w$, what kind of statement would it have to execute?

Suppose process $P_1$ does this while another process is doing the reverse—holding variable $w$ and trying to access $v$. What happens?

This code brings this about—

A compiler should be able to check for this situation and issue a diagnostic.

To solve some synchronization problems, it is necessary to use conditional critical regions—critical regions that are entered only if a certain expression $B$ is true:

```
region $v$ when $B$ do $S$;
```

Suppose a process executes this statement. When it enters the critical region,
• If $B$ is true, it executes statement $S$.
• If $B$ is false, it releases mutual exclusion and waits until $B$
becomes true and no other process is in the critical region.

Let’s recode the bounded-buffer problem using conditional
critical regions:

```plaintext
var buffer: shared record
    pool: array [0 .. n-1] of item;
    count, in, out: integer;
end;

parbegin
{ Means “do the following in parallel.”}
producer:
    nextp: item;
    while ... do
    begin
        ...
        produce an item in nextp;
        ...
        region buffer when count < n do
        begin
            pool[in] := nextp;
            in := (in + 1) mod n;
            count := count + 1;
        end;
    end;

consumer:
    nextc: item;
    while ... do
    begin
        region buffer when count > 0 do
        begin
            nextc := pool[out];
            out := (out + 1) mod n;
            count := count - 1;
        end;
        ...
        consume the item in nextc;
        ...
    end;
parend;
```
Why is a \textit{conditional} critical region used for the bounded-buffer problem?

\textit{The \texttt{await} statement}: The \texttt{when} clause allows a process to wait only at the beginning of a critical region.

Sometimes a process needs to wait in the middle of the region. For that, an \texttt{await} statement can be used:

\begin{verbatim}
region v do
  begin
    S1;
    await(B);
    S2;
  end;
\end{verbatim}

As before, if a process executes this statement,

\begin{itemize}
  \item If $B$ is true, it executes statement $S2$.
  \item If $B$ is false, it releases mutual exclusion and waits until $B$ becomes true and no other process is in the critical region. Then it continues where it left off, at statement $S2$.
\end{itemize}
We can use `await` statements to code a solution to the readers-and-writers problem.

Our semaphore solution to readers-and-writers was a *reader-preference* solution, since if a reader and a writer were both waiting, the reader went ahead.

The solution we will code now is a writer-preference solution.

```pascal
var v: shared record
    nreaders, nwriters: integer;
    busy: boolean;
end;
parbegin
reader:
    region v do
    begin
        await(nwriters = 0);
        nreaders := nreaders + 1;
    end;
    ...
    read file
    ...
    region v do
    begin
        nreaders := nreaders - 1;
    end;
writer:
    region v do
    begin
        nwriters := nwriters + 1;
        await((not busy) and (nreaders = 0));
        busy := true;
    end;
    ...
    write file
    ...
    region v do
    begin
        nwriters := nwriters - 1;
        busy := false;
    end;
parend;
```
Although critical regions avoid low-level programming problems like forgetting to release a semaphore, they can’t assure that a resource is used correctly.

For example, other code could write a file without bothering to check if someone is reading it at the same time.

Or, a process could release a file it hadn’t acquired yet.

Monitors (S&G §6.7): According to the object model, each object has a certain number of operations defined on it.

For example, a stack object has (at least) two operations: push and pop.

Some programming languages (e.g., Modula-2, Ada) allow this to be programmed (approximately) as follows:

```pascal
type stack = monitor
  { Declarations local to stack }
var  elts : array [1 .. 100] of integer;
    ptr : 0 .. 100;

procedure entry push(x: integer);
begin
  ptr := ptr + 1;
  elts[ptr] := x;
end;

function entry pop : integer ;
begin
  pop := elts[ptr];
  ptr := ptr − 1;
end;

begin { Initialization code }
  ptr := 0;
end stack;
```

A class is a special data type, which has operations associated with it.

Operations denoted as entry operations can be invoked from outside by any process that has a variable of the type (e.g., stack).
Operations not marked **entry** can only be invoked from inside the class declaration itself.

Given this declaration of the **stack** data type, other programs can declare stacks like this

```
var  s1, s2: stack ;
```

and invoke procedures of the **stack** class like this:

```
s1.push(45)
x := s2.pop ;
```

One problem with **classes**: How to stop two or more processes from manipulating stack at same time.

One problem with **semaphores**: Code is low level and easy to get wrong.

For example, if programmer forgets to put in a corresponding **V** operation for some **P**, deadlock can result.

Timing dependent ⇒ hard to debug.

Both these deficiencies can be overcome by monitors.

A **monitor** is a special kind of class that allows only one process at a time to be executing any of its procedures. (This property is guaranteed by the implementation of the monitor.)

The stack **class** above can be changed into a monitor by just replacing the word “**class**” by the word “**monitor**.”

**Wait** and **Signal** operations: Inside a monitor, variables of type **condition** may be declared.

Operations on condition variables:

- **wait** on a condition variable suspends process and causes it to leave monitor

- **signal** on a condition variable causes resumption of exactly one of the waiting processes. (If there are no waiting processes, it has no effect.)

Condition variables are private to a particular monitor; they may not be used outside of it.
Scenario:

- Process $P$ attempts to enter monitor $M$ by invoking one of its operations.
- $P$ may wait in $M$’s entry queue.
- After $P$ is allowed to enter $M$, it has exclusive access to all of $M$.
- Exclusive access ends when
  - $P$ exits normally from the monitor because its operation is completed, or
  - $P$ waits on a condition variable (in this case, $P$ is blocked). Exclusive access is regained after the event waited on is signaled.

Meanwhile, the signaling process leaves the monitor.

A monitor has these queues:

- the entry queue
- one queue associated with each condition variable
- the queue for signaling processes.

A simple monitor (merely implements a binary semaphore):

```pascal
type semaphore = monitor
var   busy : boolean ;
     nonbusy : condition ;

procedure entry P ;
begin
  if busy then nonbusy.wait ;
  busy := true ;
end ;
procedure entry V ;
begin
  busy := false ;
  nonbusy.signal ;
end ;

begin { Initialization code }
  busy := false
end ;
```
A semaphore is declared as—

```
var sem : semaphore ;
```

and operated on by—

```
sem.P;   or   sem.V;
```

**Example 1: The bounded-buffer problem.**

```
type bounded_buffer = monitor
    var buffer : array [0 . . N –1] of portion ;
    lastpointer : 0 . . N –1;
    count : 0 . . N ;
    nonempty , nonfull : condition ;

    procedure entry append ( x: portion );
        begin
            if count = N then nonfull.wait ;
            { 0 ≤ count < N }
            buffer[lastpointer] := x ;
            lastpointer :=
                (lastpointer +1) mod N ;
            count := count + 1 ;
            nonempty.signal
        end ;

    function entry remove : portion ;
        begin
            if count = 0 then nonempty.wait ;
            { 0 < count ≤ N }
            remove :=
                buffer[(lastpointer–count) mod N ];
            count := count –1 ;
            nonfull.signal ;
        end ;

    count := 0 ; lastpointer := 0 ;
end ;
```
Example 2: Dining philosophers. Here is a deadlock-free solution to the dining-philosophers problem with monitors.

```plaintext
var state: array [0 .. 4] of (thinking, hungry, eating);

Philosopher \( i \) sets \( \text{state}[i] = \text{eating} \) only if his neighbors are not eating.

var self: array [0 .. 4] of condition;

Philosopher \( i \) delays himself on \( \text{self}[i] \) when he is hungry but unable to obtain two forks.

type dining_philosophers = monitor
var state: array [0 .. 4] of (thinking, hungry, eating);
self: array [0 .. 4] of condition;

procedure entry pickup (i: 0 .. 4);
begin
  state[i] := hungry;
  test(i);
  if state[i] \neq \text{eating} then
    self[i].wait;
end;

procedure entry putdown (i: 0 .. 4);
begin
  state[i] := thinking;
  test((i-1) mod 5);
  test((i+1) mod 5);
end;

procedure test(k: 0 .. 4);
begin
  if state[(k-1) mod 5] \neq \text{eating} and state[k] = hungry
    and state[(k+1) mod 5] \neq \text{eating}
  then begin
    state[k] := eating;
    self[k].signal;
  end;
end;

begin
  for i := 0 to 4 do state[i] := thinking;
end.
```
An instance of `dining_philosophers` can be declared by—

```haskell
var dp : dining_philosophers ;
```

Here is the code for a process for one of the philosophers:

```haskell
process Pi : { philosopher i }
begin
    ...
    dp.pickup(i);
    ... { eat }
    dp.putdown(i);
end;
```

The nested monitor-call problem: A process in one monitor $M_1$ can call a procedure in another monitor $M_2$.

The mutual exclusion in $M_1$ is not released while $M_2$ is executing.

This has two implications:

- Any process calling $M_1$ will be blocked outside $M_1$ during this period.
- If the process enters a condition queue in $M_2$, deadlock may occur.

```
M1
```
```
M2
```

Conditional waiting in monitors: If several processes are suspended on condition $x$ and a $x$.signal operation is executed, which process is resumed next?

Sometimes processes should be resumed in priority order. E.g., if SJN scheduling used, when the processor becomes free, the shortest job should be resumed next.
The priority-wait (conditional-wait) construct: Associates an integer value with a wait statement

\[ x.wait(p) \]

\( p \) is an integer called the priority. When \( x.signal \) is executed, the process associated with the smallest priority is resumed next.

The empty construct: \( x.empty \) is a boolean function that tests whether any processes are waiting on condition variable \( x \).

Example: A monitor for the LOOK algorithm.

The LOOK algorithm keeps the head moving in the same direction until there are no more requests to service in that direction.

If another request comes in ahead of the head, it will be serviced on the current sweep:

If another request comes in behind the head, it will wait until the next sweep (the disk head will have to go to the other end and turn around before reaching the new request):

\[
\text{type disk_head = monitor} \\
\text{var head_pos: 1 .. max_cyl_number; busy: boolean; direction: (up, down); downsweep, upsweep: condition;}
\]
procedure entry acquire (dest_cyl: 1 .. max_cyl_number);
begin
  if busy then
    if head_pos < dest_cyl
      or (head_pos = dest_cyl and direction = down) then
      upsweep.wait (dest_cyl);
    else
      busy := true;
      { Issue command to move disk arm. }
      head_pos := dest_cyl;
  end
end;

procedure entry release;
begin
  busy := false;
  if direction = up then
    if upsweep.empty then
      begin
        direction := down;
        downsweep.signal;
      end
    else
      upsweep.signal
  else
    if downsweep.empty then
      begin
        direction := up;
        upsweep.signal;
      end
    else
      downsweep.signal
  end;

begin
  head_pos := 1;
  direction := up;
  busy := false;
end;

If da is an instance of the disk_head monitor, a process that wants to use the disk executes code of this form:

da.acquire (i);

... { Access cylinder i }

  da.release ;
Notes on the program: We need two condition variables to specify priority.

Without priority wait, we would need one condition variable per cylinder.

release signals a request in the current direction. If there is none, the direction is reversed.

Which condition variable does a would-be acquirer wait on—

• if the head position is below the desired cylinder?
• if the head position is above the desired cylinder?

Does it make a difference which direction the head is moving?

What priorities are specified if—

• the head is at cylinder 25 and processes request cylinders 100 and 200?
• the head is at cylinder 250 and requests arrive for cylinders 50 and 150?

Why isn’t the head_pos test simply

\[
\text{if } \text{head_pos} < \text{dest_cyl} \ ?
\]

How can we be sure that processes use this monitor correctly?

• If processes always make calls on the monitor in a correct sequence, and

• if no uncooperative process accesses the disk without using the monitor. For large systems, access control methods needed.
**Path expressions**: Monitors are compatible with the object model, because they define the operations on a resource in the same place as the resource itself.

But synchronization using *wait* and *signal* tends to be scattered throughout the monitor’s procedures.

This impairs the readability of the code.

To attack this problem, *path expressions* express order of execution in a single line.

Path expressions are used within *objects* that are similar to monitors.

- Procedures within objects are not automatically mutually exclusive.
- All execution constraints are declared at the beginning of the object, using a path expression.

Its syntax is

```
path <restriction expression> end
```

A `<restriction expression>` is defined as follows.

1. A procedure name $P$ is a `<restriction expression>`; a single procedure name implies no restriction.

2. If $P_1$ and $P_2$ are `<restriction expressions>`, then each of the following is also a `<restriction expression>`:
   - $P_1, P_2$ denotes *concurrent execution*.
     No restriction is placed on the order in which $P_1$ and $P_2$ are invoked, or on the number of concurrent executions (i.e., any number of each can execute concurrently).
   - $P_1; P_2$ denotes *sequential execution*.
     A separate invocation of $P_1$ must complete before each invocation of $P_2$.
     The execution of $P_2$ does not inhibit the execution of $P_1$;
     - many different invocations of $P_1$ and $P_2$ may be active concurrently,
     - as long as the # of $P_2$s that have begun execution is $\leq$ the # of $P_1$s that have completed.
• \( n: (P1) \) denotes resource restriction.
  It allows at most \( n \) separate invocations of \( P1 \) to execute simultaneously.

• \([P1]\) denotes resource derestricion.
  It allows an arbitrary number of invocations of \( P1 \) to coexist simultaneously.

Examples:

1. **path** 1: \((P1)\) **end**
   
   Procedure \( P1 \) must be executed sequentially; only one invocation may be active at a time.

2. **path** 1: \((P1), P2\) **end**
   
   Multiple invocations of \( P1 \) result in sequential execution, while no restriction is placed on \( P2 \) (i.e., any number of invocations of \( P2 \) are allowed.)

3. **path** 1: \((P1), 1: (P2)\) **end**
   
   A maximum of one \( P1 \) and one \( P2 \) can execute concurrently.

4. **path** 6: \(5:(P1), 4:(P2)\) **end**
   
   As many as five invocations of \( P1 \) and four invocations of \( P2 \) can proceed concurrently, as long as the overall limit of six invocations is not exceeded.

5. **path** 5: \((P1; P2)\) **end**
   
   Each invocation of \( P2 \) must be preceded by a complete execution of \( P1 \); at most five invocations of \( P1 \) followed by \( P2 \) may proceed concurrently.

   (i.e., \( P1 \) can get ahead of \( P2 \) by at most five invocations.)

6. **path** 1: \([P1], [P2]\) **end**
   
   Procedures \( P1 \) and \( P2 \) operate in mutual exclusion, but each can concurrently execute as many times as desired.

   If all executions of one procedure have finished, either procedure may start again.
What would be the path expression for the weak reader-preference solution to the reader/writer problem?

\[
\text{path}
\]

The bounded-buffer problem with path expressions:

\[
\text{type bounded_buffer = object} \\
\quad \text{path } N: (\text{append; remove}), \\
\quad \quad 1: (\text{append, remove}) \text { end}
\]

\[
\text{var buffer : array } [0 . . N-1] \text{ of portion;} \\
\quad \text{lastpointer : } 0 . . N-1; \\
\quad \text{count : } 0 . . N;
\]

\[
\text{procedure entry append (x: portion);} \\
\quad \text{begin} \\
\quad \quad \text{buffer [lastpointer] := x;} \\
\quad \quad \text{lastpointer :=} \\
\quad \quad \quad (\text{lastpointer +1) mod N; } \\
\quad \quad \text{count := count +1; } \\
\quad \text{end;}
\]

\[
\text{function entry remove: portion;} \\
\quad \text{begin} \\
\quad \quad \text{remove :=} \\
\quad \quad \quad \text{buffer [(lastpointer-count)} \\
\quad \quad \quad \quad \text{mod N]; } \\
\quad \quad \text{count := count-1; } \\
\quad \text{end;}
\]

\[
\text{begin} \\
\quad \text{count := 0; lastpointer := 0; } \\
\text{end;}
\]

Path expressions can easily be implemented using semaphores.
Synchronization in high-level languages: Several high-level languages provide constructs for creating and synchronizing parallel processes.


Premises:

• Input and output are basic primitives of programming.

• Parallel composition of communicating sequential processes is a fundamental program-structuring method.

Communication occurs when one process names another as a destination for output, and the second process names the first as the source for input.

Then the output value is copied to input, without automatic buffering.

The language:

• Guards.

  ◦ A guard is a boolean expression.

  ◦ A guarded command is a guard, together with a list of statements which are executed only if the guard does not fail.

  \[
  \langle \text{guarded command} \rangle : :: = \langle \text{guard} \rangle \to \langle \text{statements} \rangle
  \]

  ◦ A guarded command set is a list of alternative guarded commands. As the list is scanned, an arbitrary command whose guard does not fail will be executed.

  \[
  \langle \text{guarded command set} \rangle : :: = \\
  \langle \text{guarded command} \rangle \sqcup \ldots \sqcup \langle \text{guarded command} \rangle
  \]
• Semantics of alternative commands.
  - All guards execute concurrently. (The statements within a
    guard are executed left-to-right.)
  - If all guards fail, the alternative command fails.
  - For performance reasons, the system might select the
    earliest completed guard. The statement list associated
    with the selected guard is executed.
  - Execution of all other guards is discontinued.
  - A nondeterministic choice is made between two
    simultaneous guards.

• Syntax for input/output commands.
  - \( \langle \text{input command} \rangle : :: = \langle \text{source} \rangle ? \langle \text{target variable} \rangle \)
  - \( \langle \text{output command} \rangle : :: = \langle \text{destination} \rangle ! \langle \text{expression} \rangle \)

\( \langle \text{source} \rangle \) and \( \langle \text{destination} \rangle \) are executing concurrently.

• Conditions for communication.
  - One process names the other as source for input.
  - The other process names the first as destination for output.
  - The type of the target variable matches the type of the
    output variable.

Then the two statements execute concurrently, assigning the
output value to the target variable.

• An I/O command fails when—
  - the source or destination process is terminated;
  - its expression is undefined; or
  - the type of the target variable of an input command is
    not the same as the type of the value denoted by the
    expression of the output command.
• Examples:
  ° `card_reader ? card_image`
  ° `line_printer ! line_image`

• Parallel commands. Commands that are to execute in parallel are surrounded by brackets and separated by `||`'s:

```
[ ⟨process⟩ || … || ⟨process⟩ ]
```

Here are some examples of parallel commands:

```
[ card_reader ? card_image
   || line_printer ! line_image ]

[ room : : ROOM || fork (i : 0 . . 4) : : FORK
   || phil (i : 0 . . 4) : : PHIL ]
```

° All of the parallel commands start simultaneously.

° The parallel command terminates when all of its commands have terminated.

° Each process must be disjoint. This means that—
  ◊ it doesn’t assign to non-local variables mentioned in the others; and
  ◊ it doesn’t use non-local process names used by the others.

• Repetition is specified by an `*`:

```
* ⟨alternative command⟩
```

For example, a linear search through an `n`-element array can be coded like this:

```
i := 0; *[i < n ; content(i) ≠ key
          → i := i + 1]
```

Iteration occurs as long as one of the alternatives succeeds. (In this example, there was only one alternative.)
The bounded-buffer problem in CSP: Below is an implementation of a bounded-buffer manager.

The consumer process asks for the next portion and then attempts to input it:

\[ X \text{ !more ( ); } X \text{ ?p} \]

The producer provides a buffer portion via
\[ X \text{ !p} \]

\[ X ::
buffer : (0 . . 9) \text{portion} ;
in, out : integer ; \text{in} := 0 ; \text{out} := 0 ;
\text{comment} 0 \leq \text{out} \leq \text{in} \leq \text{out} + 10 ;
\]
\*\[ \text{in} < \text{out} + 10 ;
\text{producer} \text{ ?buffer (in mod 10)}
\rightarrow \text{in} := \text{in} + 1
\]
\*\[ \text{out} < \text{in} ; \text{consumer} \text{ ?more ( )}
\rightarrow \text{consumer} \text{ !buffer (out mod 10)} ;
\text{out} := \text{out} + 1 \]

Notes:

- When \( \text{out} < \text{in} < \text{out} + 10 \), the selection of the alternative depends on whether \text{producer} produces first or \text{consumer} consumes first.

- When \( \text{out} = \text{in} \), the buffer is empty, and the second alternative can’t be selected even if \text{consumer} is waiting.

- \text{Producer} can’t go ahead when \( \text{in} = \text{out} + 10 \).

- \( X \) terminates when \( \text{out} = \text{in} \) and \text{producer} has terminated.

- Strictly speaking, \text{more( )} should be a variable. It doesn’t have to have the same name as in the sending process.

Dining philosophers in CSP: There are eleven processes—

- One for the room.
- Five for the philosophers.
- Five for the forks.
$PHIL = \begin{array}{c}
* [THINK; \\
\text{room } ! \text{enter( );} \\
\text{fork}(i) ! \text{pickup( );} \\
\text{fork}(\langle i +1 \rangle \mod 5) ! \text{pickup( );} \\
\text{EAT;} \\
\text{fork}(i) ! \text{putdown( );} \\
\text{fork}(\langle i +1 \rangle \mod 5) ! \text{putdown( );} \\
\text{room } ! \text{exit( );} \\
] \\
\end{array}$

$FORK = \begin{array}{c}
* [\text{phil}(i) ? \text{pickup( )} \\
\quad \rightarrow \text{phil}(i) ? \text{putdown( );} \\
\quad \rightarrow \text{phil}(\langle i -1 \rangle \mod 5) ? \text{pickup( )} \\
\quad \rightarrow \text{phil}(\langle i -1 \rangle \mod 5) ? \text{putdown( )} \\
] \\
\end{array}$

$ROOM = \text{occupancy} : \text{integer};$

$\text{occupancy} := 0;$

$\begin{array}{c}
* [(i : 0 . . 4) \text{phil}(i) ? \text{enter( )} \\
\quad \rightarrow \text{occupancy} := \text{occupancy} + 1 \\
\quad \rightarrow \text{occupancy} := \text{occupancy} - 1 \\
\] \\
\end{array}$

comment — All these processes operate in parallel.

$[\text{room} :: \text{ROOM} \\
|| \text{fork}(i : 0 . . 4) :: \text{FORK} \\
|| \text{phil}(i : 0 . . 4) :: \text{PHIL}]$

This solution does not avoid starvation or deadlock.

Ada tasks: A task provides several types of services, each designated by an entry name. A service request is made by an entry call.

Only one entry can be executed at a time. Thus, every entry in a task behaves like a critical section with respect to itself and all other entries of the task in which it is defined.

A task consists of two parts:

• The specification of the task.

  task $t$ is
  — entry declarations
  end $t$;
• The task body.

  task body \( t \) is
    — declarations
  begin
    — statements
  end \( t \);  

Different tasks proceed in parallel, except at points where they synchronize.

Tasks synchronize when they arrive at the same rendezvous point.

• One task calls an \texttt{entry} of the other task.
• The other task executes an \texttt{accept} statement.

Either action may occur first.

\begin{align*}
\text{Calling task} & \quad \text{Called task} \\
\text{task } t1 \text{ is} & \quad \text{task } t2 \text{ is} \\
& \quad \text{entry } e(\ldots); \\
& \quad \ldots \\
& \quad \text{accept } e(\ldots) \text{ do } \ldots \\
& \quad \text{end } e; \\
& \quad \ldots \\
\end{align*}

• Whichever task first arrives at the rendezvous point waits for the other.
• When both tasks are there, the \texttt{accept} statement is executed.
• Thereafter, both tasks continue in parallel.

A sequence of accept statements may be used to force the calling task to request actions in a particular order. For example,

\begin{align*}
\text{accept } e1(\ldots) \text{ do } & \text{[body of } e1]\ldots \text{ end } e1; \\
\text{accept } e2(\ldots) \text{ do } & \text{[body of } e2]\ldots \text{ end } e2; \\
\text{accept } e3(\ldots) \text{ do } & \text{[body of } e3]\ldots \text{ end } e3; \\
\end{align*}
specifies that entry \( e_1 \) must be the first executed, followed by no other entry than \( e_2 \), which in turn is followed by an execution of \( e_3 \).

Often, a sequence of \texttt{accept} statements may be embedded in a loop so that they can be processed repeatedly:

```plaintext
while ... loop
    accept e1(...) do ... end e1;
    accept e2(...) do ... end e2;
    accept e3(...) do ... end e3;
end loop;
```

In a file system, for example, a file might be operated on by a file-open, followed by a sequence of read or write requests.

\textit{The select statement:} A \texttt{select} statement allows for a choice between entries. In format, it is similar to a \texttt{case} statement in other languages (but its effect is different).

Some of the alternatives in a select statement may be \texttt{qualified}—eligible to be executed only when a boolean expression is true.

```plaintext
select
    accept e1(...) do ... end;
or
    when ... =>
        accept e2(...) do ... end;
or
    ...
end select;
```

An alternative is \texttt{open} (eligible to be executed) if there is no \texttt{when} condition, or if the condition is true.

An open alternative can be selected if a corresponding rendezvous is possible. If several alternatives can thus be selected, one of them is chosen at random.

If no rendezvous is immediately possible, the task waits until an open alternative can be selected.

The implementation of a bounded buffer below uses a \texttt{when} clause to prevent—

- inserting an item in a full buffer, or
- removing an item from an empty buffer.
task bounded_buffer is
  entry insert (it: in item);
  entry remove (it: out item);
end bounded_buffer;

task body bounded_buffer is
  buffer: array (0 .. 9) of item;
  in, out: integer := 0;
  count: integer range 0 .. 10 := 0;
begin
  loop
    select
      when count < 10 =>
        accept insert (it: in item)
        do buffer (in mod 10) := it;
        end;
        in := in + 1;
        count := count + 1;
      or
      when count > 0 =>
        accept remove (it: out item);
        do it := buffer (out mod 10);
        end;
        out := out + 1;
        count := count – 1;
    end select;
  end loop;
end bounded_buffer;

In this code, the “in := in + 1; count := count + 1” could be placed inside the do … end.

What difference would it make?

An else clause can be added to a select statement in case no other alternative can be executed:
The else part is executed if no alternative can be immediately selected.

Here is a flowchart of how an entry is chosen:

The **delay** statement: Allows a program to wait to see if an alternative will become open within a specified period of time.

**delay** (duration)

An open alternative beginning with a **delay** statement will be executed if no other alternative has been selected before the specified duration has elapsed.

Any subsequent statements of the alternative are then executed:
select
  accept insert( … )
or
  accept remove( … )
or
  delay 10;
end select;

The calls on insert and remove are canceled if a rendezvous is not started within 10 seconds.

Delay statements and else clauses cannot both be used.

Synchronization in Solaris 2 (S&G, §6.8): Before Solaris 2, SunOS used critical sections to guard multiply accessed data structures.

Interrupt level was set high enough so that no interrupt could cause modification to the critical data.

Solaris 2—

• is multi-threaded,
• provides real-time capabilities, and
• supports multiprocessors.

Why wouldn’t the approach of raising the interrupt level work anymore?

Even if critical sections could have been used, they would have caused a large performance degradation.

For synchronization, Solaris 2 uses three mechanisms:

• Adaptive mutexes.
• Condition variables.
• Readers-writers locks.

Adaptive mutexes adapt their behavior, depending on how long they expect to have to wait.

At the outset, an adaptive mutex is similar to a spinlock. But it behaves differently according to what conditions prevail.
Data locked?

Yes

Locked by thread that is running?

Yes

Wait for lock to become free.

No

Set lock and access data

Yes

Go to sleep until lock is released.

No

Why the difference in behavior depending on whether the thread is running?

How would the behavior be different on a uniprocessor system?

Condition variables: Under what condition would it be inefficient to use an adaptive mutex?

With a condition variable, if the desired data is locked, the thread goes to sleep.

When a thread frees the lock, it signals the next sleeping thread.

In order for this to be worthwhile,

\[ \text{the cost of putting a thread to sleep and waking it} + \text{cost of _____ context switches} \]
must be

\[ \leq \text{cost of wasting several hundred instructions in a spinlock.} \]

*Readers-writers locks:* Under what condition are both adaptive mutexes and condition variables inefficient?

---

**Atomic transactions** (S&G, §6.9):

Sometimes it is necessary to make sure that a critical section performs a single logical unit of work.

That is, if some of the critical section executes, then

This problem is has been around for a long time in databases.

A collection of instructions that performs a single logical function is called a *transaction*.

A transaction is

- a sequence of *read* and/or *write* operations,
- terminated by a *commit* or an *abort*.

If a transaction aborts, its effects must be completely undone.

But, the effect of a transaction that can has committed cannot be undone by an *abort* operation.

If a transaction aborts, the system state must be restored to

*Log-based recovery:* Atomicity can be insured by recording on stable storage all actions performed by the transaction.

The most widely used method for this is *write-ahead logging*.

Before storage is modified, the following data is written to stable storage:
• Transaction name.
• Data-item name.
• Old value of data item.
• New value of data item.

Special log records record actions such as
• starting a transaction
• of a transaction.

In general, a transaction record looks like this:

\[ \langle T_i \text{ starts} \rangle \]
\[ \langle T_i \text{ writes}, n_j, v_j, v_j \rangle \]
\[ \cdots \]
\[ \langle T_i \text{ commits} \rangle \]

What should happen first? Updating a data item, or writing its log record out to stable storage?

This imposes a certain overhead.

The recovery algorithm uses two procedures:

\[ \text{undo}(T_i) \]

\[ \text{redo}(T_i) \]

Undo and redo must be idempotent (multiple executions have same effect as one execution).

If a transaction \( T_i \) aborts, then we can restore the state of the data that it has updated by executing

If a system failure occurs, which transactions need to be undone, and which need to be redone?

• Transaction \( T_i \) needs to be undone if the log contains the record
  but does not contain the record
• Transaction $T_i$ needs to be redone if the log contains the record

and contains the record

_Checkpoints_ (S&G, §6.9.3): Obviously, we don’t want to have to search the whole log whenever there is a failure.

It is better if the system periodically performs _checkpoints_. At a checkpoint, the system—

• outputs all log records currently residing in volatile storage

• outputs all modified data residing in volatile storage, and

• outputs a log record $\langle$checkpoint$\rangle$.

What is the recovery procedure? There are three kinds of transactions to consider.

1. Transactions that **start** and **commit** before the most recent checkpoint.

2. Transactions that **start** before this checkpoint, but **commit** after it.

3. Transactions that **start** and **commit** after the most recent checkpoint.

Which of these transactions need to be considered during recovery?

Do all such transactions need to be **undo** or **redo**?

Once transaction $T_i$ has been identified, the **redo** and **undo** operations need to be applied only to $T_i$ and the transactions that started after it.

Let’s call these transactions the set $T$.

The recovery algorithm is this:
• For all transactions $T_k$ in $T$ such that the record $\langle T_k$ 
commits$\rangle$ appears in the log, execute

• For all transactions $T_k$ in $T$ that have no__________ in 
the log, execute

Serializability and concurrent atomic transactions (S&G, §6.9.4):
Atomic transactions $\Rightarrow$ concurrent execution must be equivalent
to serial execution in some order.

A schedule is called equivalent to another schedule if each 
schedule results in the same sequence of values being written to 
each location.

One way to assure serializability is simply to execute each 
transaction in a critical section. What’s wrong with this?

We want to determine when an execution 
sequence is equivalent to a serial schedule.

Suppose a system has two data items, $A$ and $B$, which are both 
read and written by two transactions, $T_0$ and $T_1$. A serial 
schedule is shown below.

<table>
<thead>
<tr>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>write($A$)</td>
<td>write($A$)</td>
</tr>
<tr>
<td>read($B$)</td>
<td>read($B$)</td>
</tr>
<tr>
<td>write($B$)</td>
<td>write($B$)</td>
</tr>
</tbody>
</table>

Given $n$ transactions, how many valid serial schedules are there?

A non-serial schedule is formed when transactions overlap their 
execution.

A non-serial schedule may be equivalent to a serial schedule if 
there aren’t any “conflicting” operations in the “wrong” places.
Two operations are said to be *conflicting* if they access the same data, and at least one of them is a write.

In the schedule above, what operations does the write$(A)$ of $T_0$ conflict with?

Given a schedule of operations, we can produce an equivalent schedule by this procedure:

If two operations are consecutive, and do not conflict,
then their order can be swapped to produce an equivalent schedule.

Here is a non-serial schedule.

<table>
<thead>
<tr>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read$(A)$</td>
<td>read$(A)$</td>
</tr>
<tr>
<td>write$(A)$</td>
<td>write$(A)$</td>
</tr>
<tr>
<td>read$(B)$</td>
<td>read$(B)$</td>
</tr>
<tr>
<td>write$(B)$</td>
<td>write$(B)$</td>
</tr>
</tbody>
</table>

How can we transform it into a serial schedule?

If schedule $S$ can be transformed into a serial schedule $S'$ by a sequence of swaps of non-conflicting operations, it is said to be *conflict serializable*. 
Locking protocol: How can serializability be ensured?

One way is to

• associate a lock with each data item,
• require that each transaction obtain a lock before it reads or writes the item, and to
• assure that a certain protocol is followed in acquiring and releasing the locks.

Among the modes in which data can be locked are—

• shared (S)—a transaction that has obtained a shared lock on data item Q can read Q but cannot write it.

• exclusive (X)—a transaction that has obtained an exclusive lock on data item Q can read and/or write it.

If a data item is already locked when a lock is requested, the lock can be granted only if the current lock

Notice the similarity to readers and writers!

Serializability is ensured if we require that each transaction is two phase. This means that it consists of—

• a growing phase, in which the transaction may request new locks, but may not release any locks, and
• a shrinking phase, in which a transaction may release locks, but may not request any new locks.

The two-phase locking protocol ensures conflict serializability.

It does not ensure freedom from deadlock.

Timestamp-based protocols: How is the order of execution determined for conflicting transactions?

With a locking protocol, it is determined by which of the two is first to request a lock that involves incompatible modes.

A timestamp-based protocol decides instead based on the order in which the transactions began.
With a timestamp-based protocol, each transaction $T_i$ is assigned a timestamp $TS(T_i)$ when it starts execution.

If $T_i$ starts before $T_j$, then $TS(T_i) < TS(T_j)$.

How are timestamps determined?

- By the
- By a

The system must ensure that the resulting schedule is equivalent to a serial schedule in which the transactions execute in timestamp order.

Two timestamp values are associated with each data item:

- W-timestamp($Q$) — the highest timestamp of any transaction that successfully executed $\text{write}(Q)$.
- R-timestamp($Q$) — the highest timestamp of any transaction that successfully executed $\text{read}(Q)$.

When does a transaction $T_j$ need to be rolled back? Iff—

- it attempts to read a value that has already been overwritten by a transaction that started later. I.e., if

- it attempts to write a value that has either been read or written by a transaction that started later. I.e., if

When are the R- or W-timestamps set to new values?

What new values are these timestamps set to?
Here is a possible schedule under the timestamp protocol \( (TS(T_0) < TS(T_1)) \):

<table>
<thead>
<tr>
<th></th>
<th>T_0</th>
<th>T_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(B)</td>
<td></td>
<td>read(B)</td>
</tr>
<tr>
<td>read(A)</td>
<td></td>
<td>write(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write(A)</td>
</tr>
</tbody>
</table>

What change to the above schedule would make it impossible under the timestamp protocol?

<table>
<thead>
<tr>
<th></th>
<th>T_0</th>
<th>T_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(B)</td>
<td></td>
<td>read(B)</td>
</tr>
<tr>
<td>read(A)</td>
<td></td>
<td>write(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>write(A)</td>
</tr>
</tbody>
</table>

**Deadlock** (S&G §7.1): A situation where two or more processes are prevented from proceeding by the demands of the other(s).

Simple example:

- Process A has the printer and needs a tape drive. It can’t release the printer till it acquires the tape drive.
- Process B has the tape drive and needs a printer. It can’t release the tape drive till it acquires the printer.

Outline for Lecture 11

I. Deadlock: necessary conditions
II. Resource-allocation graphs
   A. Reusable vs. consumable resources
   B. Operations on resources
   C. Suspending a process
III. Deadlock prevention
In general a set of processes is deadlocked if each process is “monopolizing” a resource and waiting for the release of resources held by others in the set.

Deadlock can arise in other contexts:

- Crossing a river over a path of stones wide enough for only one person.
- Traffic gridlock.

Note that deadlocked processes are not the same as blocked processes.

**Necessary conditions for deadlock:** A deadlock may arise only if these four conditions hold simultaneously—

(C1) Mutual exclusion. At least one resource must be held in a non-sharable mode.

(C2) No preemption. Resources cannot be preempted, but can only be released voluntarily by the processes that are using them.

(C3) Hold and wait. At least one process must be holding a resource and waiting to acquire resources that are held by other processes.

(C4) Circular wait. There must exist a set of processes 

\[ \{p_0, p_1, \ldots, p_n\} \] such that
\[ p_0 \] is requesting a resource that is held by \[ p_1 \],
\[ p_1 \] is requesting a resource that is held by \[ p_2 \],
\[ \vdots \]
\[ \vdots \]
\[ p_{n-1} \] is requesting a resource that is held by \[ p_n \], and
\[ p_n \] is requesting a resource that is held by \[ p_0 \].

These are necessary but not sufficient conditions for deadlock to occur.

**Resource-allocation graphs:** Show the relationship between processes and resources. Useful in visualizing deadlock.

**Definition:** A resource allocation graph is an ordered pair \( G = (V, E) \) where—

- \( V \) is a set of vertices. \( V = P \cup R \), where
  \[ P = \{p_1, p_2, \ldots, p_n\} \] is a set of processes, and
  \[ R = \{r_1, r_2, \ldots, r_m\} \] is a set of resource types

- \( E \) is a set of edges. Each element of \( E \) is an ordered pair \((p_i, r_j)\) or \((r_j, p_i)\), where—
  \( p_i \) is a process (an element of \( P \)), and
  \( r_j \) is a resource type (an element of \( R \))

An edge \((p_i, r_j)\) “from a process to a resource type” indicates that process \( i \) is requesting resource \( j \) (a request edge).

An edge \((r_j, p_i)\) “from a resource type to a process” indicates that resource \( j \) is allocated to process \( p_i \) (an allocation or assignment edge).

Here is an example of a resource allocation graph.
In this graph,

\[ P = \{ p_1, p_2, p_3 \} \]

\[ R = \{ r_1, r_2, r_3, r_4 \} \]

\[ E = \{ (p_1, r_1), (p_2, r_3), (r_1, p_2), (r_2, p_2), (r_2, p_1), (r_3, p_3) \} \]

Each dot designates one instance of a resource.
- Assignment edges point from specific dots.
- Request edges do not point to specific dots.

No cycles exist in this graph. But if we add an edge \((p_3, r_2)\)—

Now there are two cycles:

and

Is there a deadlock here?

However, a cycle does not always mean there is a deadlock.

In the example at the right, there is a cycle but no deadlock, because
All of the resources we have seen so far are called *reusable* resources:

- There is a fixed total inventory. Additional units are neither created nor destroyed.
- Units are requested by processes, and when they are released, are returned to the “pool” so that other processes may use them.

Some resources, however, are *consumable*:

- There is no fixed total number of units. Units may be created (produced) or acquired (consumed) by processes.
- An unblocked producer of the resource may create any number of units. These units then become immediately available to consumers of the resource.
- An acquired unit ceases to exist.

*Examples*: Reusable resources—

Consumable resources—

In a resource-allocation graph, there is a third kind of edge:

A *producer edge* \((r_j, p_i)\) “from a resource type to a process” indicates that process \(i\) produces units of resource \(j\).

Here is a resource-allocation graph with a producer edge:

In this graph,
- \(p_1\) holds 2 units of reusable resource \(r_1\),
- \(p_2\) holds 1 unit of \(r_1\),
- \(p_2\) requests 1 unit of \(r_2\), and
- \(p_2\) is the only producer of \(r_2\).
Operations on resources:

Requests. If \( p_i \) has no outstanding requests (i.e., if it is unblocked), then it may request units of any number of resources \( r_j, r_k, \ldots \)

To reflect this in the graph, add edges \((p_i, r_j), (p_i, r_k), \ldots\) in numbers corresponding to the number of units of each resource requested.

Allocations. If—

- process \( p_i \) has outstanding requests, and
- for each requested resource \( r_j \), the number of units requested \( \leq \) the number of units available \( a_j \),

then \( p_i \) may acquire all requested resources.

To reflect this in the graph,

- reverse the direction of each request edge to a reusable resource to make it an allocation edge, and
- remove each request edge to a consumable resource, simulating the consumption of units by \( p_i \).

In either case, the number of available units \( a_j \) is reduced by the number of units acquired or consumed by \( p_i \).

Releases. If process \( p_i \) has no outstanding requests (i.e., it is executable), and there are assignment or producer edges \((r_j, p_i)\), then \( p_i \) may

- release any subset of the reusable resources it holds, or
- produce any number of units of consumable resources for which it is a producer.

To reflect this in the graph,

- remove assignment edges from reusable resources, but
- producer edges are permanent.

Inventories \( a_j \) are incremented by the number of units of each resource \( r_j \) released or produced.
Example: Consider the resource-allocation diagram two pages back.

Suppose \( p_1 \)'s requests one unit of \( r_1 \) and two units of \( r_2 \).

Then edge \((p_1, r_1)\) and two edges \((p_1, r_2)\) appear in the diagram.

Since not all \( p_1 \)'s requests are satisfiable, \( p_1 \) is blocked in this state.

Now suppose \( p_2 \)'s request for one unit of \( r_2 \) is granted. What happens?

Now suppose \( p_2 \) produces three units of \( r_2 \). What happens to the diagram?

Methods for handling deadlocks: There are basically two methods for dealing with deadlocks. Either

- never allow it to happen, or
- when it does happen, try to get out of it.

The “never allow it” strategy can be subdivided into two cases, giving three methods overall.
• Deadlock prevention. Disallow one of the four necessary conditions for a deadlock to exist.

• Deadlock detection and recovery. Try to notice when deadlock has occurred, then take action (usually drastic action!) to remove it.

• Deadlock avoidance. Rather than disallowing any of the four conditions, dynamically “sidestep” potential deadlocks.

**Deadlock prevention**: Must disallow one of the four necessary conditions.

(C1) Mutual-exclusion condition. Make resources sharable—e.g., read-only files. (Not always possible).

(C2) No-preemption condition. Make resources preemptible. Two strategies:

* If a process that is holding some resources requests more resources that are not available, preempt the resources it is holding.

  The preempted resources are added to the list of resources it was waiting for.

* If a process requests some resources that are not available, preempt them from another waiting process if there is a waiting process.

  Otherwise the requesting process must wait. It may lose some of its resources, but only if another process requests them.


(C3) The hold-and-wait condition. Again, two techniques can be used.

* Preallocation. A process requests all the resources it will ever need at the start of execution.
Problems: Expense of unused resources.

Lack of foreknowledge, especially by interactive processes.

- Allow a process to request resources only when it has none allocated. Must always release resources before requesting more.

Problem: Starvation more likely.

(C4) Circular-wait condition. Impose a linear ordering of resource types. For example, let

\[
\begin{align*}
F(\text{card reader}) &= 1 \\
F(\text{disk drive}) &= 5 \\
F(\text{tape drive}) &= 7 \\
F(\text{printer}) &= 9
\end{align*}
\]

More formally, let \( R = \{r_1, r_2, \ldots, r_m\} \) be the set of resource types, and \( N \) be the natural numbers. Define a one-to-one function

\[
F: R \rightarrow N
\]

A process must request resource types \( r_i \) in increasing order of enumeration \( F(r_i) \).

- A process can initially request any resource type, say \( r_i \).
- Later, it can request a resource type \( r_j \) iff \( F(r_j) > F(r_i) \).
  [Alternatively, it can be required to release all \( F(r_k) \) such that \( F(r_k) \geq F(r_j) \).]

Proof that this prevents deadlock is easy, follows by transitivity of \( "<" \).
**Deadlock detection:** We have seen that a cycle in a resource-allocation graph may mean that there is a deadlock.

However, if there is no cycle in the graph, there is no deadlock. How do we know this?

To help us, let us define—

A node \( z \) of a graph is a *sink* if there are no edges \((z, b)\) for any node \( b \).

To use resource-allocation graphs to show that a deadlock exists, we need one more property of the system state:

The system state is *expedient* if all processes requesting resources (that are not available) are blocked.

(In such a system, a new allocation of resources can take place only at the time of a request or the time of a release.)

If a resource-allocation graph is expedient, a *knot* is a sufficient condition for deadlock.

The *reachable set* of a node \( z \) is the set of all nodes to which there is a path beginning with \( z \). The reachable set of \( z \) may contain \( z \) if there is a cycle.

A knot \( K \) is a nonempty set of nodes with the property that, for each node \( z \) in \( K \), the reachable set of \( z \) is exactly the knot \( K \).

To see this, note that each process node in a knot is directly or indirectly waiting for other process nodes in the knot to release resources.
Since all are waiting on each other, no process node in the knot can be granted its request.

In considering deadlock detection and avoidance, we often need to (conceptually) remove non-deadlocked processes from the system.

We do this by graph reduction. Informally, we can reduce a graph by removing a process (“reduce graph $G$ by process $p_i$”) if we can show that process $p_i$ can continue execution.

Reduction by a process $p_i$ simulates—

- the acquisition of all of $p_i$’s outstanding requests,
- the return of all units of reusable resources allocated to $p_i$, and
- if $p_i$ is a producer of a consumable resource, the production of a “sufficient” number of units to satisfy all subsequent requests by consumers.

Then the new inventory $a_j$ of that consumable resource $r_j$ is represented by $\infty$ to indicate that all future requests for $r_j$ are grantable.

Obviously, if we can reduce a graph by all of its processes, there is no deadlock.

Formally, a resource-allocation graph may be reduced by a non-isolated node representing an unblocked process $p_i$ in the following manner:

- For each resource $r_j$, delete all edges $(p_i, r_j)$, and if $r_j$ is consumable, decrement $a_j$ by the number of deleted request edges.
- For each resource $r_j$, delete all edges $(r_j, p_i)$. If $r_j$ is reusable, then increment $a_j$ by the number of deleted edges. If $r_j$ is consumable, set $a_j := \infty$.

Reducing a graph by one process can make it reducible by another process. If the graph can be reduced by all its processes, it is called completely reducible.
If a graph is completely reducible, then all its processes can become unblocked. Therefore—

If a graph is completely reducible, then it represents a system state that is not deadlocked.

**Example:** Reduce this resource-allocation graph.

Let us consider deadlock-detection algorithms for various kinds of systems.

**Single-unit requests:** Suppose a system allows a process to have at most 1 outstanding request for a single unit of some resource. Suppose that the system is expedient.

Then there is an efficient algorithm for deadlock detection.

**Lemma:** If the system is deadlocked, then there is a knot in the resource-allocation graph.

**Proof:** Assume that there is no knot. Then each process $p_i$—

- is a sink (i.e., it is not blocked), or
- $\exists$ a path $(p_i, r_j, p_k, \ldots, p_x, r_y, p_z)$ so that node $p_z$ is a sink. (The sink must be a process because of expedience; if it were a resource type, then the resource could and would be granted to the process requesting it.)

Then $p_z$ is not blocked, so we can reduce the graph by $p_z$.

Eventually it will finish using its resources, and thus increase the inventory of $r_y$.

Then $p_x$ will no longer be blocked, and we can reduce the graph by $p_x$.

We can continue up the path in this way, until $p_i$ is unblocked.
Since we can reduce the graph by any process \( p_i \), there is no deadlock.

Therefore, in a single-unit request system,

a knot exists \( \iff \) the system is deadlocked.

Hence, the following algorithm can be used to detect deadlock in such a system.

**Algorithm:** Is a state \( S \) of a resource-allocation graph a deadlocked state?

\[
L := [\text{List of sinks in state } S]
\]

for the next node \( N \) in \( L \) do

for all \( F \) where \( (F, N) \) is an edge do

\[
L := L \parallel F \{ \text{Append node } F \text{ to list } L. \}
\]

\[
\text{deadlock} := L \neq \{\text{set of all nodes}\};
\]

Note that the algorithm starts with a list of sinks, and on each iteration, adds the nodes that are on a path of length 1 away from sinks (or nodes that can reach sinks).

Therefore, when the algorithm has no more nodes from list \( L \) to test, \( L \) will contain all nodes that are not in some knot.

If there are some nodes that are not in \( L \), then these are the processes and resources participating in a deadlock. ✦✦✦

A process \( p \) is not deadlocked iff it is a sink or on a path to a sink. Thus,

**Algorithm:** Is process \( p \) deadlocked?

\[
\text{deadlocked} := \text{true} \{\text{until known otherwise}\};
\]

\[
L := [p];
\]

for the next node \( N \) in \( L \) while \( \text{deadlocked} \) do

remove \( N \) from \( L \);

for all \( F \) where \( (N, F) \) is an edge do

if \( F \) is a sink then \( \text{deadlocked} := \text{false} \)

else if \( (F \) is not in \( L \) then \( L := L \parallel F;\)


© 1998 Edward F. Gehringer

CSC/ECE 501 Lecture Notes
The list $L$ initially contains only $P$. It then travels on all paths from nodes of $L$ to neighbor nodes $F$.

If the neighbor is a sink, the algorithm terminates, reporting no deadlock.

If the entire list is processed and no sink is found, then $p$ is part of a knot and is deadlocked.

Since there are $n$ processes and $m$ resources, the algorithms execute in $O(mn)$ time.

The second algorithm runs more quickly than the first. Why?

Therefore, the second algorithm can be executed whenever $p$ requests an unavailable resource unit.

However, it may still be worthwhile to execute the first algorithm “occasionally.”

**Systems with only reusable resources:** We will establish that a state is deadlock-free by reducing its resource allocation graph.

Note that it does not matter what node(s) we reduce first. Either—

- the graph is completely reducible, or
- the same irreducible subgraph will be left after the graph has been reduced as much as possible.

This follows from the fact that a reduction never decreases the available inventory of a resource. (This is true iff the graph contains only reusable resources.)

So if the graph is reducible by, say $p_i$ first, and then $p_j$, it could also be reduced by $p_j$ and then by $p_i$.

Assume that the state $S$ of the system is not deadlocked.

- Then no process in the system is deadlocked.
- Thus any sequence of reductions will leave all processes unblocked.
- Therefore the graph is completely reducible.
Earlier (p. 117) we showed that if a graph is completely reducible, the state is not deadlocked. Thus we have established—

State not deadlocked \( \equiv \) Graph completely reducible.

This allows us to test for deadlock by checking whether the resource-allocation graph is reducible.

The algorithm is very straightforward.

**Algorithm:** Simple algorithm for deadlock detection in reusable systems.

\[
L := \text{[List of non-isolated process nodes]};
\]
\[
finished := false;
\]
\[
\textbf{while } ( (L \neq \emptyset) \textbf{ and not } finished ) \textbf{ do}
\]
\[
\begin{align*}
\text{begin} \\
& p := \text{first process in } L \text{ by which} \\
& \quad \text{graph can be reduced}; \\
& \text{if } p \neq \text{nil } \textbf{then begin} \\
& \quad \text{Reduce graph by } p; \\
& \quad \text{Remove } p \text{ from } L; \\
& \textbf{end} \\
& \textbf{else } finished := true; \\
\textbf{end;}
\end{align*}
\]
\[
\text{deadlock} := L \neq \emptyset;
\]

Assume there are \( n \) processes and \( m \) resource types in the system. What is the time complexity of this algorithm?

- What is the most resource nodes that can be involved in a reduction?
- How many iterations of the while loop are (may be) needed?
- Are there any other sources of complexity?

Therefore the complexity of this algorithm is
How can we make this algorithm more efficient? We need to avoid the search for a process by which the graph can be reduced.

Recall: A graph can be reduced by a process $p$ iff the process can acquire all the resource units it is requesting.

- For each resource, we keep a list ($\text{ordered_requests}[r]$) of the processes that need it.
- On this list, the processes are ordered by increasing need for the resource.

Thus, whenever a resource is released, without searching we can tell which processes’ needs can be satisfied.

For each process, we also keep track of the number of resources it “still needs”—i.e., the number of resource types for which its request cannot be satisfied.

This is called $\text{wait\_count}[p]$, for each process $p$.

Whenever the graph is reduced by a process, that process may release enough resources to satisfy process $p_i$’s request for other resource types. Whenever that happens, $\text{wait\_count}[p_i]$ is decremented.

(Note: The order in which the graph is reduced represents one order that the processes could finish—not necessarily the order they will finish. Reduction does not involve predicting the future!)

The algorithm below uses a list $\text{list\_to\_be\_reduced}$ that contains processes that are known not to be deadlocked.
**Algorithm:** Algorithm for deadlock detection in reusable systems, with ordered request lists.

\[
L := \{\text{List of non-isolated process nodes}\};
\]

\[
\text{list_to_be_reduced} := \{\text{List of non-isolated process nodes whose wait_count is 0}\};
\]

\[
\text{while list_to_be_reduced} \neq \emptyset \text{ do begin}
\]

Remove a process \( p_i \) from \( \text{list_to_be_reduced} \);

\[
\text{for all } r_j \in \{\text{resources assigned to } p_i\} \text{ do begin}
\]

Increase available units of \( r_j \) by the number of units assigned to \( p_i \);

\[
\text{for all } p_k \text{ in } \text{ordered_requests}[r_j] \text{ whose request can be satisfied do begin}
\]

\[
\text{wait_count}[p_k] := \text{wait_count}[p_k] - 1;
\]

\[
\text{if } \text{wait_count}[p_k] = 0 \text{ then}
\]

Add \( p_k \) to \( \text{list_to_be_reduced} \);

\[
\text{end;}
\]

\[
\text{end;}
\]

Remove \( p_i \) from \( L \);

\[
\text{end;}
\]

\[
\text{deadlock} := L \neq \emptyset;
\]

Since the original deadlock-detection algorithm had only one loop and this has three nested loops, how can this one be faster?

### Example
Consider a system of three processes, \( p_1, p_2, \) and \( p_3 \), and three reusable resources, \( r_1, r_2, \) and \( r_3 \).

1. Initial state

<table>
<thead>
<tr>
<th></th>
<th>( p_1 )</th>
<th>( p_2 )</th>
<th>( p_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{wait_count} )</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
\text{ordered_requests}
\]

\[
\emptyset \quad \emptyset \quad \emptyset
\]

\[
\bullet \quad \bullet \quad \bullet
\]

\[
\bullet \quad \bullet \quad \bullet
\]

\[
\bullet \quad \bullet \quad \bullet
\]
2. $p_1$ requests and acquires two units of $r_3$.

\[
\begin{array}{ccc}
  p_1 & p_2 & p_3 \\
  \text{holds:} & r_3 & r_3 & \emptyset & \emptyset \\
  \text{wait_count} & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
  r_1 & r_2 & r_3 \\
  \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\end{array}
\]

Ordered Requests:

\[
\begin{array}{cccc}
  r_1 & r_2 & r_3 \\
  \emptyset & \emptyset & \emptyset \\
\end{array}
\]

The deadlock-detection algorithm is not applied, since all wait_counts are 0.

3. $p_3$ requests and acquires two units of $r_2$.

\[
\begin{array}{ccc}
  p_1 & p_2 & p_3 \\
  \text{holds:} & r_3 & r_3 & \emptyset & r_2 & r_2 \\
  \text{wait_count} & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
  r_1 & r_2 & r_3 \\
  \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\end{array}
\]

Ordered Requests:

\[
\begin{array}{cccc}
  r_1 & r_2 & r_3 \\
  \emptyset & \emptyset & \emptyset \\
\end{array}
\]

4. $p_3$ requests two units of $r_3$.

\[
\begin{array}{ccc}
  p_1 & p_2 & p_3 \\
  \text{holds:} & r_3 & r_3 & \emptyset & r_2 & r_2 \\
  \text{wait_count} & 0 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
  r_1 & r_2 & r_3 \\
  \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\end{array}
\]

Ordered Requests:

\[
\begin{array}{cccc}
  r_1 & r_2 & r_3 \\
  \emptyset & \emptyset & \emptyset \\
\end{array}
\]

{p_3}

The deadlock-detection algorithm is applied with $L = \{p_1, p_3\}$ and list_to_be_reduced = \{p_1\}.

The algorithm terminates in one pass with the conclusion that deadlock does not exist.

5. $p_2$ requests and acquires two units of $r_1$.

\[
\begin{array}{ccc}
  p_1 & p_2 & p_3 \\
  \text{holds:} & r_3 & r_3 & r_1 & r_2 & r_2 \\
  \text{wait_count} & 0 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
  r_1 & r_2 & r_3 \\
  \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\end{array}
\]

Ordered Requests:

\[
\begin{array}{cccc}
  r_1 & r_2 & r_3 \\
  \emptyset & \emptyset & \emptyset \\
\end{array}
\]

{p_3}
6. $p_2$ requests two units of $r_2$.

\[
\begin{array}{cccc}
& p_1 & p_2 & p_3 \\
\text{holds:} & r_3 & r_3 & r_1 & r_2 & r_2 \\
\text{wait_count} & 0 & 1 & 1 \\
\end{array}
\]

The deadlock-detection algorithm is applied with $L = \{p_1, p_2, p_3\}$ and $\text{list_to_be_reduced} = \{p_1\}$. What does the algorithm do?

7. $p_1$ requests a unit of $r_1$.

\[
\begin{array}{cccc}
& p_1 & p_2 & p_3 \\
\text{holds:} & r_3 & r_3 & r_1 & r_2 & r_2 \\
\text{wait_count} & 1 & 1 & 1 \\
\end{array}
\]

The deadlock-detection algorithm is applied with $L = \{p_1, p_2, p_3\}$ and $\text{list_to_be_reduced} = \emptyset$. What happens?

Since only unsatisfiable requests can cause deadlock, this algorithm only needs to be run at request time.

Consequently, it detects deadlock immediately (after the request that caused the deadlock).

Therefore, the detection algorithm needs to be run only until $\text{wait_count}$ of the requesting process = 0.

**Single instance of each resource type**: The time complexity of the fast deadlock-detection algorithm is $O(mn \log n)$. Graphical methods can achieve $O(n^2)$ if there is only one instance of each resource type.

Define a wait-for graph, which is a resource-allocation graph, modified as follows:

- Resource nodes are removed.
- Edges are “collapsed” so that an edge $(p_i, p_k)$ exists if and only if the resource-allocation graph contained edges $(p_i, r_j)$ and $(r_j, p_k)$ for some resource $r_j$. 

© 1998 Edward F. Gehringer  
CSC/ECE 501 Lecture Notes  
Page 124
• A deadlock exists iff the wait-for graph contains a cycle.
• Cycles can be detected by $O(n^2)$ algorithm.
• If deadlock does exist, it can be eliminated by selecting a single victim.

**Deadlock avoidance (in reusable-resource systems):** Requires some additional information on resource requests—for example, we can require each process to declare the *maximum* amount of each resource that it will ever need.

The concept of a *safe state* is important. Here, a resource-allocation *state* consists of—

- the number of allocated and available resources, and
- the maximum *claim* (or maximum demands) of each process.

A state is *safe* if the system can—

- allocate resources to each process (up to its maximum), and
- still avoid a deadlock.

*Example:* Suppose the system has three processes $p_1$, $p_2$, and $p_3$, and twelve units of one resource (e.g., tape drives).
State 1: Process Allocated Maximum Need

\[
\begin{array}{ccc}
\text{State } 1 & & \\
& p_1 & 1 & 4 \\
& p_2 & 4 & 6 \\
& p_3 & 5 & 8 \\
\end{array}
\]

Available: 2

This state is safe, because requests of \(p_1\), \(p_2\), and \(p_3\) can be satisfied, in that order.

State 2: Process Allocated Maximum Need

\[
\begin{array}{ccc}
\text{State } 2 & & \\
& p_1 & 8 & 10 \\
& p_2 & 2 & 5 \\
& p_3 & 1 & 3 \\
\end{array}
\]

Available: 1

This state is unsafe: regardless of which process is next granted the available drive, we can’t guarantee that all 3 processes will finish.

Notice that

\[
\text{unsafe state} \Rightarrow \text{deadlock will occur.}
\]

It just implies that \textit{some unfortunate sequence} of requests might lead \textit{unavoidably} to deadlock.

“Unavoidably” \Rightarrow we can’t avoid deadlock by choosing to delay some processes’ resource requests.

To avoid entering an unsafe state, we must carefully consider resource requests before granting them. For example, suppose the system is in State 1 (safe state).
Now suppose $p_3$ requests and is granted an additional tape drive:

State 3: Process | Allocated | Maximum Need
---|---|---
$p_1$ | 1 | 4
$p_2$ | 4 | 6
$p_3$ | 6 | 8
Available: 1

Why is this now an unsafe state?

**Example:** Assume a maximum-claim reusable-resource system with four processes and three resource types. The maximum-claim matrix is given by

$$\begin{bmatrix}
4 & 1 & 4 \\
3 & 1 & 4 \\
4 & 7 & 12 \\
1 & 1 & 6
\end{bmatrix}$$

where $Max_{ij}$ denotes the maximum claim of process $i$ for resource $j$. The total number of units of each resource type is given by the vector $(4 \ 8 \ 16)$. The allocation of resources is given by the matrix

$$\begin{bmatrix}
0 & 1 & 4 \\
2 & 0 & 1 \\
1 & 2 & 1 \\
0 & 0 & 3
\end{bmatrix}$$

where $A_{ij}$ denotes the number of units of resource $j$ that are currently allocated to process $i$.

Processes are numbered $p_1$ through $p_4$, and resources are numbered $r_1$ through $r_3$.

Suppose process 1 requests 1 more unit of resource 1.
Then, after the allocation is made, the matrix $A$ is

$$
\begin{bmatrix}
1 & 1 & 4 \\
2 & 0 & 1 \\
1 & 2 & 1 \\
0 & 0 & 3
\end{bmatrix}
$$

and the maximum-remaining demand matrix

$$
Max - A = \begin{bmatrix}
3 & 0 & 0 \\
1 & 1 & 3 \\
3 & 5 & 11 \\
1 & 1 & 3
\end{bmatrix}
$$

The $Avail$ vector is $[0 \ 5 \ 7]$. The state is unsafe, because the system cannot guarantee to meet the maximum demands of any process.

For example, if all of the processes request an extra unit of resource 1 before any process releases a resource 1, all the processes will be deadlocked.

They are not deadlocked now, because they might release their resources before demanding more.

Now, suppose process 4 requests 3 more units of resource 3. The processes can finish in the order $p_4, p_2, p_3, p_1$. To see this, note that after the allocation is made, the matrix $A$ is

$$
\begin{bmatrix}
0 & 1 & 4 \\
2 & 0 & 1 \\
1 & 2 & 1 \\
0 & 0 & 6
\end{bmatrix}
$$

and the maximum-remaining demand matrix

$$
Max - A = \begin{bmatrix}
4 & 0 & 0 \\
1 & 1 & 3 \\
3 & 5 & 11 \\
1 & 1 & 0
\end{bmatrix}
$$

The $Avail$ vector is $[1 \ 5 \ 4]$. The remaining demand of $p_4$ is $\leq Avail$, so $p_4$ may complete.
Now \textit{Avail} is $[1 \ 5 \ 10]$. The remaining demand of $p_2$ is $\leq \text{Avail}$, so $p_2$ may complete.

Now \textit{Avail} is $[3 \ 5 \ 11]$. The remaining demand of $p_3$ is $\leq \text{Avail}$, so $p_3$ may

Now \textit{Avail} is $[4 \ 7 \ 12]$. The remaining demand of $p_1$ is $\leq \text{Avail}$, so $p_1$ may complete. Thus the state is safe.

\textit{Algorithm for deadlock avoidance (Banker’s algorithm)}: The deadlock-avoidance algorithm is very similar to the deadlock-detection algorithm, but it uses the processes’ \textit{maximum claim} rather than their current allocations.

Let us introduce a new kind of edge, a \textit{claim edge}, represented by a dashed line. Like a request edge, a claim edge points from a process to a resource type.

\begin{itemize}
  \item A claim edge indicates that a process $p_i$ \textit{may request} the resource $r_j$ some time \textit{in the future}.
  \item All claim edges for $p_i$ must be present before $p_i$ starts executing.
  \item Thus, a request edge $(p_i, r_j)$ may be added only if a claim edge $(p_i, r_j)$ is already present.
\end{itemize}

A resource-allocation graph with claim edges is called a \textit{maximum-claim graph}. It reflects the projected worst-case future state in resource allocation.

A state is safe iff its corresponding maximum-claim graph is deadlock free.
Whenever a process makes a request, the following algorithm is executed:

**Algorithm:** Deadlock avoidance (Banker’s algorithm).

1. Project the future state by changing the request edge to an assignment edge.

2. Construct the maximum-claim graph for this state and analyze it for deadlock. If deadlock would exist, defer the request. Otherwise grant the request.

**Example:** Suppose we are given the following system. The process’s maximum claims are represented as a *claim matrix* $C$, where $c_{ij}$ is the maximum claim of process $p_i$ for units of type $r_j$.

$$C = \begin{bmatrix} 1 & 2 \\ 2 & 2 \end{bmatrix}$$

Suppose $p_1$ requests one unit of $r_2$. Can this request be granted? To see, “grant” the request, then construct the maximum-claim graph and see if it contains a deadlock:

Suppose that process $p_2$ then requests a unit of $r_1$. Can this request now be granted?
**Single instance of each resource type**: The deadlock avoidance algorithm can require $O(mn^2)$ operations. If there is only one instance of each resource type, an $O(mn)$ algorithm can be devised.

**Algorithm**: Deadlock-avoidance with a single instance of each resource type.

- Assume we are given a maximum-claim graph.
- Then when a process makes a request, it can be granted only if converting the request edge $(p_i, r_j)$ to an assignment edge $(r_j, p_i)$ does not create a cycle in the resource-allocation graph. (The detection of a cycle takes time $O(mn)$.)

For example, in the diagram below, we cannot grant $r_2$ to $p_2$. Why?

In general, deadlock avoidance is more expensive than deadlock detection:

- The algorithm must be executed for every request prior to granting it.
• It restricts resource utilization, so it may degrade system performance. (Assumes that the “worst-case” request sequence will come true.)

This is especially severe if the claims are not precisely known.

**Deadlock recovery:** Several methods can be used.

• Abort one or more processes to break the circular-wait condition.

  *Considerations:* Priority, used computing time, expected remaining time, etc.

• Preempt resources from one or more processes.

• “Roll back” a process. Sometimes also called “checkpoint/restart.” “Snapshots” of process state saved at periodic intervals, usually specified by the process itself. To break a deadlock,
  ◦ Process is aborted.
  ◦ Later resumed at the previous checkpoint.

Rarely used except for very long processes.

Also useful in dealing with system crashes.

In general, *victims* are selected for deadlock recovery on the basis of priority. May also depend on how “close” a process is to finishing.

**Combined approach to deadlock handling:**

• Order resource *classes* (not resource types), e.g.,
  ◦ Internal resources (resources used by operating system, including process-control blocks).
  ◦ Central memory.
  ◦ Peripherals (tape drives, printers, etc.).
  ◦ Swappable space (secondary storage required for process).

• If a deadlock does occur, it must involve only one *class* of resources. Strategy for recovery depends on which class is involved.
For example, assume that the four resource classes are ordered as shown above, and

- For internal resources, resource ordering can be used (run-time choices between pending requests are unnecessary).
- Central memory. Preemption can be used (by suspending the process).
- Peripherals. In a batch system, avoidance can be used, using information from control cards.
- Swappable space. Preallocation can be used, if maximum storage requirements are usually known.

**Deadlocks—present and future:** Deadlock is becoming a more important problem, because

- Distributed processing: Resources may be shared over large networks ⇒ more possibility for deadlock.
- Processes will be interactive ⇒ less possibility of system (or user) knowing resource needs in advance.
- User-friendly interfaces will not want to bother users for maximum usage information.
- Large databases ⇒ possibility of deadlock for *portions* of a database.
Memory management: Memory is a device for storing information. Its function is to return the value associated with a particular name used by the programmer.

The name space: The programmer refers to values by means of

- An unfixed collection of logical names.
- Symbolic rather than numeric.
- Have no inherent order.
- “Lengths” can vary.
- Same name can be used by other programmers.

The name space is the set of names used by the programmers who wrote the code for a single process.

The system must provide a mapping between the set of names used by the programmer and the set of values which the program uses.

Each name can be mapped (at any given time) to a unique value.

Both the software and the hardware play a role in performing this mapping:

- The compiler (or assembler) translates the name to a logical address.
  The set of all logical addresses is known as the logical address space.

- The logical address is translated to a physical address. The set of all physical addresses is known as the

This translation may be performed by

- the linker,
- the loader, and/or
- the relocation hardware (and index registers) in the computer itself.

This process is known as address translation.
• Given a physical address, the memory either retrieves or stores a value.

Our discussion of address mapping will focus on two aspects—

• The software that performs this mapping (linkers and loaders).
• The hardware that performs this mapping (we will consider several computer architectures).

Loaders: The mapping of names to values can take place at 4 times:

1. Compile time 3. Load time
2. Link time 4. Execution time

1. The compiler (or assembler) translates a program from a source language to a machine language.

2. The linker takes several separately compiled machine-language programs and combines them so that they may be executed as a unit.

3. The loader takes a machine-language program and places it in memory so that it can be executed.

4. At execution time, further translation may take place, if the loader has not produced absolute physical addresses.

(There is a good discussion of these in S&G, pp. 240–241, though it counts linking as part of the loading process.)

Sometimes,

• steps 2 and 3 are both performed by a linking loader; or
• steps 1, 2, and 3 are performed by a load-and-go compiler.

Also,

• step 2 is not necessary if the compiled program is to be run without other programs.

The simplest kind of loader, an absolute loader, just

• takes a program from secondary storage,
• places it in memory, word by word, then
• transfers control to its first instruction.
What is the most obvious limitation of this kind of loader?

In general, the user does not know \textit{a priori} where the program will reside in memory. A \textit{relocating loader} is capable of loading a program to begin anywhere in memory:

The addresses produced by the compiler run from 0 to $L - 1$. After the program has been loaded, the addresses must run from $N$ to $N + L - 1$.

Therefore, the relocating loader adjusts, or \textit{relocates}, each \textit{address} in the program.

\begin{equation}
\begin{array}{c}
0 \\
L - 1 \\
\text{Object program (logical addresses)} \\
\hline
\hline
\text{Main memory (physical addresses)} \\
N \\
N + L - 1
\end{array}
\end{equation}

Fields that are relocated are called \textit{relative}; those which are not relocated are called \textit{absolute}.

Which of these are relative?

- Opcodes
- Register numbers
- Direct addresses
- Shift amounts
- Immediate operands

On the next page is a sample program (Calingaert), together with its object code.
In that program, what address *expressions* need to be relocated?

How about—

- FRONT+7
- FRONT–FINAL
- FRONT+FINAL–LIMIT

Address expressions normally may include addition and subtraction, but not multiplication or division by other addresses.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>COPY</td>
<td>ZERO</td>
<td>OLDER</td>
<td></td>
<td>00</td>
<td>3</td>
<td>011</td>
<td>13 33 35</td>
<td>40</td>
<td>13 73 75</td>
</tr>
<tr>
<td>COPY</td>
<td>ONE</td>
<td>OLD</td>
<td></td>
<td>03</td>
<td>3</td>
<td>011</td>
<td>13 34 36</td>
<td>43</td>
<td>13 74 76</td>
</tr>
<tr>
<td>READ</td>
<td>LIMIT</td>
<td></td>
<td></td>
<td>06</td>
<td>2</td>
<td>01</td>
<td>12 38</td>
<td>46</td>
<td>12 78</td>
</tr>
<tr>
<td>WRITE</td>
<td>OLD</td>
<td></td>
<td></td>
<td>08</td>
<td>2</td>
<td>01</td>
<td>08 36</td>
<td>48</td>
<td>08 76</td>
</tr>
<tr>
<td>FRONT</td>
<td>LOAD</td>
<td>OLDER</td>
<td></td>
<td>10</td>
<td>2</td>
<td>01</td>
<td>03 35</td>
<td>50</td>
<td>03 75</td>
</tr>
<tr>
<td>ADD</td>
<td>OLD</td>
<td></td>
<td></td>
<td>12</td>
<td>2</td>
<td>01</td>
<td>02 36</td>
<td>52</td>
<td>02 76</td>
</tr>
<tr>
<td>STORE</td>
<td>NEW</td>
<td></td>
<td></td>
<td>14</td>
<td>2</td>
<td>01</td>
<td>07 37</td>
<td>54</td>
<td>07 77</td>
</tr>
<tr>
<td>SUB</td>
<td>LIMIT</td>
<td></td>
<td></td>
<td>16</td>
<td>2</td>
<td>01</td>
<td>06 38</td>
<td>56</td>
<td>06 78</td>
</tr>
<tr>
<td>BRPOS</td>
<td>FINAL</td>
<td></td>
<td></td>
<td>18</td>
<td>2</td>
<td>01</td>
<td>01 30</td>
<td>58</td>
<td>01 70</td>
</tr>
<tr>
<td>WRITE</td>
<td>NEW</td>
<td></td>
<td></td>
<td>20</td>
<td>2</td>
<td>01</td>
<td>08 37</td>
<td>60</td>
<td>08 77</td>
</tr>
<tr>
<td>COPY</td>
<td>OLD</td>
<td>OLDER</td>
<td></td>
<td>22</td>
<td>3</td>
<td>011</td>
<td>13 36 35</td>
<td>62</td>
<td>13 76 75</td>
</tr>
<tr>
<td>COPY</td>
<td>NEW</td>
<td>OLD</td>
<td></td>
<td>25</td>
<td>3</td>
<td>011</td>
<td>13 37 36</td>
<td>65</td>
<td>13 77 76</td>
</tr>
<tr>
<td>BR</td>
<td>FRONT</td>
<td></td>
<td></td>
<td>28</td>
<td>2</td>
<td>01</td>
<td>00 10</td>
<td>68</td>
<td>00 50</td>
</tr>
<tr>
<td>FINAL</td>
<td>WRITE</td>
<td>LIMIT</td>
<td></td>
<td>30</td>
<td>2</td>
<td>01</td>
<td>08 38</td>
<td>70</td>
<td>08 78</td>
</tr>
<tr>
<td>STOP</td>
<td></td>
<td></td>
<td></td>
<td>32</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>72</td>
<td>11</td>
</tr>
<tr>
<td>ZERO</td>
<td>CONST</td>
<td>0</td>
<td></td>
<td>33</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>73</td>
<td>00</td>
</tr>
<tr>
<td>ONE</td>
<td>CONST</td>
<td>1</td>
<td></td>
<td>34</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>74</td>
<td>01</td>
</tr>
<tr>
<td>OLDER</td>
<td>SPACE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OLD</td>
<td>SPACE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEW</td>
<td>SPACE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LIMIT</td>
<td>SPACE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Also, (the number of addresses added) \(-\) (the number of addresses subtracted) must = 0 or 1.

Therefore, FRONT\(+\)FINAL\(–\)LIMIT is legal, but FRONT\(+\)FINAL\(+\)LIMIT and FRONT\(–\)FINAL\(–\)LIMIT are not.

*Relocation bits:* Associated with each address in the program (and sometimes with each opcode too!) is a *relocation bit*, telling whether it needs to be relocated.

\[
\begin{align*}
0 &\rightarrow \text{No.} \\
1 &\rightarrow \text{Yes.}
\end{align*}
\]

The relocation bits may either

- immediately follow each instruction, or
- be collected together into a single contiguous “relocation map” which follows the text of the object code, or
- a compromise: divide the object code into fixed-length chunks and group as many relocation bits with each chunk as there are potentially relative fields in a chunk of code.

What are the advantages and disadvantages of each of these methods?

- If relocation bits (and length fields) are interleaved with the program text, it cannot be read directly into the storage locations it will occupy.
  
  The text needs to be handled in small units of variable length.

- If the relocation bits are collected in a relocation map, the text and relocation bits can be moved as a block. But a lot of extra memory may be needed to hold the map.

- With fixed-length chunks, one chunk plus its relocation bits can be moved as a block. Then the relocation bits are used to relocate the current chunk. The relocation bits may then be overwritten by the next chunk.

Some computers have *base-register addressing*, that is, they implicitly add the contents of a base register \(B\) to each address:

\[
\text{Physical address} = \text{logical address} + \text{contents of } B.
\]
Then, the program normally incorporates instructions to load the base registers with a procedure origin when the procedure is entered.

The loader rarely needs to relocate an address, since relocation is performed at execution time.

Base-register addressing is sometimes called *dynamic relocation*, as opposed to the *static relocation* performed by a loader.

If more than one base register is used, the program can address more than one region of memory:

One advantage of using base registers: The program can be run if there is enough free memory to hold it, even if that memory is not all in one large block.

What (few) addresses need to be relocated by the linker or loader when base-register addressing is used?

In the program below, note that a base register will be loaded with the value 15, which is the start address of the program.
need to be relocated! Since there are not very many of them, relocation bits are not used.

(However, address constants only need to be relocated in architectures that do not automatically relocate all addresses relative to some base, as we shall see.)

Instead, the assembler appends a relocation list, consisting of one entry for each address constant.

**Linkers:** If a program has been compiled in more than one part, the parts must be linked together before they can be executed.

*The problem:* Suppose two parts of a program have been compiled separately.

- Part I uses logical addresses $0, \ldots, x_1$, and
- Part II uses logical addresses $0, \ldots, x_2$.

There must be some way to keep them from overwriting each other.

Linking of separately compiled procedures or modules can be accomplished in at least two different ways.

- Direct linking. A piece of software known as a *linker* fills in addresses of externally referenced symbols. (It fills them in in the code for each procedure.)
The assembler or compiler has prepared

- A *dictionary* (or “definition table”), which lists each internally defined symbol (1 entry/symbol) that is used by other modules.
  
  (symbol name, relative address, relocatability mode)

- An *external-reference list* (or “use table”), which lists each internally used non-local symbol (1 entry/occurrence).
  
  (symbol name, relative address)

At compile time, translation of externally defined symbols must be incomplete. (They are temporarily assigned address zero and absolute mode.)

*The action of a two-pass direct linker:*

- Pass 1: Merge the definition tables.
  
  Modify the address of each relative symbol.

- Pass 2: External references are patched by means of the use table.

It is also possible to write a one-pass linker, if chains are kept pointing to each symbol as it is encountered. Similar to techniques used in compiler symbol tables.

The linker and loader are often combined to form a *linking loader*, since both need to read all procedures and inspect the relocation bits.

To summarize,

- A linker prepares relocatable code with interspersed relocation information.

- A linking loader produces executable code with no insertions.

*Example:* Consider the two modules on the next page. The first uses two external references (defined in assembly code via INTUSE) and defines four external symbols (via INTDEF).
The second uses four external references and defines two symbols that will be referenced from outside.

**Source Program**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Mode</th>
<th>Source Program</th>
<th>Object Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROG2</td>
<td>11</td>
<td>+</td>
<td>START 0</td>
<td>10 01 a 00 a</td>
</tr>
<tr>
<td>TABLE</td>
<td>21</td>
<td>+</td>
<td>INTUSE TEST</td>
<td>12 03 a 30 r</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>INTDEF RESUMEPT</td>
<td>14</td>
</tr>
<tr>
<td>TEST</td>
<td></td>
<td></td>
<td>INTDEF HEAD</td>
<td>20 03 a 02 a</td>
</tr>
<tr>
<td>RESUMEPT</td>
<td>21</td>
<td>+</td>
<td>TABLE+2 LOAD</td>
<td>22</td>
</tr>
<tr>
<td>HEAD</td>
<td></td>
<td></td>
<td>STORE TABLE+HEAD–TEST</td>
<td>29 11 a</td>
</tr>
<tr>
<td>LIST</td>
<td>37</td>
<td></td>
<td>STOP LIST</td>
<td>30 37 a</td>
</tr>
<tr>
<td>END</td>
<td></td>
<td></td>
<td>HEAD LIST</td>
<td>31</td>
</tr>
</tbody>
</table>

**Use Table**

**Definition Table**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Sign</th>
<th>Symbol</th>
<th>Addr</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROG2</td>
<td>11</td>
<td>+</td>
<td>PROG1</td>
<td>00</td>
<td>r</td>
</tr>
<tr>
<td>TABLE</td>
<td>21</td>
<td>+</td>
<td>TEST</td>
<td>10</td>
<td>r</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RESUMEPT</td>
<td>12</td>
<td>r</td>
</tr>
<tr>
<td>HEAD</td>
<td></td>
<td></td>
<td>HEAD</td>
<td>22</td>
<td>r</td>
</tr>
</tbody>
</table>

**Source Program**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Mode</th>
<th>Source Program</th>
<th>Object Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROG2</td>
<td>START 0</td>
<td>INTUSE TABLE</td>
<td>Addr Word</td>
<td>M Word</td>
</tr>
<tr>
<td>TEST</td>
<td>INTDEF</td>
<td>INTUSE</td>
<td>STORE TABLE+HEAD–TEST</td>
<td>15 07 a 27 r</td>
</tr>
<tr>
<td>RESUMEPT</td>
<td>INTUSE</td>
<td>INTUSE</td>
<td>BR RESUMEPT</td>
<td>25 00 a 00 a</td>
</tr>
<tr>
<td>HEAD</td>
<td>INTUSE</td>
<td>INTUSE</td>
<td>TABLE SPACE</td>
<td>27 XX a</td>
</tr>
<tr>
<td>LIST</td>
<td>SPACE</td>
<td>SPACE</td>
<td>SPACE</td>
<td>28 XX a</td>
</tr>
<tr>
<td>TWO</td>
<td>SPACE</td>
<td>SPACE</td>
<td>SPACE</td>
<td>29 XX a</td>
</tr>
<tr>
<td>ADDRTEST</td>
<td>CONST 2</td>
<td>A(TEST)</td>
<td>30 02 a</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
<td></td>
<td>END</td>
<td>31 00 a</td>
</tr>
</tbody>
</table>

**Use Table**

**Definition Table**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Sign</th>
<th>Symbol</th>
<th>Addr</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEAD</td>
<td>16</td>
<td>+</td>
<td>PROG2</td>
<td>00</td>
<td>r</td>
</tr>
<tr>
<td>TEST</td>
<td>16</td>
<td>–</td>
<td>TABLE</td>
<td>27</td>
<td>r</td>
</tr>
<tr>
<td>RESUMEPT</td>
<td>26</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEST</td>
<td>31</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A global symbol table is prepared by pass 1 (below). Then external references are resolved during pass 2.

**Second segment after Pass 1 of linker:**

<table>
<thead>
<tr>
<th>Source Program</th>
<th>Object Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROG2</td>
<td>START 0</td>
</tr>
<tr>
<td></td>
<td>INTDEF TABLE</td>
</tr>
<tr>
<td>TEST</td>
<td>INTUSE</td>
</tr>
<tr>
<td>RESUMEPT</td>
<td>INTUSE</td>
</tr>
<tr>
<td>HEAD</td>
<td>INTUSE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addr</th>
<th>Word</th>
<th>M</th>
<th>Word</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORE TABLE+HEAD–TEST</td>
<td>46</td>
<td>07</td>
<td>a</td>
<td>58</td>
</tr>
<tr>
<td>BR RESUMEPT</td>
<td>56</td>
<td>00</td>
<td>a</td>
<td>00</td>
</tr>
<tr>
<td>TABLE SPACE</td>
<td>58</td>
<td>XX</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>SPACE</td>
<td>59</td>
<td>XX</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>SPACE</td>
<td>60</td>
<td>XX</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>TWO CONST</td>
<td>61</td>
<td>02</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>ADDRTST</td>
<td>CONST</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END A(TEST)</td>
<td>62</td>
<td>00</td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use Table</th>
<th>Global Symbol Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>Addr</td>
</tr>
<tr>
<td>HEAD</td>
<td>47</td>
</tr>
<tr>
<td>TEST</td>
<td>47</td>
</tr>
<tr>
<td>RESUMEPT</td>
<td>57</td>
</tr>
<tr>
<td>TEST</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Indirect linking.** Essentially the same, except that pointers into the *external-reference list*, rather than addresses, are placed in the code. The linker places absolute addresses into the external-reference list.

External references proceed *indirectly*, through the external-reference list.

- Disadvantages: References slower, but perhaps this is not too serious if only addresses of called procedures need to be referenced in this way.

The external reference lists must be kept around at execution time.
• Advantage: Linking is much faster, since only the external reference lists need to be modified, not the code. (But advantage wanes if a linking loader is used.)

Program after Pass 2 of linker:

<table>
<thead>
<tr>
<th>Source Program</th>
<th>Object Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROG1</td>
<td>START 0</td>
</tr>
<tr>
<td>PROG2</td>
<td>INTUSE</td>
</tr>
<tr>
<td>TABLE</td>
<td>INTUSE</td>
</tr>
<tr>
<td></td>
<td>INTDEF TEST</td>
</tr>
<tr>
<td></td>
<td>INTDEF RESUMEPT</td>
</tr>
<tr>
<td></td>
<td>INTDEF HEAD</td>
</tr>
<tr>
<td>TEST</td>
<td>BRPOS PROG2 10 01 a 31 r</td>
</tr>
<tr>
<td>RESUMEPT</td>
<td>LOAD LIST 12 03 a 30 r</td>
</tr>
<tr>
<td>HEAD</td>
<td>LOAD TABLE+2 20 03 a 60 r</td>
</tr>
<tr>
<td></td>
<td>22</td>
</tr>
<tr>
<td>LIST</td>
<td>CONST 37</td>
</tr>
<tr>
<td>PROG2</td>
<td>START 0</td>
</tr>
<tr>
<td>TEST</td>
<td>INTUSE</td>
</tr>
<tr>
<td>RESUMEPT</td>
<td>INTUSE</td>
</tr>
<tr>
<td>HEAD</td>
<td>INTUSE</td>
</tr>
<tr>
<td></td>
<td>STORE TABLE+HEAD–TEST 46 07 a 70 r</td>
</tr>
<tr>
<td></td>
<td>48</td>
</tr>
<tr>
<td>TABLE</td>
<td>BR RESUMEPT 56 00 a 12 r</td>
</tr>
<tr>
<td></td>
<td>SPACE 58 XX a</td>
</tr>
<tr>
<td></td>
<td>SPACE 59 XX a</td>
</tr>
<tr>
<td></td>
<td>SPACE 60 XX a</td>
</tr>
<tr>
<td>TWO</td>
<td>CONST 2</td>
</tr>
<tr>
<td>ADDRTEST</td>
<td>CONST A(TEST) 62 10 r</td>
</tr>
</tbody>
</table>

• Dynamic linker. It is often preferable to delay linking certain modules or procedures until execution time. (E. g., error-handling routines.) Usually used with segmented virtual memory.

We will describe dynamic linking after we describe segmented memory.
Memory models and computer architectures: We will describe several architectures that have been developed to allow more than one program to share memory, and discuss the problems of memory management in each.

Architecture I: Basic von Neumann architecture. Represented most recently by 8-bit microcomputers.

- The address space is of fixed size; no address translation. (Logical address = physical address.)

Hence, any process can write anywhere in physical memory.

- May have index registers, which implement the simplest kind of address modification. They are needed for addressing array elements.

An indexed address is a pair \((i, x)\), where \(i\) denotes an index register and \(x\) denotes the displacement. Then

\[
\text{Physical address} = (\text{contents of index register } i) + x
\]

Problems not solved by the basic von Neumann architecture:

- Size of operating system. When the size of the OS changes, all programs may have to be recompiled.

- Size of address space. Programs cannot be larger than physical memory, since there is no way to generate an address larger than the maximum physical address.

- Protection. A program can write anywhere in physical memory.

- Sharing of code. Two users are running the same code “simultaneously.” How are they to be kept from overwriting each other’s data?
The first of these problems can be solved by partitioning main memory into several regions.

**Architecture II:** Partitions within main memory.

Each program runs within a *partition* of memory.

In the simplest case, there are just two partitions, one containing the OS, and the other containing a single user program.

Usually the OS is at the low end of memory, because that is where the interrupt vector is.

In a more general case, there can be more than one user program. Each program has associated with it a single *base-length* register pair.

The example below shows three programs occupying main memory.

![Diagram showing address spaces and mapping function for three processes.](image)

Typically, the contents of the base register are added to each logical address. *(Note: The base register is loaded by the operating system.)*
\[ Q \equiv \text{base register} \]
\[ R \equiv \text{length register} \]

Physical address = contents of \( Q \) + contents of index register
+ \( x \)

If the logical address is greater than the contents of the length register, it is an error, and the requested memory access is not performed.

\[
\text{if contents of index register} + x 
\geq \text{contents of } R \text{ then error.}
\]

Hence this architecture provides protection between processes.

From now on, we will “forget about” index registers in giving memory-mapping functions; they always take part in the computation of a logical address, but for simplicity, we omit the details.

Architecture II still has these limitations:

- It is still impossible to share programs or data.
- It is still impossible to protect one module of a program from another.
- Though many programs can fit in physical memory at once, no single program can be larger than physical memory.

**Swapping** (S&G, §8.3): In a multitasking system, several processes may be resident simultaneously in main memory.

- The processor “takes turns” working on each one.
- But some processes may be idle for a “long time” (if, e.g., the user walks away from his terminal).

Such processes “waste space” in memory that could better be used by other processes. So the solution is to swap out the idle processes to secondary storage.

Swapping of processes is very similar to swapping of segments in virtual memory, and we will treat it in more detail later.

But three special aspects of process swapping should be noted.
• Can a process be swapped out while waiting for input/output?

• Can swapping be done if the program has been relocated by a *loader* instead of by base-length registers?

• Suppose a process is only using *part* of the physical memory that is allocated to it. How can swapping be made more efficient?

Nowadays, process swapping is used mainly on PCs.

• In Microsoft Windows, when a new process is loaded and there is not enough memory, an old process is swapped to disk.
  The user decides when to bring it back.

• Windows/NT, however, supports paging.

Physical memory can be managed like a single large array, using any appropriate memory-management strategy (a few of which we will consider later).
Architecture III: Two segments per process. Same as Architecture II, except for a hack to allow sharing:

Each process is given two base-length registers, \((Q_0, R_0)\) and \((Q_1, R_1)\).

- The first register points to a non-sharable partition into which the program may write.
- The second register points to a sharable partition that contains code and (sometimes) read-only data.
- A bit in the address field of each operand can select which partition is to be used for each reference.

Still, programs can’t be larger than main memory.

Example: The DEC-10 kept code and read-only data in “high” memory— if the leading bit of an address was 1, the contents of \(Q_1\) (in the diagram above) were added to the logical address to find the physical address.

If the leading bit of an address was 0, it referred to non-sharable “low” memory, where read/write data is kept.

Example: The Univac 1108 had a similar scheme, but did not allow read-only data to be kept with code.

Instruction references referred automatically to memory in the sharable region; data references referred automatically to memory in the non-sharable region.

Thus the Univac 1108 did not need an extra bit in each address.

Architecture IV: Segmented virtual memory. To run programs which are larger than physical memory, there must be some way of arranging to have only part of a program in memory at one time.
Any operating system which allows a program to be larger than physical memory is called a virtual-memory system, because it allows a program to “use” primary memory which is not physically present.

To achieve virtual memory, we generalize Architecture III to allow more than one sharable and/or non-sharable segment.

- It is possible for a process to share portions of its code (e.g., I/O routines) and its data (e.g., databases) without sharing all code or all data.
- The segments can now be swapped independently; not all need to be in memory at once.
- Since swapping of segments is now independent of swapping of programs, the question of when to swap something or other becomes more complicated.
- With more entities being swapped, memory fragmentation becomes more serious.

Swapping rules:

- Read-only segments need not be copied out to secondary storage.
- Segments which have not been written into since being swapped in need not be swapped out either.
- Uninitialized segments need not be swapped in.

Addressing in segmented memory: A logical address is of the form \((s, d)\), where \(s\) is the segment number and \(d\) is the displacement.

\[
\begin{array}{c|c}
s & d \\
\end{array}
\]

The number of segments may be large; hence there are no “segment registers,” but rather a segment table.

The segment table contains segment descriptors, each of which hold—

- a base address (physical address of first word or byte),
- a length field (giving length in bytes or words), and
- an access-rights field (read, write, and perhaps execute).
Here is a diagram of addressing using a segment table.

Address translation: The naive method would be to consult the segment table each time a memory reference is to be performed.

Instead, some form of content-addressable, or associative memory is usually used.

- This memory is smaller than a segment table.
- The input is compared against all keys simultaneously. If the input matches a key, then the corresponding value is output.

Usually this memory is called a translation lookaside buffer, or TLB, because the processor “looks aside” to it as it is translating an address.

The next page shows a diagram of address translation using a TLB.
Each time the TLB is accessed, this procedure is followed:

- If an entry matching the segment number is found, the length is checked, and the base address returned.
- If no matching entry is found, a *victim* entry is chosen from among the TLB slots.

The victim is chosen to be an entry that is not very “active.” We will discuss what this means later.

A TLB’s *hit ratio* is the number of entries found / the number of times searched.

The TLB does not have to be very large to produce a good hit ratio. Early studies showed—

- Burroughs B6700—98% with 8 cells.
- Multics—98.75 with 16 cells.

However, this was 20 years ago, when programs were much smaller. Today’s TLBs typically have from 128 to 2048 entries.
Let $a$ be the time to search the TLB, $m$ be the time to access main memory, and $h$ be the hit ratio.

Then the effective access time is

$$h(a + m) + (1-h)(a + 2m) = a + (2-h)m$$

(This formula only applies in a one-level segment table.)

**Segment faults:** When a segment is referenced, it may not be in main memory. Each segment descriptor contains a *presence bit*, which tells whether or not the segment is “present.”

(For simplicity’s sake, the presence bit was not shown in the diagrams above.)

If a segment is not in main memory when it is referenced, it must be **swapped in**.

- This is called a *segment fault*.
- Usually, it will necessitate swapping some other segment(s) out.
- Thus the segment descriptor must also contain a *secondary-storage address*.

How many fields does a segment descriptor contain altogether?
Sharing of segments (similar to S&G, §8.5.5): If several processes are using the same code, it is advantageous for all processes to use just one copy.

In a time-sharing system, programs like editors are frequently shared.

S&G (p. 271) gives an example of three processes sharing a text editor. Memory is laid out as shown below.

Note that the processes share code, but each has its own data.

What access rights should each of the pages have?

However, sharing is harder than it looks! Shared references must map to the same virtual address in all processes, because—

- Code segments contain references to other instructions, e.g., CALL instructions. An instruction might be

  CALL 4, 125

  meaning call the procedure at segment 4, byte 125. This means that the procedure must be in segment
• Code segments also contain references to data, like

\[
\text{LOAD } R1, (2, 321)
\]

meaning load R1 from byte 321 of segment 2. Again, the data segment must be segment

The segment-table organization shown above won’t suffice for sharing in general. Why not?

There are several ways of solving this problem. For now, we will mention two.

• Give a process two separate segment tables,

  ° a user segment table, whose segments are not sharable, and where any segment can have any segment number.

  

<table>
<thead>
<tr>
<th>User segment table for ( P_1 ) (private)</th>
<th>User segment table for ( P_2 ) (private)</th>
<th>System segment table (shared)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>7000</td>
<td>3000</td>
</tr>
<tr>
<td>4000</td>
<td>6000</td>
<td></td>
</tr>
</tbody>
</table>

One bit in each address tells which segment table to use.

• Use indirection. Instead of containing segment numbers, the code just contains pointers to segment numbers. These pointers point to linkage sections, which contain the true segment numbers.

<table>
<thead>
<tr>
<th>Linkage sections for …</th>
<th>Segment table</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td>3000</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>4000</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>6000</td>
</tr>
<tr>
<td>1</td>
<td>7000</td>
</tr>
<tr>
<td>2</td>
<td>1000</td>
</tr>
<tr>
<td>3</td>
<td>2000</td>
</tr>
</tbody>
</table>

This is the approach taken by Multics dynamic linking, which we consider below.

Dynamic loading (S&G §8.1.2): With segmented memory, a process can begin execution before all the modules or procedures it may call are loaded.

• When a routine needs to call another routine, the calling routine checks to see whether the callee has been loaded.
• If not, the loader is called to load the desired routine into memory and update the program’s address tables to reflect this change.
• Then control is passed to the newly loaded routine.

**Dynamic linking** (S&G §8.1.3): If dynamic loading is used, linking must still take place in advance.

This means that copies of code libraries are linked in with the object code of different applications in advance.

With *dynamic linking*, however, incorporation of code libraries into the executable image is deferred until the program is loaded.

What is the advantage of this?

•

•

The dynamic-linking strategy described below was used on the Multics system in the late ’60s and ’70s. It is of renewed interest with the coming of the Java class loader.

Each module or procedure is found in some segment.

External references to code or data are handled differently from other systems.

External references are compiled as *indirect* references, just as in indirect linking.

But unlike in indirect linking, the address of the external reference is not stored in the external-reference list itself. (The reason for this is to allow sharing, as we will see later.)

To expedite dynamic linking, extra “baggage” is attached to each segment.
- The dictionary and external-reference list for each segment are placed *at the beginning of that segment*.

- Each segment also contains a *template* for linkage sections, which are described in greater detail below.

- In each entry in the external-reference list, the compiler places
  - the symbolic name* of the referenced symbol,
  - the symbolic name of the segment in which it is found, and
  - the *displacement* from the symbol that is to be referenced.

For example, in a reference to  
\[
\langle \text{seg}\rangle[[\text{mysym}]+5
\]
- *seg* is the name of the segment,
- *mysym* is the name of the symbol,
- and 5 is the displacement.

The reference is to five words past the symbol *mysym* in segment *seg*.

<table>
<thead>
<tr>
<th>Segment</th>
<th>Dictionary</th>
<th>Ext.-ref. list</th>
<th>Code and/or data</th>
<th>Template for linkage sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>External reference list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment name</td>
</tr>
<tr>
<td>seg</td>
</tr>
</tbody>
</table>

- There is a separate linkage section for each (process, code segment) combination.

---

*Actually, *pointers* to the symbolic names are used to save space.
[The linkage section is created by copying the linkage-section template when the process first uses the segment.]

Consider the linkage section for (process $p$, code segment $c$).

- The linkage section contains one entry corresponding to each entry in the external-reference list (the external-reference list of the code segment $b$ in which process $p$ is now running).

- External references are made indirectly through the linkage section (similar to indirect linking). (Multics has an indirect addressing mode.)
  - The code (in segment $b$) contains an index into the linkage section.
  - The linkage section contains the segment number of the referenced segment (say, segment $c$).
  - The segment table contains a descriptor for the segment (of course).

How are segments linked in dynamically?
• The first time an external reference is made to code segment \( c \), a fault occurs. This fault is similar to a segment fault, except that the dynamic linker is invoked instead of the segment-fault handler.

• The dynamic linker looks up the segment name in a file-system directory and fills in the segment-number field.

[Actually, the first time a process refers to a segment \( \text{seg} \), an entry for \( \text{seg} \) is made in the process’s Known Segment Table (KST). So if the process has ever used this segment before, the file system need not be consulted.]

<table>
<thead>
<tr>
<th>Known-Segment Table (KST)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment name</td>
</tr>
</tbody>
</table>

• The dynamic linker
  - looks up the symbol name in the dictionary of the newly linked segment and
  - calculates the displacement for the external reference (i.e., adds
    the displacement field from the external-reference table
    + the displacement given by the dictionary for this symbol).

  The resulting value is the \( d \) portion of the linkage-segment entry.

• The first reference to a segment probably causes a segment fault (if no other process was using that segment already).

Why are linkage sections necessary?
To allow sharing!
Suppose two processes running \textit{different} programs want to share a segment.
Suppose process $P_1$ uses modules—
- $A$, which has 5 segments, and
- $B$, which has 4 segments.

and process $P_2$ uses modules—
- $D$, which has 16 segments, and
- $B$.

Suppose also that $P_1$ uses $B$ before $P_2$ does. What problem will arise?

<table>
<thead>
<tr>
<th>Code within segment $B$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Segment table for $P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Segment table for $P_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

How do linkage sections solve this problem?

For efficiency’s sake, the linkage section can be bypassed by loading an external address into a processor register.

For example,
- if a process referred to an externally referenced segment which turned out to be segment 5, offset 17,
- it could load the (5, 17) into a processor register to access that segment without going through the linkage section each time.
Architecture V: Paging.

- Pages are “fixed-size” segments.
- They are of length $2^k$ (bytes, words, etc.)
- In main memory, a page occupies a fixed-size page frame.

Page tables are similar to segment tables, except that no length field is needed in a page descriptor.

TLBs work as with segmented memory, except that no length field is needed in their entries.

Note that physical addresses are formed by concatenation, rather than by addition.

Page size (S&G, §9.8.2): This is an architectural question that is affected by programming considerations.

- On average, a program half-fills its last page.

Thus, for a page size of $n$ words, an average of $n/2$ words will be wasted in the last page. This is called internal fragmentation. As page size grows larger, internal fragmentation grows more serious.
• With a large page size, large amounts of possibly unneeded information are paged in with the needed information.

Suppose a $20 \times 1000$ array is stored by rows:

$A[0, 0], A[0, 1], \ldots, A[0, 999],
A[1, 0], \ldots, A[1, 999], \ldots$

If a program performs calculations on the first fifteen columns, it will use 20 regions of fifteen words each, separated by 985 unused words.

◊ Assume page size is 2048 words. Since there are no gaps of $\geq 2048$ words, all 10 (or 11) pages comprising the array will be brought into main memory.

◊ If page size is 128 words, then even if each region of fifteen referenced words crossed a page boundary, a maximum of 2 pages/row, or 40 pages, totaling 5120 words, would be brought in.

However, often the number of words needed in main memory at one time depends on the organization of the array. See S&G, §9.8.4.

• As page size grows smaller, the page table grows larger. A larger and more expensive TLB is needed.
• Small pages make inefficient use of disks. It takes a long time for a disk to move its arm to the right cylinder, and then a much shorter time to transfer a page into main memory. We say that transfer time $\ll$ seek time.

• In addition, larger pages decrease the number of page faults, and thus decrease system overhead.

Page sizes in several computer systems:

<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Model</th>
<th>Page Size</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Honeywell</td>
<td>Multics</td>
<td>1024</td>
<td>36-bit words</td>
</tr>
<tr>
<td>IBM</td>
<td>360/67</td>
<td>1024</td>
<td>32-bit words</td>
</tr>
<tr>
<td>IBM</td>
<td>370/168</td>
<td>1024 or 512</td>
<td>32-bit words</td>
</tr>
<tr>
<td>DEC</td>
<td>PDP 10/PDP 20</td>
<td>512</td>
<td>36-bit words</td>
</tr>
<tr>
<td>DEC</td>
<td>Vax/11</td>
<td>512</td>
<td>8-bit bytes</td>
</tr>
<tr>
<td>Intel</td>
<td>80386</td>
<td>4096</td>
<td>8-bit bytes</td>
</tr>
<tr>
<td>Motorola</td>
<td>68030</td>
<td>256 to 32K</td>
<td>8-bit bytes</td>
</tr>
<tr>
<td>Intel</td>
<td>Pentium</td>
<td>4 K to 4M</td>
<td>8-bit bytes</td>
</tr>
</tbody>
</table>

In the 1970s, page size fell, due to reasons associated with locality of reference.

Since the 1980s, page size has been rising as increases in CPU speeds and memory capacity have been outpacing rises in disk speed.

**Paging vs. Segmentation**

• Advantages of paging.

1. No external fragmentation of main memory. (Funny-size holes don’t fit together well.)

2. No external fragmentation of secondary storage.

3. No bounds checking needed. This saves a small amount of time per reference. More importantly, the length information doesn’t need to be stored in the TLB, so a TLB entry can be smaller.
4. No need to bring in a *large* segment just to reference a few words.

• Advantages of segmentation.

1. No “internal fragmentation” of memory. In paged memory, a whole page must be allocated and transferred in even if only a few words are used.

2. Easier to handle data structures which are growing or shrinking (because maximum segment size is usually much larger than will “ever” be needed).

   ° In paged memory, if a process has many data structures that can grow or shrink (e.g., the process stack, another stack, a queue, etc.) the structures may “run into each other”:

   ° In segmented virtual memory, each data structure can be allocated its own segment, which can grow (or shrink) if necessary.

3. Facilitates sharing. A segment that is shared need not be any larger than necessary.

   In paged memory, by contrast, each shared item must occupy a set of pages, even if it uses just one word of the last page.

4. Array bounds can be checked automatically. What is the advantage of this?
Architecture VI: Multilevel page tables.

In recent years, the virtual-memory requirements of processes have been growing rapidly.

The Intel Pentium Pro, for example, has a maximum physical-memory size of $2^{36}$ and a maximum virtual-memory space of $2^{46}$.

Suppose that pages are $2^{10}$ bytes long, but the virtual address space is $2^{32}$ bytes. Suppose that a page-table entry occupies 8 bytes.

How large would the page table be?

Clearly, it is not possible to keep the entire page table in main memory.

We can instead arrange for the page table itself to be paged. Then a virtual address has the form

$$(p_1, p_2, d)$$

where $p_1$ is a page-table number, $p_2$ is the page number within that page table, and $d$ is the displacement.

A two-level page table is appropriate for a 32-bit address space.
If the outer page table (see the diagram above) would be larger than a page, a third level can be used. In practice, page tables of up to four levels have been used.

- The SPARC architecture supports a three-level page table and a 32-bit address space.
- The M68030 supports a four-level paging scheme.

**Architecture VII: Paging and segmentation combined.**

Architecture VI has assumed that all of the (lowest-level) page tables are a full page long. Suppose we relax this requirement to allow some of the tables to be shorter than a page.

Then we can call the set of pages pointed to by a lowest-level page table a “segment.”

- Each segment is divided into one or more pages.
- Avoids the storage-allocation problems posed by excessively large segments.

An address has the form \((s, p, d)\), where—

- \(s\) is the segment number,
- \(p\) is the number of a page within the segment, and
- \(d\) is the displacement within the page.

Each segment descriptor points to a page table.

Each page descriptor in the page table points to a page.

The TLB usually contains entries of the form

\(((s, p), f)\)

where \(f\) is a page-frame number.
• Thus, the TLB only needs to be searched once in case of a hit.
• In case of a miss, the segment table and then the page table must be consulted.

An example—OS/2 (S&G §8.7.2): OS/2 runs on Intel 386 and 486 architectures. The 386 uses paged segments:

• A process may have up to 16K segments.
• A segment may have up to $2^{32}$ bytes.

The virtual address space of a process is divided into two partitions:

• 8K segments that are private to the process. Information about this partition is kept in the Local Descriptor Table (LDT).
• 8K segments that are shared among all processes. Information about this partition is kept in the Global Descriptor Table (GDT).

Each entry in the LDT and GDT consists of 8 bytes. It contains, e.g., base and length of the segment.

A logical address is a (selector, offset) pair. A selector has this format:

<table>
<thead>
<tr>
<th>s</th>
<th>g</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

where—

• s designates the segment number,
• g indicates whether the segment is in the GDT or LDT, and
• p specifies protection.

How are segment descriptors accessed? The 80386 doesn’t use a conventional TLB to map segments.
Instead, an address in a program refers to one of six *segment registers*. These registers are loaded by the program. Among them are—

- **CS**—contains the segment selector of the *Code Segment*
- **DS**—contains the segment selector of the *Data Segment*
- **SS**—contains the segment selector of the *Stack Segment*
- **ES**—contains the segment selector of an *Extra Segment*

<table>
<thead>
<tr>
<th>“Segment selectors”</th>
<th>Access rights</th>
<th>Segment base address</th>
<th>Segment size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7 023</td>
<td>015 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Segment registers (loaded by program)</th>
<th>Segment-descriptor cache registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 0</td>
<td>47 4039 1615 0</td>
</tr>
</tbody>
</table>

There are also four *segment-descriptor cache registers*, which are loaded by the system (above).

A segment-descriptor cache register is loaded by this sequence of actions:

- The program places a segment-table index (a “selector”) in the appropriate segment register.
- The processor adds the selector value to the base address of the segment table, to select a segment descriptor.
- The processor copies the descriptor to the corresponding cache register.
- On accesses to the segment, the cache register, rather than the segment table, is consulted.

This scheme is not as fast as a TLB, because the cache registers must be reloaded every time one of the segments changes. (The code segment, for example, usually changes each time a procedure call occurs.)

Using the segment registers, the segment’s base and limit are used to generate a *linear address*. 
The limit is used to check for address validity.
If the address is not valid, an exception is generated.
If it is valid, the base and offset are added, resulting in a 32-bit linear address.

The linear address is then translated into a physical address.

How large is the page table for each segment? How do we determine that?

Thus, page tables need to be paged. A linear address is divided into—

- A page number consisting of 20 bits.
- A page offset consisting of 12 bits.

Here is a diagram of 80386 address translation.
The problem of shared references (S&G §19.6): How can processes share segments or pages? (R. S. Fabry, “Capability Addressing,” CACM, July ’74, pp. 403-412)

Consider a process that calls a procedure:

Outline for Lecture 18

I. Capability addressing
   - Sharing & protecting data
   - Uniform-address solution
   - Indirect-eval. solution
   - Mult. segment tables
   - Capability addressing

II. Protecting capabilities
   - Partitioned memory
   - Tagged memory
   - Fenced segments
   - Encryption
   - The mapping table

III. Arch. IX: Single virtual addr. space

IV. Protection rings
The process needs to call the procedure in segment 1, and read and write the data in segment 2.

Segment 2 contains *private data*, which must not be shared among different processes that execute the procedure.

Notice that the program counter (PC) points to somewhere in segment 0.

If two processes want to share the procedure, each must use a different set of data:

If process 1 and process 2 are running different *programs*, they will not necessarily use the same segment numbers for the procedure segment or the data segment.

For example, in the code at the right—

- Process 1 knows the procedure as segment _____.
- Process 2 knows the procedure as segment _____.

How can the code be made to work for both processes?

We will consider four different solutions.
Uniform-address solution: Each shared integer segment number must be interpreted in a functionally equivalent manner.

- Some segment numbers refer to private objects, and others to shared objects.
- Each shared segment number is dedicated to the same use in all processes that share it.
- Thus, functions of shared segment numbers must be defined centrally. Makes modularization difficult.
- This solution has been used by the Burroughs B6700 and the Medusa operating system for the Cm* multiprocessor.

Indirect-evaluation solution: Indirect linking (possibly dynamic linking) is used; processes make all references indirectly via linkage sections.

- When process 1 executes the code for FETCH 2, word 2 of the linkage section for process 1’s main program is fetched. It contains 0, so process 1’s segment 0 is used.
- A CALL “calls” the linkage section whose 0th entry points to the code.
- There is a linkage section for each separately compiled module used by a process.
• A “base” register points to the current linkage section.
• The contents of this register must be changed at the time of (inter-module) procedure calls. For example …
  ° When process 1 executes CALL 1, the entry in word 1 of process 1’s main program linkage section is fetched.
  ° It contains 4, which is then loaded into the linkage-section base register.
  ° Word 0 of the new linkage section is fetched. By convention, this register always points to the procedure code.
  ° It contains 2, which is then placed in PC.

• Although linkage-section indices are used for all ordinary addresses, if a segment is to be passed as a parameter, a segment number must be passed instead (since called procedure cannot consult linkage section of caller).

• Disadvantages:
  ° Extra space needed for linkage sections.
  ° More segments ⇒ increased swapping overhead.
  ° Takes time to initialize linkage sections.
  ° Extra indirection in address translation.

*Multiple segment-table solution:*
There is a different segment table for each procedure and each process.

• Problem: Difficult to pass segment references as parameters.
  (What are you going to use for a segment number?)
• Problem: Overhead of all those segment tables.
Architecture VIII: Capability-addressing solution.

We can do away with all those segment tables if we can, in effect, put segment descriptors in any ordinary data structure.

- Such "descriptors" are called capabilities. They are protected pointers to objects.
- Since they are protected from modification by users, they can be held in data structures or processor registers, or on the process stack.
- So it is easy to pass capabilities as parameters.

Means of protecting capabilities:

- Partitioned memory. There are two kinds of segments (and registers), capability segments and data segments.

Disadvantage: Overhead of extra

- Tagged memory. Each memory word contains a one-bit tag— "0" means data, "1" means capability.
This is an attractive solution for a word-addressable machine.

- Fenced segments. Each segment can hold both data and capabilities. It has a “fence” separating data in one portion from capabilities in the other.

- Encryption. When a capability is created by the kernel, it is encrypted by an algorithm that maps an unencrypted capability to a much longer encrypted version.

Before a capability is used, hardware (or microcode) decrypts it. If the result is not a legal capability, an exception is raised.

**Disadvantages:**

**Advantages:**
The mapping table: How does a capability point to an object?

Here is one method:

- All of the segment tables are replaced by a single systemwide object table, which contains object descriptors.

A capability contains a name field.

- This name field tells where in the object table to find the object descriptor (just like a segment number tells where to find a segment descriptor in a segment table).

This diagram shows an access to the data portion of an object.

An access to the capability portion is similar, except that—

- The base and fence are added to the displacement to yield the physical address.
- The displacement is compared with the length instead of the fence.
• If the name field is large (50 bits or so), it will never be necessary to re-use object names.

However, the object table will have to be multilevel, like the multilevel page table of Architecture VI.

• Sophisticated means are used to manage the object table (e.g., hashing of object names)

• A TLB can be used to hold recently accessed object-table entries. (Just like in paged or segmented memory, except that association is on the object name instead of the page or segment number.)

An architecture that uses capabilities to control access can be structured according to the object model—each process has capabilities only for the procedures and data structures it needs to access.

However, if objects are very small and numerous, the object table can get to be very large (maybe 1/4 or 1/3 of memory!).

But it is possible to implement objects without a central object table.

Architecture IX: A single large virtual address space.

With paged and segmented memory, each process has its own virtual address space. Suppose we decree that there will be only one virtual address space in the entire system?

• Virtual address space must be very large (e.g. $2^{64}$).

• Virtual address space must be allocated to processes (just like physical memory must).

• An object name is just its virtual address. (Of course, the object names must be protected by placing them in capability segments, tagging or encrypting them, etc.)
Two processes that share the object will use the same virtual address for it.

- Memory is paged; several objects can fit on a page, or an object can span several pages.

<table>
<thead>
<tr>
<th>Object A</th>
<th>Object B</th>
<th>Object C</th>
<th>Object D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page 1</td>
<td>Page 2</td>
<td>Page 3</td>
<td></td>
</tr>
</tbody>
</table>

- Now an object table is not needed, just a page table.
  - Objects can be much more numerous, and hence smaller.
  - A multilevel page table will be required, with about 4 levels.
  - In general, the pages used by a process may be scattered throughout the virtual address space.

  Hence, very few entries in each lowest-level page table may be in use.

  So, the page table is implemented as a hash table (since page numbers are sparse).

- The object length (and perhaps the object type) is stored at the beginning of each object instead of in the object table.

Capabilities are one way to provide protection in hardware. Here is another one—

**Protection Rings:** A generalization of supervisor/user modes. Instead of only two states, user and supervisor (monitor), there are $N$ rings, numbered 0 to $N-1$.

- Processes running in ring 0 are the most privileged.
- Segments as well as processes have ring numbers.
- A process running in ring $i$ may access segments belonging to rings $i$ to $N-1$ (providing it has a segment-table entry for them).
- A procedure may call code segments in more privileged rings only through well defined entry points known as gates.
A call from a process in ring \( j > i \) must be directed to a gate (i.e., a displacement in the set \( \{0, \ldots, k\} \)).

A call from a process in a ring whose number is \( \leq j \) may be directed anywhere within the code segment.

- If a more privileged procedure passes privileged data to a less-privileged procedure, it may only pass by value.

For example, if a parameter is in a segment in ring 3, and it is passed to a procedure in ring 4, it may not be passed by reference (because that would allow the procedure in ring 4 to write into ring 3).

**Advantages of ring structure** (compared to supervisor/user mode):

- The entire OS need not reside

- Users may

**Disadvantage of ring structure** (compared to capability-based protection):

Protection domains are

- Privilege of ring 0
- Privilege of ring 1
- \( \vdots \)
- Privilege of ring \( N-1 \)
Distributed shared memory: In Lecture 2, we saw that multiprocessors come in two varieties:

- **Shared-memory** multiprocessors can have more than one processor sharing a region of memory. Processes can therefore communicate via shared memory.

- **Distributed-memory** multiprocessors usually have to perform interprocess communication by passing messages from one processor to another.

Many techniques are known for programming shared-memory multiprocessors.

- Communication can be performed by simply writing to memory for the other processors to read.

- For synchronization, critical sections can be used with semaphores, monitors, etc. providing mutual exclusion.

A large body of literature is available on these techniques.

For distributed-memory multiprocessors, the situation is quite different.

- Communication must use message-passing, making I/O the central abstraction.

- Message-passing involves many complicating issues, e.g., flow control, lost messages, buffering, blocking.

Remote procedure call (RPC; see Lecture 3) can be used in many cases, but—

- it can’t be used to pass graphs and other data structures containing pointers;
- it doesn’t work when programs contain global variables, and
- it makes it expensive to pass large arrays, since they must be passed by value.
Furthermore, shared-memory machines can use message-passing primitives when appropriate, but the reverse is not true.

Thus, shared-memory multiprocessors are much easier to program.

Unfortunately, distributed-memory machines are much easier to build.

- Single-board computer modules, containing processor, memory, and network interface, can be connected together in almost unlimited numbers.
  Multicomputers with thousands of modules are available from various manufacturers.

- It is impossible to build a shared-memory multiprocessor with more than two or three dozen processors, because the shared bus becomes a bottleneck.

Thus, in the last 10 years, a lot of attention has focused on implementing a shared address space on a distributed-memory machine.

Here is a simple description of how distributed shared memory (DSM) works:

1. Each page (or segment) is present on exactly one machine.

2. A processor can refer to local pages using the ordinary hardware, at full memory speed.

3. An attempt to reference a page located on a different module causes a hardware page fault, which is handled by the operating system.

   The operating system sends a message to the remote machine, which finds the needed page and sends it to the requesting processor.

4. The faulting instruction is restarted, and can now complete.

What is the main difference between this and ordinary virtual memory?
What would appear to be the major shortcoming of this approach?

Fortunately, it is not necessary to share the entire virtual memory; indeed, it is not even a good idea. Why?

Instead, advanced DSM systems share only particular objects within the address space.

**NUMA multiprocessors:** Any process in a DSM system can access memory belonging to any processor, but not equally fast.

Access times are non-uniform, and so these machines are known as non-uniform memory access (NUMA) multiprocessors.

NUMA multiprocessors vary in how much hardware support is provided for DSM.

- The fastest NUMA multiprocessors, such as the SGI S2MP architecture and Origin multiprocessor, provide extensive hardware support for very fast reference to remote memory.
- However, it is also possible to implement DSM systems on networks of workstations (NOWs).

Regardless of the implementation, NUMA machines have these properties:

- It is possible for a processor to access memory anywhere in the system.
- It takes longer to access remote memory.

Thus, it is important to locate pages (or objects) close to the processes that are accessing them. How important depends on how fast remote access is.
When a page fault occurs, the OS has a choice.

- If the page is *read only*, it may—
  - *map* the local page to remote memory. This means that all references to the page must
  - *replicate* the page (make a copy on the local machine).

- If the page is *read-write*, it may—
  - *map* the local page to remote memory, or
  - *migrate* the page to the faulting processor.

If the page is migrated to the faulting processor, what else must be done?

For either RO pages or RW pages, there is a tradeoff. The OS must guess if the page will be heavily used.

Of course, the OS cannot foretell the future.

- To correct mistakes in page placement, and
- to adapt to changes in reference patterns,

NUMA systems usually have a daemon process, called the *page scanner*, running in the background.

Periodically, the page scanner gathers usage statistics about local and remote references.

(These statistics are maintained with help from the hardware, as we will discuss in Lecture 23 and 24.)

If usage statistics indicate that a page is in the wrong place, the page scanner unmaps the page so that the next reference causes a page fault.

Then a new placement decision can be made.
If a page moves too often within a short interval, the page scanner can mark it as *frozen*, which causes it to stay put for a certain interval of time or until conditions change.

Possible scanner strategies:

- Invalidate any page for which there have been more remote references than local references.
- Invalidate a page iff the remote reference count has been > local reference count the last $k$ times the scanner has run.

Studies have shown that no single algorithm works well for all programs.

*The Mach virtual memory*: Mach is a distributed version of Unix™, developed at Carnegie Mellon University.

In Mach, each process (called a *task*) is assigned a single paged address space.

A page in the process’s address space is either allocated or unallocated.

- An unallocated page cannot be addressed by the threads of a task. An allocated page can.
- An allocated page does not necessarily occupy main memory

In Mach, memory is allocated and deallocated in terms of regions.

An address is valid only if it falls into an allocated region.

Most memory-management hardware today supports an address space of at least 4 GB.

Some applications benefit by using a large address space sparsely. E.g.—

- an application that maps several large files into its address space, and
- then ceases to use them.
Sharing memory: Mach provides several primitives for sharing memory within a task.

But sometimes it is necessary to share memory with other tasks.

For example, a debugger needs to examine and modify the address space of a task being debugged. Mach provides primitives to perform these operations.

In Mach,

- All threads of a task automatically share all the memory objects that reside in the task’s address space.
- Different tasks can share a page (or a memory object) by installing it into their virtual address spaces. This requires their page-table entries to cause addresses within the page to be translated to the same page frame.

In Mach, all sharing is by inheritance.

When a task is created, a new address space is created for it. The address space can either—

- be empty, or
- be based on an existing address space.

When a new address space is based on an existing address space, a page in the new address space is initialized based on the inheritance value of the corresponding page in the existing address space.

An inheritance value of—

- none means the child task does not inherit that page,
- copy means the child task receives a copy of the page, which it can manipulate without affecting the original page, and
- share means the same copy of the page is shared between child and parent.
Mach uses the principle of *lazy evaluation* to avoid unneeded work.

- Virtual-to-physical mapping of a page is postponed till the page is referenced.
- Page tables are not allocated until they are needed.
- When a page needs to be copied, it is not copied until it is actually written.

This kind of copying is called *copy-on-write*. How is it implemented?

When $T_1$ and $T_2$ want to share a page with copy inheritance, the system gives each of them *read-only* access to the page. Then when $T_1$ writes into the page, it gets a new copy.

Advantages of copy-on-write:

- 
- 

The Unix *fork* operation, for example, is implemented using copy-on-write.

Pages that are never modified are thus never copied.

*Address mapping in Mach*: Mach distinguishes machine-independent and machine-dependent aspects of address mapping.

- Page-table organization is machine dependent.
  - The VAX supports separate system and user page tables.
  - The Intel x86 architectures support multilevel page tables.
  - The RS/6000 uses an inverted page table (explained later).

- Other aspects of address mapping are not machine dependent.
For example, all architectures divide a process’s address space into code, data, and stack areas.

Also, in Mach, files are “mapped into” the address space. (This means they are assigned a set of addresses within the address space.)

An address map in Mach is a structure that tells what memory objects each part of the address space is associated with.

For example, a typical VAX Unix process has four entries in its address map when it is created:

- Code
- Stack
- Initialized data
- Uninitialized data

It indicates, for example, that the code occupies virtual addresses 0–150K.

A typical address map is shown at the right, above.

Each entry points to a memory object (the triangles in the diagram).

But more than one entry may point to a single object, as when different parts of the same object have different protection or copy attributes.

This is illustrated by the diagram of an address-map entry:
Address maps—
   • allow efficient implementation of the most common operations on the address space of a task, e.g.,
     ° page-fault lookup, as when a file is accessed for the first time, or
     ° copy/protection operations on a memory region.
   • efficiently maintain sparse address spaces.

When a new address space is created, and an object has the copy inheritance value, a new memory object is created for it.

This new object is called a shadow object.

A shadow object need only contain those pages that have been copied. (For other pages, the original memory object is used.)

A shadow object may itself be shadowed as the result of a subsequent copy-on-write copy, creating a shadow chain.

This structure cannot, however, maintain shared memory. A change to one copy of a shared object needs to cause all other copies to change.

This requires an extra level of indirection when accessing a shared object. The address-map entry points to a share map, which in turn points to the memory object.
Memory consistency: In a distributed memory system, references to memory in remote processors do not take place immediately.

This raises a potential problem. Suppose that—

- The value of a particular memory word in processor 2’s local memory is 0.
- Then processor 1 writes the value 1 to that word of memory. Note that this is a remote write.
- Processor 2 then reads the word. But, being local, the read occurs quickly, and the value 0 is returned.

What’s wrong with this?

This situation can be diagrammed like this (the horizontal axis represents time):

\[
P_1: \quad W(x) \quad 1 \\
P_2: \quad R(x) \quad 0
\]

Depending upon how the program is written, it may or may not be able to tolerate a situation like this.

But, in any case, the programmer must understand what can happen when memory is accessed in a DSM system.

Consistency models: A consistency model is essentially a contract between the software and the memory.

It says that if the software agrees to obey certain rules, the memory promises to store and retrieve the expected values.

Strict consistency: The most obvious consistency model is strict consistency.

**Strict consistency:** Any read to a memory location x returns the value stored by the most recent write operation to x.
This definition implicitly assumes the existence of a global clock, so that the determination of “most recent” is unambiguous.

Strict consistency is supported by uniprocessors, but it is generally impossible to implement in DSM systems.

In a DSM system, a read from a nonlocal memory location, or a write to a nonlocal memory location, requires sending a message.

This message requires a finite time to travel to and from the node where the memory is located. During that time, some other processor might change the value in the memory location.

Regardless of how efficient a message system is, it cannot cause messages to travel faster than the speed of light.

If the difference in the two access times is, say, one ns., and the nodes are 3 m. apart, the signal would have to travel at 10 times the speed of light to return the most recent value.

Fortunately, strict consistency is rarely necessary.

To summarize, with strict consistency,

- all writes are instantaneously visible to all processes;
- all subsequent reads see the new value of the write, no matter how soon after the write the read is done, and no matter where the process executing the read is located.

**Sequential consistency**: Strict consistency isn’t really necessary to write parallel programs.

Earlier in this course, we learned that parallel programs shouldn’t make any assumptions about the relative speeds of the processes, or how their actions would interleave in time.

Counting on two events within one process to happen so quickly that another process won’t have time to do something in between is asking for trouble.

Let us weaken the requirements we impose on the memory so that the resulting model is realizable.
Sequential consistency: The result of any execution is the same as if

- the memory operations of all processors were executed in some sequential order, and
- the operations of each individual processor appear in this sequence in the order specified by its program.

What’s the difference between strict consistency and this?

- In sequential consistency, the temporal ordering of events does not matter.
- All that is required is that the processors see the same ordering of memory operations (regardless of whether this is the order that the operations actually occurred).

So, with sequential consistency we don’t have to worry about providing “up to the nanosecond” results to all of the processors.

The example below shows the difference between strict and sequential consistency. The two sequences of operations are equally valid.

Note that a read from P₂ is allowed to return an out-of-date value (because it has not yet “seen” the previous write).

```
P₁: W(x)₁
P₂: R(x)₀ R(x)₁
P₁: W(x)₁
P₂: R(x)₁ R(x)₁
```

From this we can see that running the same program twice in a row in a system with sequential consistency may not give the same results.

While it is possible to implement, sequential consistency has very poor performance. Can you guess why?
Causal consistency: The next step in weakening the consistency constraints is to distinguish between events that are potentially causally connected and those that are not.

Two events are causally related if one can influence the other.

\[
P_1: \ W(x)1 \\
\frac{}{P_2: \ R(x)1 \ W(y)2}
\]

Here, the write to \(x\) could influence the write to \(y\), because

On the other hand, without the intervening read, the two writes would not have been causally connected:

\[
P_1: \ W(x)1 \\
\frac{}{P_2: \ W(y)2}
\]

The following pairs of operations are potentially causally related:

- A read followed by a later write.
- A write followed by a later read to the same location.
- The transitive closure of the above two types of pairs of operations.

Operations that are not causally related are said to be **concurrent**.

**Causal consistency:** *Writes that are potentially causally related must be seen in the same order by all processors.*

**Concurrent writes may be seen in a different order by different processors.**

Here is a sequence of events that is allowed with a causally consistent memory, but disallowed by a sequentially consistent memory:
P1: \( W(x)1 \quad W(x)3 \)

\[ \begin{array}{c}
P_2: \quad R(x)1 \quad W(x)2 \\
P_3: \quad R(x)1 \quad R(x)3 \quad R(x)2 \\
P_4: \quad R(x)1 \quad R(x)2 \quad R(x)3 \\
\end{array} \]

Why is this not allowed by sequential consistency?

Why is this allowed by causal consistency?

What is the violation of causal consistency in the sequence below?

\[ \begin{array}{c}
P_1: \quad W(x)1 \\
P_2: \quad R(x)1 \quad W(x)2 \\
P_3: \quad R(x)2 \quad R(x)1 \\
P_4: \quad R(x)1 \quad R(x)2 \\
\end{array} \]

Without the \( R(x)1 \) by \( P_2 \), this sequence would’ve been legal.

Implementing causal consistency requires the construction of a dependency graph, showing which operations depend on which other operations.

**PRAM consistency:** Causal consistency requires that all processes see causally related writes from all processors in the same order.

The next step is to relax this requirement, to require only that writes from the same processor be seen in order. This gives pipelined-RAM (PRAM) consistency.

**PRAM consistency:** \textit{W}rites performed by a single process are received by all other processors in the order in which they were issued.
Writes from different processors may be seen in a different order by different processors.

PRAM consistency is so named because writes can be pipelined; that is, a processor does not have to stall waiting for a write to be completed before starting the next one.

PRAM consistency would permit this sequence that we saw violated causal consistency:

\[
\begin{array}{c}
P_1: \ W(x)1 \\
P_2: \ R(x)1 \ W(x)2 \\
P_3: \ R(x)2 \ R(x)1 \\
P_4: \ R(x)1 \ R(x)2 \\
\end{array}
\]

Another way of looking at this model is that all writes generated by different processors are considered to be concurrent.

Sometimes PRAM consistency can lead to counterintuitive results.

\[
\begin{align*}
P_1: & \quad a := 0; \\
& \quad a := 1; \\
& \quad \text{if } b = 0 \text{ then } \text{kill}(p_2); \\
P_2: & \quad b := 0; \\
& \quad b := 1; \\
& \quad \text{if } a = 0 \text{ then } \text{kill}(p_1);
\end{align*}
\]

At first glance, it seems that no more than one process should be killed. With PRAM consistency, however, it is possible for both to be killed.

Processor consistency: Processor consistency is very similar to PRAM consistency, but it has one additional condition: memory coherence.

Memory coherence requires that writes to the same location be viewed in the same order by all the processors.

 Writes to different locations need not be seen in the same order by different processors.
**Processor consistency** = *PRAM consistency + memory coherence.*

*Weak consistency*: PRAM and processor consistency are still stronger than necessary for many programs, because they require that writes originating in a single processor be seen in order everywhere.

But it is not always necessary for other processors to see writes in order—or even to see all writes, for that matter.

Suppose a processor is in a tight loop in a critical section, reading and writing variables.

Other processes aren’t supposed to touch these variables until the process exits its critical section.

Under PRAM consistency, the memory has no way of knowing that other processes don’t care about these writes, so it has to propagate all writes to all other processors in the normal way.

To relax our consistency model further, we have to divide memory operations into two classes and treat them differently.

- Accesses to *synchronization variables* are sequentially consistent.
- Accesses to other memory locations can be treated as concurrent.

This strategy is known as *weak consistency*.

With weak consistency, we don’t need to propagate accesses that occur during a critical section.

We can just wait until the process exits its critical section, and then—

- make sure that the results are propagated throughout the system, and
- stop other actions from taking place until this has happened.

Similarly, when we want to enter a critical section, we need to make sure that all previous writes have finished.
These constraints yield the following definition:

**Weak consistency:** A memory system exhibits weak consistency iff—

1. **Accesses to synchronization variables are sequentially consistent.**

2. **No access to a synchronization variable can be performed until all previous writes have completed everywhere.**

3. **No data access (read or write) can be performed until all previous accesses to synchronization variables have been performed.**

Thus, by doing a synchronization before reading shared data, a process can be assured of getting the most recent values.

Note that this model does not allow more than one critical section to execute at a time, even if the critical sections involve disjoint sets of variables.

This model puts a greater burden on the programmer, who must decide which variables are synchronization variables.

Weak consistency says that memory does not have to be kept up to date between synchronization operations.

This is similar to how a compiler can put variables in registers for efficiency’s sake. Memory is only up to date when these variables are written back.

If there were any possibility that another process would want to read these variables, they couldn’t be kept in registers.

This shows that processes can live with out-of-date values, provided that they know when to access them and when not to.

The following is a legal sequence under weak consistency. Can you explain why?

\[
\begin{array}{c}
\text{P}_1: & W(x)1 & W(x)2 & S \\
\text{P}_2: & & R(x)2 & R(x)1 & S \\
\text{P}_3: & & R(x)1 & R(x)2 & S \\
\end{array}
\]
Here’s a sequence that’s illegal under weak consistency. Why?

\[ P_1: \ W(x)1 \ W(x)2 \ S \]
\[ P_2: \ S \ R(x)1 \]

**Release consistency:** Weak consistency does not distinguish between entry to critical section and exit from it.

Thus, on both occasions, it has to take the actions appropriate to both:

- making sure that all locally initiated writes have been propagated to all other memories, and
- making sure that the local processor has seen all previous writes anywhere in the system.

If the memory could tell the difference between entry and exit of a critical section, it would only need to satisfy one of these conditions.

Release consistency provides two operations:

- *acquire* operations tell the memory system that a critical section is about to be entered.
- *release* operations say a c. s. has just been exited.

It is possible to acquire or release a single synchronization variable, so more than one c.s. can be in progress at a time.

When an acquire occurs, the memory will make sure that all the local copies of shared variables are brought up to date.

When a release is done, the shared variables that have been changed are propagated out to the other processors.
But—
  • doing an acquire does not guarantee that locally made changes will be propagated out immediately.
  • doing a release does not necessarily import changes from other processors.

Here is an example of a valid event sequence for release consistency (A stands for “acquire,” and Q for “release” or “quit”):

\[
\begin{align*}
P_1: & \quad A(L) \ W(x)_1 \ W(x)_2 \ Q(L) \\
P_2: & \quad A(L) \ R(x)_2 \ Q(L) \\
P_3: & \quad R(x)_1
\end{align*}
\]

Note that since \( P_3 \) has not done a synchronize, it does not necessarily get the new value of \( x \).

**Release consistency:** A system is release consistent if it obeys these rules:

1. **Before an ordinary access to a shared variable is performed, all previous acquires done by the process must have completed.**

2. **Before a release is allowed to be performed, all previous reads and writes done by the process must have completed.**

3. **The acquire and release accesses must be processor consistent.**

If these conditions are met, and processes use acquire and release properly, the results of an execution will be the same as on a sequentially consistent memory.

**Summary:** Strict consistency is impossible. Sequential consistency is possible, but costly. The model can be relaxed in various ways.
Consistency models not using synchronization operations:

<table>
<thead>
<tr>
<th>Type of consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strict</td>
<td>All processes see absolute time ordering of all shared accesses.</td>
</tr>
<tr>
<td>Sequential</td>
<td>All processes see all shared accesses in same order.</td>
</tr>
<tr>
<td>Causal</td>
<td>All processes see all causally related shared accesses in the same order.</td>
</tr>
<tr>
<td>Processor</td>
<td>PRAM consistency + memory coherence</td>
</tr>
<tr>
<td>PRAM</td>
<td>All processes see writes from each processor in the order they were initiated. Writes from different processors may not be seen in the same order.</td>
</tr>
</tbody>
</table>

Consistency models using synchronization operations:

<table>
<thead>
<tr>
<th>Type of consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak</td>
<td>Shared data can only be counted on to be consistent after a synchronization is done.</td>
</tr>
<tr>
<td>Release</td>
<td>Shared data are made consistent when a critical region is exited.</td>
</tr>
</tbody>
</table>

The following diagram contrasts various forms of consistency.
In summary,

- In sequential consistency, R and W follow program order.
- In PRAM OR processor consistency, R may precede buffered W; other processors may see different order of \{W, R\} access.
- In weak consistency, SYNCH operations are sequentially consistent; other memory operations can occur in any order.
- In release consistency, SYNCH operations are split into ACQUIRE (lock) and RELEASE (unlock), and these operations are processor consistent.

Page-based distributed shared memory:  
So far, we have not differentiated between DSM systems built on multiprocessors and on networks.

In both types of systems, memory consistency is needed, and we can use inheritance to decide whether to share or copy pages when a new process is created.

However, there are differences, due to

- latency of interprocessor communication, and
- whether address-mapping hardware is available to translate remote references.

Thus, we will differentiate between

- NUMA systems, which have hardware for accessing remote memory, and
- NORMA (no remote memory access) systems, where remote memory accesses must be performed totally in software.

When a NUMA system references a remote page, it may—
- fetch the page, or
- use it remotely, with the MMU mapping each access.
On a NORMA system, the MMU can’t address remote memory, so the page must be fetched.

Small (bus-based) multiprocessors provide sequential consistency, and early NORMA DSM systems tried to replicate this, so that programs would not have to be rewritten.

But it was soon realized that major performance gains could be achieved by relaxing the consistency model.

Thus, programs had to be rewritten.

Replication of memory: In a DSM system, when a processor references an address that is not local,

- a trap occurs, and
- the DSM software fetches the chunk of memory containing the address, and
- restarts the faulting instruction, which can now complete successfully.

For example, in the diagram below, if processor 0 references instructions or data in chunks 0 or 2, the references are done locally (chunk may or may not = page).

References to other chunks cause

For example, a reference to chunk 10 will cause a trap to the DSM software, which then moves chunk ______ from machine ___ to machine ___.

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

```
0 2

CPU 0
```

```
1 3 4

8 11

CPU 1
```

```
5 7 9

12 14

CPU 2
```

```
6 10 13

15

CPU 3
```
However, suppose that page 10 contains code. After the move, if processor 3 is running this program, the first reference to page 10 will cause a fault. Page 10 will ping-pong back and forth.

Hence, it is a good idea to allow copies of pages to be made, as in Mach:

Another possibility is to replicate all chunks. As long as a chunk is being shared, multiple copies can exist.

When a shared chunk is first written, special action must be taken to prevent inconsistency.

**Granularity of chunks:** So far, we have not specified how large a chunk is.

On a multiprocessor, it may be quite small. The MMU of the requesting processor knows what word is being fetched, and can immediately request it over the interconnection network within microseconds or nanoseconds.

So usually the unit of transfer is a cache line ($\approx 32–128$ bytes).

But in NORMA systems, the overhead of transfer is much greater.

This argues for a _______ chunk size, so that transfers don’t need to be made so _______.
One possibility is to transfer a page at a time.

This makes it simpler to integrate DSM with virtual memory, since the unit of transfer (page) is the same for both.

But it may be even better to transfer a larger chunk, say 2, 4, or 8 pages.

- Advantage:
  - Disadvantages:

A large chunk size exacerbates the problem of false sharing.

Suppose that—

- processor 1 is making heavy use of a variable $a$.
- processor 2 is making heavy use of a variable $b$.

Variables $a$ and $b$ are not on the same page, but they are in the same 2-page chunk.

What happens?

- If 1 chunk = 1 page,
- If 1 chunk > 1 page,

Clever compilers that understand which variables are shared can place variables in different chunks and help avoid false sharing.
However, this is not always possible, especially if $a$ and $b$ are array elements in close proximity.

**Achieving sequential consistency:** Whenever pages are copied, e.g., for performance reasons, the copies must be kept consistent.

If the pages are read-only, no special action need be taken. An ordinary entry is made in the page table of each process using the page, and the MMU is used as normal to perform address translation.

If the copies are writable, however, the first write to the page must be trapped (treated like a page fault).

The operating system then gains control. It has two choices.

- **Update** the other copies of the page. This involves sending the address of the modified word and its new value to all other processors that might have copies.
  The other processors then update their local copy of the page.

- **Invalidate** the other copies. This involves sending the address (only) to all of the other processors.
  The other processors mark the page as “not present,” and on their next reference to that page, a page fault occurs.
  Hence, only one copy of the page remains.

NUMA multiprocessors can use either of these two strategies (usually on cache lines, rather than whole pages; this is covered in CSC/ECE 506).

However, in a NORMA DSM system, one of these strategies is not feasible. Can you guess which?

The problem is that in a NORMA system, without special hardware, the only way to mark a reference for special handling (e.g., invalidation or update) is to mark the page as “read-only” or “not present”

Then, when the page is written, a trap will invoke the operating system, which can take care of notifying the other processors.
With a protocol based on invalidation, the OS needs to be invoked only the first time the page is written. There is some overhead to this, but in general, it is feasible.

With a protocol based on __________, the OS needs to be invoked each time the page is written. This causes a context swap at every write reference, and is obviously prohibitive.

Besides, there is a consistency problem. Do you see what it is?

An invalidation protocol for page-based DSM: At any time, each page is either in state

- $R$ (readable), or
- $W$ (readable and writable).

The state may change as execution progresses.

Each page has an owner, which is the process that most recently wrote the page.

- If a page is in state $R$, the owner has a copy
- If a page is in state $W$, only one copy exists (the owner’s).

There are twelve possible transitions, depending on whether—

- A process, say $p_0$, reads or writes.
- $p_0$’s processor is the owner of the page.
- The state of the page is—
  - $R$, with a single outstanding copy,
  - $W$, with a single outstanding copy, or
  - $R$, with multiple copies.
The simplest case is when $p_0$ is running on the owning processor, and the page is held with write access. In this case, either a read or a write can proceed immediately.

If $p_0$ is running on the owning processor and the page is held with read access, a read proceeds immediately, but a write requires a change of state.

If $p_0$ is running on the owning processor and there are multiple read copies, a read proceeds immediately, but a write requires that the other copies

If $p_0$ is running on a non-owning processor, a read still proceeds immediately, but a write requires that the requesting processor gain ownership.
Action when \( \text{process } p_0 \) reads

1. Ask for
2. Mark page as \( R \)
3. Do read

Action when \( \text{process } p_0 \) writes

1. Ask for invalidation
2. Ask for ownership
3. Ask for page
4. Mark page as \( W \)
5. Do write

With the situation the same as above, but no local copy,

- a write proceeds as before, except that the page needs to be ____,
- a read requires the page to be _____ first.

With the situation the same as before, but a non-local page in state \( W \),

- a write proceeds as in the previous case.
- a read requires the access mode ____ ________ to be **degraded** to \( R \).

Note that the other processor can still keep a copy of the page.

Which of these cases require traps to the operating system?

Note that in all cases, before a write is performed, the protocol guarantees that only a single copy of the page exists.

Thus, _______ is maintained.
Finding the owner: The replication protocol requires that the owner be found whenever

- 
- or
- 

It is not practical to broadcast these requests to every processor. Why?

For this reason, one process may be designated as the page manager.

The page manager keeps track of which processor owns each page.

When a process $p_0$ wants to contact the owner of a page,

- it sends a message to the page manager asking

- The page manager replies.
- $p_0$ sends a message to the owner, which performs the action and confirms.

Four messages are required, as shown below.

An obvious optimization is to have the page manager forward the request to the owning processor. This reduces the number of messages to three.
Still, the performance of this scheme could be bad. Why?

The solution is to provide

This introduces a problem—finding the right page manager.

Suppose we use a few bits of the page number to identify the page manager—the three least-significant bits, for example.

Then

- page manager 0 would handle all pages whose number ends with 000,
- page manager 1 would handle all pages whose number ends with 001, etc.

Why use the least-significant bits, rather than, e.g., the most-significant bits?

Finding copies of pages: How can copies be found when they need to be invalidated?

The first way is to broadcast a message and rely upon all of the processors to look at it and invalidate the page if they hold a copy.

However, this is not only slow, but unreliable if a message can be lost.
A better way is to have each page manager keep a list of which processors have copies of each page it manages.

When a page must be invalidated, the old owner, new owner, or page manager sends a message to each processor holding the page, and waits for an ACK.

When the message has been acknowledged, the invalidation is complete.

Synchronization: Recall from Lecture 5 that a test_and_set instruction.

It returns the value of its argument, and sets the variable specified by the argument to true, in a single atomic operation.

In a DSM system, the code works fine as long as only one process at a time is competing for entry to the critical section.

If \( p_0 \) is inside the critical section, and \( p_1 \) (on a different machine) wants to enter,

- \( p_1 \)
- The page containing the variable remains on \( p_1 \)'s machine.
- When \( p_0 \) exits the critical section, it pulls over this page and sets the variable to false.
- \( p_1 \) immediately encounters a page fault and pulls the page back.

So, the exit and entry have been performed with just two page faults.

However, the performance can be much worse than this. How?
Not only is this slow for the processors, but it creates enormous network traffic.

Therefore, a DSM system often provides an additional mechanism for synchronization.

A synchronization manager(s) can accept messages asking to enter and leave critical sections (or set and reset locks), and send a reply back when the requested action has been accomplished.

If the critical section can't be entered immediately, a reply is deferred, causing the requester to block. When the critical section becomes available, a message is sent back.

This accomplishes synchronization with a minimum of network traffic. We will take this up in more detail when we consider distributed process management in Lecture 25.

Managing the physical address space:
So far, we've been studying the virtual address space—what's in it, how it's addressed, when it's moved.

Now we want to concentrate more on physical memory. Techniques depend upon the architecture.

- **Paged virtual memory**—main concern is when to bring a page of data into main memory, and which page to remove.
- **Segmented virtual memory**, or
- **Multiprogrammed non-VM**—management of free memory is also a major concern.

Management of available physical memory: We will study two methods: a *chained pool* of free storage, and the *buddy system*.

With either method, managing free storage consists of three tasks:

1. Finding a hole large enough for a segment of a given size.
2. Updating the list of holes when a segment is allocated.
3. Updating the list of holes when a segment is deallocated.

Outline for Lecture 22

I. Management of available storage
   A. Chained pool
   B. Buddy system
   C. The 50% rule

II. Swapping & paging
   A. Handling a page fault
   B. The page table
   C. The frame table
   D. The TLB
   E. Locked-in pages

III. Inverted page tables
**Chained Pool:** An available-space list (list of holes) is used. Has these characteristics:

- Ordered by address. Aids in coalescing holes when a segment is deallocated.

- Consists of *nodes*, each of which begins with a header composed of
  - a size field and
  - a pointer to the next node on the list.

- A “dummy” node of length 0 resides permanently at the beginning of the list.

- For allocated segments, we will assume that the size of the segment is obtained from the segment table, not from a field within the segment. (Bad for robustness.)

When memory is to be allocated, the list may be searched by one of several methods:

1. First fit. Allocate the first hole on the list that is big enough.

2. Best fit. Allocate the hole which most nearly matches the requested size.

3. Modified best fit. Best fit, except that either—
   - small leftover holes of size $\leq e$ are included in the allocated block, or
   - all blocks allocated are required to be multiples of a certain size $e$.

4. Next fit—first fit that “remembers” where it left off.

The search algorithm for next fit—

- Needs to delete a hole if its size is exactly the size of the allocated segment.
Exact-size hole:

Before

After

Non-exact size hole:

Before

After

- Keeps a variable trail which points to where the search left off last time.

The space-allocation algorithm—

- If the hole is the same size as the segment, the hole is deleted.
- Otherwise, space is taken from the end of the hole so that links don’t have to be changed.

The space-deallocation algorithm—

- Note that freed space might have to be merged with the holes preceding and/or following it.

Note that this method requires searching the free list on both allocation and deallocation.

Another method, using boundary tags, avoids the need to search at deallocation by using

- a doubly linked but unordered available-space list, and
• in both the first and last word of each block, size fields and boolean boundary tags, which tell whether the block is allocated or not.

**Allocated**

+ Size | Size +

**Unallocated**

– Size | Link | Link –

**Compaction** (S&G, §8.4.3): After the system executes for awhile using first-fit, next-fit, or a related algorithm, memory is likely to be fragmented, as shown on the right (S&G, Fig. 8.11).

Now if a request for 500K is received, it cannot be honored, despite the fact that a total of 900K is free.

One solution is to move all of the allocated blocks to one end of memory, as shown below, left.

How many words need to be moved in this case?

A more intelligent approach would move fewer words to achieve the same result, as shown above, middle. How many words are moved here?
If we are willing to allow the hole to be somewhere other than one end of memory, we may reduce the copying requirement still further. How many words need to be moved in the diagram on the right?

Storage compaction is expensive.

- It should be used only when no hole is large enough to satisfy a request.
- It is usually cheaper to swap segments.

Storage compaction is sometimes impossible. Why? Hint: Consider address relocation.

---

The **buddy system**: An alternative to the chained-pool methods.

- Block sizes are restricted to powers of two.
- A separate free list is used for each size of block.
- Each address that ends in \( k \) zeros is considered to be the starting address of a block of length \( 2^k \).
- If a block of size \( 2^k \) is requested and none is available, list \( k + 1 \) is checked for a block of size \( 2^{k+1} \).
  - If one is found, it is split into two **buddies**, each of size \( 2^k \), and one of the buddies is allocated, and the other is placed on the size-\( 2^k \) free list.
  - If none is found, the next larger size block (size \( 2^{k+2} \)) is sought, and so on.
- When a block of size \( 2^k \) is deallocated, a test is made to see whether its buddy is free.
  - If so, the two blocks are recombined into one of the next larger size (\( 2^{k+1} \)), which in turn is recombined with its buddy, if available, and so forth.
• The splitting method guarantees that the starting addresses of two buddy blocks of size $2^k$ differ only in the $(k+1)$st digit from the right.

<table>
<thead>
<tr>
<th>Addr(B)</th>
<th>1 0 1 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr(Buddy(B))</td>
<td>1 0 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Let Addr(B) represent the address of a block of size $2^k$. Then what is the address of B’s buddy?

The fifty-percent rule: Regardless of what storage-allocation algorithm is being used, at equilibrium, there will be half as many holes as allocated blocks.

Consider an allocated block in memory.

• Half of the operations on the block above it will be allocations, and half will be deallocations.
• Thus, half the time it has another allocated block as an upper neighbor, and half the time it has a hole for an upper neighbor.

Therefore, if the mean number of allocated blocks is $n$, the mean number of holes is $n/2$.

Swapping and paging: Information may be transferred into main memory according to one of two policies.

• Demand paging (or swapping). A page is not swapped in until it is referenced by a process. A page fault then occurs, and the process must wait.
• Prepaging (pre-swapping). The program requests transfers of pages some time before it needs to use them. Useful in closed (e.g. some real-time) systems.

A page fault may occur while—

• fetching an instruction, or
• fetching or storing data (an operand of an instruction).
Steps in handling a page fault:

1. On a TLB miss, check the page-table entry for the referenced page. If this is a page fault, the presence bit will be off.

2. Trap to the operating system.

3. Select a free page frame (or a “victim,” if all frames are occupied).
   Write out the victim if it has changed.
   Determine the location of the desired page on secondary storage.
   Block the faulting process.

4. Issue a read request from secondary storage to the free frame.

5. When the read operation is complete, modify the page table and frame table (described below).

6. Restart the instruction that caused the page fault. (The instruction must be re-fetched, then executed.)

The page table contains this information in each entry:

<table>
<thead>
<tr>
<th>Presence bit</th>
<th>Valid bit</th>
<th>Access rights</th>
<th>Page-frame number</th>
<th>Secondary-storage address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The presence bit tells whether the page is in main memory.
- The valid bit tells whether this page exists. If the referenced address is too high, an exception will be generated.
- The access rights tell what kinds of access the process has to this page.
- The page-frame number and secondary-storage address can be stored in the same field, as explained below.
The *frame table* is a companion to the page tables.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Page-descriptor address</th>
<th>Secondary-storage address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- It tells how to find the page descriptor when it’s time to swap out a page.
- The *secondary-storage* field allows page tables to be made more compact. How?

- The *mode* field has four values.
  - *free*  
  - *in transition*
  - *in use*  
  - *locked in*

- Given a physical frame number, it is easy to derive the location of the page descriptor. How?

Each entry in the TLB contains six fields:

```
Key
```

```
| PID | Page # | Use | Dirty | Access rts. | Page-frame # |
```

- Process ID. Identifies the process.
- Key. The page number.
- Use bit. Records whether page has been used “recently.”
- Dirty bit. Records whether page has been written into “recently.”
- Access rights.
- Page frame.
Locked-in pages (S&G, §9.8.5): Why is it necessary to lock some pages in memory?

For one thing, e.g.,

Consider I/O. I/O devices usually write to and read from physical addresses, not virtual addresses.

What problem does this pose?

There are two common solutions to this problem.

- Always allocate I/O buffers in the operating system, not in user space.
  Disadvantage?

- Lock in pages that have I/O requests in progress.

Another use of locked-in pages is to prevent a low-priority process’s pages from being reclaimed before they are ever used.

Inverted page tables (S&G §9.8.3): We have assumed that a system maintains both a page table and a frame table.

The frame table holds descriptor addresses. Then from each frame-table entry, the corresponding page-table entry can be found.

Suppose instead of having a frame-table entry point at a page-table entry, the frame-table entry included all the information in a page-table entry.

Then we wouldn’t have to keep the page table in main memory at all!
What would be the advantage of this?

As processes’ address space grows large, it also tends to be sparsely populated.

When the frame table is used as a page table, it is called an inverted page table.

Systems using an inverted page table include—

- IBM System/38 and AS 400
- IBM RT
- IBM RS/6000
- HP Spectrum workstations.

Here is a simplified description of how the inverted page table is used in the RT.

A virtual address always has its process number prepended:

\(<\text{process-id}, \text{page \#}, \text{displacement}>\)

If there is a TLB hit, translation proceeds just as in a system with an ordinary page table.

If there is a TLB miss, the \(<\text{process-id, page \#}>\) pair is looked up in the inverted page table.

But the inverted page table is indexed by physical address, not virtual address. So in the worst case, it might be necessary to search all the way through the table.

Instead, a hash table is used to map each \(<\text{process-id, page \#}>\) pair to an inverted-page-table entry.

But whenever there is a page fault, the \(<\text{process-id, page \#}>\) pair will not be in the inverted page table.

Then (and only then) the (regular) page table must be brought into main memory. Since the regular page table is paged, only a single page needs to be brought in.
• Downside: A single page fault may now cause another page fault.
• Upside: It is not necessary to keep pages of the page table in memory just in case of a TLB miss. They are now needed only when a page fault occurs.

_Single-virtual-address-space systems:_ Inverted page tables are especially useful on systems where all processes share a single large address space.

In such systems, it is much easier to share pages and other objects.

But the address space will be used very sparsely by a single process. Hence, keeping regular page tables in main memory would be a waste.

**Performance of demand paging:** Let—

\[
    a \equiv \text{memory access time (usually 50–500 ns.)},
    p \equiv \text{probability that a memory reference causes a page fault, and}
    s \equiv \text{time to service a page fault.}
\]

Then the effective access time

\[
    e = (1-p) a + ps
\]

Major components of a page-fault service time are—

• Time to service the page-fault interrupt.
• Time to swap in the desired page.
• Time to restart the faulting process

Assume \( a = 100 \text{ ns.}, s = 1 \text{ ms.} \)

\[
    p = 10^{-3} \quad \Rightarrow \quad e =
\]

To achieve less than 10% degradation, we need \( p < \)
Replacement strategies: A replacement strategy answers the question,

When a page (or segment) must be brought in, and there is not enough free memory to do so, which page should be removed?

The goal of a replacement strategy is to minimize the fault rate. To evaluate a replacement algorithm, we need to specify—

- the size of a page,
- a set of reference strings (or, “page traces”), and
- the number of page frames.

Given these inputs, we can determine the number of page faults, and hence the page-fault rate.

A page trace (or reference string) is a list of pages in order of their reference by the processor. For example, the following is a page trace:

\[ a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e \]

Given a particular size memory, we can simulate which pages would be in memory for any replacement algorithm.

- FIFO—Remove the page that has been in memory the longest.

This requires implementation of a FIFO queue in which pages are ordered by arrival time.

This is not always a good algorithm, because a page that is only used once remains in memory as long as a page that is used heavily.

It also suffers from an anomaly (Belady’s anomaly): sometimes increasing the size of memory increases the fault rate.

Example: Suppose the algorithm is FIFO and there is only one page frame. The letters above the line give the page trace, and the letters below the line indicate which pages are in memory. An asterisk indicates a page fault.

In a one-page memory: 12 faults.
Indeed, since there is only one page frame, and the same page is never referenced twice in a row, each reference produces a fault.

In a two-page memory: 12 faults.

In a three-page memory: _____ faults.

In a four-page memory: 10 faults.

In a five-page memory: 5 faults.
• OPT replacement. Replace the page that will never be needed again, or at least will be needed furthest in the future. Priority lists are used throughout.

Notice that pages in the lower right have the same priority.

In a one-page memory: 12 faults.

```
abc dabeabcde
************
```

In a two-page memory: 9 faults.

```
abcdabeabcde
**** ** ** *
```

In a three-page memory: 7 faults.

```
abcdabeabcde
* * * *** *
```

In a four-page memory: 6 faults.

```
abcdabeabcde
**** * *
```

In a five-page memory: 5 faults.

```
abcdabeabcde
**** *
```

Though unrealizable, the OPT algorithm provides a standard to measure other replacement algorithms against.
A replacement algorithm satisfies the *inclusion property* if the set of pages in a \(k\)-page memory is always a subset of the pages in a \((k+1)\)-page memory.

The OPT algorithm satisfies the inclusion property.

- The LRU algorithm. Replace the page that was *least recently used.*

Based on the assumption that the recent past can be used to predict the near future.

The LRU algorithm satisfies the inclusion property. Here is an example of its performance on our page trace:

In a one-page memory: 12 faults.

\[
\begin{align*}
&ab
cda\quad bea\quad bcd\quad e \\
&ab
cda\quad bea\quad bcd\quad e
\end{align*}
\]

In a two-page memory 12 faults.

\[
\begin{align*}
&ab\quad cda\quad bea\quad bcd\quad e \\
&ab\quad cda\quad bea\quad bcd\quad e
\end{align*}
\]

In a three-page memory: 10 faults.

\[
\begin{align*}
&abcd\quad a\quad be\quad abc\quad de \\
&abcd\quad ab\quad c\quad de
\end{align*}
\]

In a four-page memory: 8 faults.

\[
\begin{align*}
&abcd\quad a\quad be\quad abc\quad de \\
&abcd\quad ab\quad c\quad de
\end{align*}
\]
In a five-page memory: 5 faults.

<table>
<thead>
<tr>
<th>(a^*)</th>
<th>(b^*)</th>
<th>(c^*)</th>
<th>(d^*)</th>
<th>(a)</th>
<th>(b)</th>
<th>(e^*)</th>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
<th>(d)</th>
<th>(e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(b)</td>
<td>(c)</td>
<td>(d)</td>
<td>(a)</td>
<td>(b)</td>
<td>(e)</td>
<td>(a)</td>
<td>(b)</td>
<td>(c)</td>
<td>(d)</td>
<td>(e)</td>
</tr>
<tr>
<td>(a)</td>
<td>(b)</td>
<td>(c)</td>
<td>(d)</td>
<td>(a)</td>
<td>(b)</td>
<td>(e)</td>
<td>(a)</td>
<td>(b)</td>
<td>(c)</td>
<td>(d)</td>
<td></td>
</tr>
<tr>
<td>(a)</td>
<td>(b)</td>
<td>(c)</td>
<td>(d)</td>
<td>(a)</td>
<td>(b)</td>
<td>(c)</td>
<td>(d)</td>
<td>(e)</td>
<td>(a)</td>
<td>(b)</td>
<td></td>
</tr>
</tbody>
</table>

**Stack algorithms and priority lists:** An algorithm that satisfies the inclusion property is known as a *stack algorithm*. The name comes from the way that the list of pages in memory can be updated after each memory reference.

At each memory reference, the referenced page is either already in memory or it isn’t.

If it is in memory at distance \(i\) from the top of the stack, the list of referenced pages changes as shown on the left, below. If it isn’t in memory, the changes are shown at the right.

Referenced page already in memory

Referenced page not in memory

Referenced page

already in memory

not in memory

referred page victim
When a new page needs to be brought in, the page that gets replaced is defined to have the lowest priority of all pages in memory.

In the LRU algorithm, the page at depth $n$ (where $n =$ # of page frames) is always the one with the lowest priority.

For other stack algorithms, this is not always the case. In OPT, for example,

- a page’s position on the stack is related to how long in the past it was last referenced, but
- its priority depends on how soon in the future it will be referenced.

This diagram shows the priority lists and stacks for a sample reference string.

```
abcadbadcd
abcadbadcd
*** *
aacaabadc
bbbddbaa
cccccbb
aaabbadcda
bbaadc dab
ccdcaabc
cb bb cd
```

In any priority list, underlined pages have the same priority.

To update the stack, we must compare the priorities of pages to determine whether they should move downward on the stack. At each comparison, the lower-priority page moves downward, but the higher-priority page does not.
Let's consider an example of stack updating from the OPT example above.

**Approximations to LRU page-replacement algorithm:** Worst problem with LRU is that it requires a lot of hardware assistance.

Since direct support for LRU replacement is expensive, approximations are usually used. Most of these methods employ a *used bit* for each page.

- When the page is referenced (read or written), the used bit is set to 1 (by hardware).
- The used bits are reset to 0 periodically (e.g. when a page is swapped out).

The *clock* algorithm is an important LRU approximation.

It also uses a *dirty bit*, which tells whether a page has been written recently. It cycles through the frame table, changing the settings of the used and dirty bits.

When the algorithm reaches the end of the frame table, it goes back to the beginning. Thus, the table can be viewed as circular, like a clock:
The algorithm proceeds as follows. Hand is a variable whose value is saved from one call to the next.

```
while not done do
begin
  if used = 1 then used := 0 {in the entry pointed to by hand}
  else if dirty = 1 then begin
    save := dirty;
    dirty := 0
  end
  else begin
    victim := hand; {Mark this page for swap out}
    done := true;
  end;
  hand := hand + 1 mod table_length;
end;
```

Notice that—

- if the algorithm encounters an in-use page frame with (used bit, dirty bit) = (1,1) or (1,0) it turns off the used bit and proceeds to the next in-use frame. (*First chance*)

- if the algorithm finds (used bit, dirty bit) = (0,1) it saves the value of the dirty bit and changes the state to (0,0). (*Second chance*)

- it gives the dirty page a third chance.
Why does the algorithm favor leaving *dirty* pages in memory?

The “second-chance” algorithm given in Silberschatz & Galvin is a simpler version of the clock algorithm that doesn’t use the dirty bit.

*Page replacement vs. segment replacement:* In segment replacement

- To bring one segment in, more than one might need to be kicked out.
- The segments to be replaced must be
- Segments are accumulated (via LRU approximation, etc.) until total (contiguous) memory is large enough to satisfy request.

Then swap out or release segments in order of descending size. (Not all may need to be released.)

*Page replacement in DSM systems:* As in any other system, when a DSM system needs to bring in a page, there may not be any free page frames.

Thus, a page needs to be evicted from memory:

- Which page?
- Where should it be put?

*Which page?* A DSM system is different from other VM systems, in that

- Pages may be invalidated spontaneously due to actions of other processes.
- A “clean” page in the local memory may also be present in other memories.

When a page is evicted, the owner or page manager must be informed of this decision.
If no replicated page is a suitable victim, then a non-replicated page must be chosen, often by some LRU approximation.

**Where to put it?** There are two choices.

- Back on disk.

It might be sent back to its “home” processor. But this might require the home processor to reserve a lot of memory for returned pages.

Another strategy would be to “piggyback” the number of free page frames on each message sent.

Then a faulting processor could hand off a page to a processor with plenty of free memory.

**Locality of reference:**

- Temporal locality. Once a location is referenced, it is often referenced again very soon.
- Spatial locality. Once a location is referenced, a nearby location is often referenced soon after.

Good, modular programs have high locality. See the graph of program locality in §9.7.1 (p. 319) of Silberschatz & Galvin.

**Importance of program structure** (S&G §9.8.4): Sometimes the order in which data items are referenced has a great effect on locality.

Consider this program fragment, and assume that pages are 128 words long.

```plaintext
var A : array [1..128] of array [1..128] of integer;
for j := 1 to 128 do
  for i := 1 to 128 do
    A[i, j] := 0;
```

Notice that the array is stored in row-major order:
Thus, each row takes one page.

What happens if the system has at least 128 page frames to hold the array?

What happens if the system has less than 128 pages for the array?

Changing the code can allow the program to execute efficiently with only a single page frame for the array:

```pascal
var A: array [1..128] of array [1..128] of integer;
A[i, j] := 0;
```

For efficient access, the compiler can try to not to split data structures across page boundaries, and to keep routines that call each other on the same page.

User-level memory managers: In the Mach operating system, most secondary-storage objects are mapped into the address space.

This means that files, databases, etc. are assigned virtual addresses. To reference them, the program simply generates their address; no special I/O calls need to be used.

The programmer may know something about the way that a particular object will be referenced. (S)He can then choose a page-replacement algorithm that works well with that structure.

For example, what would be a good page-replacement algorithm for a sequential file?

In Mach, the kernel doesn’t decide what page-replacement algorithm is used.

Rather, a separate page-replacement algorithm could be used for each memory object.

Mach provides a default memory manager, but the programmer does not have to use it for each memory object.
Thrashing and the working-set model: Most programs have a threshold beyond which

- incremental increases in memory → relatively little advantage, but
- a small decrease in memory → drastic increase in page-fault rate.

A plot of the number of faults as a function of the number of pages is known as a Parachor curve.

Informally, a process’s working set is the set of pages that must be in memory for the process to execute “efficiently.”

If a process does not have “enough” pages, the page-fault rate is very high

⇒ low CPU utilization.
⇒ operating system thinks it needs to increase the degree of multiprogramming
⇒ another process added to the system.

Thrashing means that the operating system is busy swapping pages in and out.

Definition: The working set of a process \( W(t, \Delta) \) at time \( t \) is the set of pages referenced by the process during the process-time interval \( (t-\Delta, t) \).

\( \Delta \) is called the window size. (Note that the interval is measured in terms of process time, not real time.)
For example, if a process executes for \( \Delta \) time units and uses only a single page, then \( |W(t, \Delta)| = 1 \).

Note that the working-set size is a monotonic function of \( \Delta \).

A working set is a reliable estimate of a process’s needs if it is large enough to contain pages being frequently accessed, and no others.

- \( \Delta \) too small \( \Rightarrow \) will not encompass working set.
- \( \Delta \) too large \( \Rightarrow \) will encompass several localities.

The working-set principle says that a process should be eligible for execution iff its working set is resident in main memory. A page may not be removed from main memory if it is in the working set of a process.

Thrashing occurs when the working sets of the active processes cannot all fit in memory:

\[ \sum \text{size of working sets} > \# \text{page frames} \]

If this happens, what should be done?

If \( \sum \text{size of working sets} \ll \text{number of page frames} \),

The working set is a local policy that allows the number of page frames used by executing processes to vary.

When a process is suspended, the pages of its working set will eventually be swapped out. When it is reactivated, the working-set principle says the pages of its working set should be prepaged in. Why?

\textit{Properties of working sets:} The size of a working set can vary. If a process uses \( n \) pages, then

\[ 1 \leq |W(t, \Delta)| \leq \min(\Delta, n). \]

So a fixed partition need not be allocated to a process.
Working sets are inclusive:

\[ W(t, \Delta) \subseteq W(t, \Delta + 1) \]

Working sets tend to increase temporarily as a program moves from one phase of execution to another:

Kahn showed that stable phases occupied about 98% of process time; nearly half the faults occurred during the other 2% of process time.

*Keeping track of working-set size:* The Maniac II implementation was due to James B. Morris. It uses—

- a *working-set register*, that contains one bit for each page frame in main memory;
  - only the bits for the page frames of the active process are set.
- a *T* register, that controls the frequency of updating the counters described below; and
- a set of *page-frame registers* that contain several fields:
  
  \[
  \begin{array}{ccc}
  D & A & \text{clock} \\
  \end{array}
  \]

  - \( D \) is the dirty bit;
  - \( A \) is the alarm bit; and
  - \( \text{clock} \) is a four-bit counter.

The algorithm works like this:

- On each page reference, the alarm bit and counter are cleared. If the reference was a write, the dirty bit is set.
• Periodically, at an interval determined by the $T$ register, the counters are incremented, but only counters for those pages whose working-set register bits are on.

• The alarm bit is set whenever the associated clock overflows. When the alarm bit is set, the page is no longer in the working set of its process, so it is eligible to be swapped out.

The $T$ register’s setting can be changed to “tune” the operation of the working-set replacement.

**The WSClock algorithm:** One problem with the Maniac II algorithm is the overhead of updating all the counters, especially when the number of page frames is large.

The **WSClock algorithm** approximates working-set replacement.

• As hand passes over a page frame $f$, its used bit is tested and reset as in the clock algorithm.

• If $f$’s used bit was set, its time of last reference is assumed to be the current process time $p_t$. This time is stored in the $lf$ field of $f$’s frame-table entry.

• If the bit was not set, the page is replaceable iff

$$p_t - lf[f] > T,$$

where $T$ is the working-set parameter.

One variety of the WSClock algorithm uses a dirty bit as well as a used bit.

When the hand passes a page whose dirty bit is set, that page is queued for “cleaning”—being written out to disk.

WSClock can be used for load control:

• When the hand goes all the way around the clock without encountering a replaceable frame, WSClock detects *memory overcommitment*.

• The page-cleaning queue is then examined.
  ◦ If there is an outstanding request for a page to be cleaned, it is processed to yield a replaceable page.
  ◦ If there is no request for a page to be cleaned, a process is suspended.
Load control in Unix™ (S&G, §21.6): Early (pre 3-BSD) versions of Berkeley Unix employed process swapping; more recent versions use paging.

Swapping: In BSD systems that swap processes, system and user memory are kept in contiguous memory. This includes—

- System memory
  - “u area” (like the PCB), and
  - kernel stack.
- User memory
  - nonsharable code,
  - data, and
  - stack.

Why do you suppose this is done?

What problem does this exacerbate?

Why is sharable code handled separately?

Memory is allocated to processes using first-fit.

If a process gets too large to fit in its area,

- some systems look for contiguous memory at the end of currently allocated memory, or
- the whole process is moved to a different area.

Swapping decisions are made by the scheduler, or swapper.

- It wakes up at least once every four seconds.
- A process is more likely to be swapped out if it—
  - is idle,
  - has been in main memory a long time, or
• A process is more likely to be swapped back in if it—
  °
  °

Swap space in 4.3 BSD is allocated in pieces that are—

• multiples of a power of 2 and a minimum size (e.g., 32 pp.)
• up to a maximum determined by the size of the swap-space partition on the disk.

Paging: Unix uses demand paging, except that when a process is started, many of its pages are prepaged.

Why is this done?

But these pages are marked “invalid,” which means that they are awaiting swap-out. Why?

Pageouts are done in clusters, too.

  On the VAX, I/O is done in 2-page chunks, or

If a page is still in the page table, but is marked “invalid” pending swap-out, it can be marked valid and “reclaimed.”

When paging is used, all code is by default shared and read only.

The VAX, the second architecture on which Unix was implemented, did not have a used bit!

So, how could it use a paging algorithm that favored pages that were recently referenced?

The second-chance algorithm is used.

• A page is marked invalid when the hand sweeps past it.
• Upon reference to an invalid page, a page fault is generated.
• But the page-fault handler
Later versions of BSD Unix use the clock algorithm. [\]

The paging algorithm tries to keep “lots of” frames free.

- The \emph{lotsfree} parameter is usually set at 1/4 of the frames.
- The \emph{pagedaemon} wakes up several times a second to see if it needs to examine any frames.
- The number of frames examined is determined by—
  - amount lacking to reach \emph{lotsfree}, and
  - number of frames the scheduler thinks it needs.
- Which pages it chooses is influenced by the priority of the processes that are using the frames.
- But it quits examining frames when the number of free frames reaches \emph{lotsfree}.

If the scheduler thinks that the paging system is overloaded, it will swap out processes.

Necessary conditions for swap-out:

- Load average is high.
- Current amount of free memory is too low.
- Not only that, but the average amount memory free recently is too low.

\textit{Performance of paging algorithms}: WSClock performs just about as well as Clock or the exact working-set method when all three were tuned for best performance.

- Clock performed much worse than the other two before tuning. Thus, it may be more sensitive to its parameters. (Its feedback mechanism is less direct than those based on working sets.)
- Clock is easier to implement and more efficient to execute.
Distributed process management: Processes need to interact in controlled ways.

This means that they need to update shared information, so mutual exclusion is necessary.

But there may be no shared memory to allow process-coordination primitives to be used.

An algorithm is either centralized or distributed.

- A centralized algorithm is characterized by these properties.
  - Only the central node makes a decision.
  - All necessary information is held at the central node.

Sometimes, central control may migrate from node to node.

Problems with this approach:

- 
- 

- A fully distributed algorithm has these properties.
  - All nodes have an equal amount of information.
  - All nodes make a decision based solely on local information.
  - All nodes bear equal responsibility for a final decision.
  - All nodes expend equal effort in effecting a final decision.
  - Usually, failure of a node does not cause system collapse.

Many algorithms are distributed, but few satisfy all of these properties.
Two fundamental assumptions are common to all distributed algorithms.

• Each node has only a partial picture.
• There is no clock that is accessible systemwide.

This raises the problem of event ordering.

Event ordering (§18.1 of S&G): In a uniprocessor, events are totally ordered.

For any two events $A$ and $B$, the clock times at which they occur determines unambiguously which was first.

In a multiprocessor, the exact sequence of events cannot be determined unless the clocks are “perfectly synchronized.”

Since it takes time to communicate clock values, events in a multiprocessor are not totally ordered.

Sometimes, however, it is necessary to know which of two events happened first.

How can the partial ordering of events in a distributed system be extended to a total ordering? (Lamport)

Definition: Assume—

• all interprocessor communication is via messages,
• a message is always sent before it is received.

Then the precedes (or happened-before) relation $\rightarrow$ is defined by

• If $A$ and $B$ are events in the same process, and $A$ was executed before $B$, then $A \rightarrow B$.
• If $A$ is the sending of a message and $B$ is the receiving of it, then
  $A \rightarrow B$.
• If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$.

Thus “$\rightarrow$” is a quasi order.
Definition: For two events \( A \) and \( B \), if not \( (A \rightarrow B \text{ or } B \rightarrow A) \), then \( A \) and \( B \) are said to have been executed *concurrently*.

The *happened-before* relationship can be illustrated by a space-time diagram.

- Horizontal dimension: space.
- Vertical dimension: time.
- Wavy line denotes a message from one processor to another.

**Example:** In the diagram at the right, of the event pairs listed, which event happened first?

\[
\begin{align*}
& p_1 & q_2 & p_3 & r_0 \\
& p_1 & q_4 & q_3 & r_0 \\
& p_2 & q_0 & q_3 & r_1 \\
& p_3 & q_3 & q_4 & r_0
\end{align*}
\]

**Implementation of happened-before without a global clock:**

- Each process \( p_i \) contains a logical clock \( LC_i \).
- \( LC_i \) assigns a unique number to each event in \( p_i \).
- All messages are timestamped by the sender.
- If \( p_i \) receives a message with a timestamp \( t \) greater than the current value of \( LC_i \), it sets \( LC_i \) to \( t+1 \).

A total ordering can be defined based on the time-stamp partial ordering.

**Distributed mutual exclusion** (S&G §18.2.2): Suppose we want to perform mutual exclusion in a distributed system.

We want to satisfy the same properties as in a uniprocessor: mutual exclusion, no mutual blocking, and bounded waiting.

Specifically, we need to make sure requests to enter the critical section are granted in the same order they are made.

Lamport used his distributed event-ordering algorithm to design a distributed mutual exclusion algorithm.
In an \( N \)-process system, Lamport’s algorithm required \( 3(N-1) \) messages for each critical section entry.

Ricart and Agrawala refined the algorithm to require only \( 2(N-1) \) messages.

\textit{Algorithm:}

- When a process \( p \) wants to enter its critical section, it generates a new time stamp \( TS \) and sends the message 
  \[ \text{request } (p, TS) \]
  to all processes in the system (excluding itself).

- When a process receives a request message from \( p \), it can either
  - send an immediate \textit{reply} back to \( p \), or
  - defer a reply, in favor of sending it later (“puts it on hold”).
    
    The list of all processes \( p \) has “on hold” is called \( p \)’s \textit{request list}.

How does a process \( p \) decide whether to reply immediately?

- If \( p \) is in its critical section, it defers its reply.
  
  Immediately upon exiting, it sends \textit{replies} to all processes on its request list.

- If \( p \) is not in its critical section and does not want to enter, it sends an immediate \textit{reply}.

- If \( p \) wants to enter its critical section, it compares the timestamp of the new message (from, say, process \( q \)) with the timestamp of all the request messages it sent out.
  
  √ If the timestamp of \( q \)’s message is earlier, then a reply is sent to \( q \).
  
  √ Otherwise \( q \)’s message is added to the request list.

- When \( p \) has received replies from all other processes, then it enters its critical section.
Here is a state-transition diagram for a process in this algorithm:

Processes must be informed about the creation or failure of another process; if any process fails, the algorithm fails.

Example: Three processes, \( p_1 \), \( p_2 \), and \( p_3 \):
Here is an Ada version of the algorithm.

- Each process is implemented by two tasks:
  - USER_TASK, which requests use of the resource, and
  - MUTUAL_EXCLUSION, which executes the mutual-exclusion algorithm when called by USER_TASK.

- MUTUAL_EXCLUSION defines five entries:
  - INVOKE requests access to the critical section,
  - WAIT, which waits for permission,
  - RELEASE, to release mutual exclusion,
  - RECEIVE_REQUEST, which receives request messages from other tasks, and
  - RECEIVE_REPLY, which receives replies from other tasks.

```
task body USER_TASK is
  begin
    MUTUAL_EXCLUSION.INVOKE;
    MUTUAL_EXCLUSION.WAIT;
    -- -- Critical section
    MUTUAL_EXCLUSION.RELEASE;
    ...
  end USER_TASK;

task MUTUAL_EXCLUSION is
  entry INVOKE;
  entry WAIT;
  entry RELEASE;
  entry RECEIVE_REQUEST
    (k, j: in INTEGER);
    -- -- k is sequence # of request.
    -- -- j is # of node that sent request.
  entry RECEIVE_REPLY;
  end;
```
task body MUTUAL_EXCLUSION is
  me: constant := ... ; -- unique # of node
  N: constant := ... ; -- # nodes in system
  OSN: INTEGER := 0; -- sequence # to
    -- be assigned to next local request
  HSN: INTEGER := 0; -- highest seq. #
    -- seen in any request msg. sent or rec'd.
  ORC: INTEGER := 0; -- # of reply msgs.
    -- still expected.
  RCS: BOOLEAN := FALSE; -- FALSE
    -- unless this node requesting
    -- access to c.s.
  RD: array (1 . . N) of BOOLEAN
    := (FALSE, FALSE, ... , FALSE);
    -- RD is TRUE when this node is
    -- deferring reply to node j's msg.

begin
  loop
    select
      accept INVOKE do
        OSN := HSN + 1;
        RCS := TRUE;
        ORC := N - 1;
        for j in 1 . . N loop
          if j ≠ me then
            send request msg. to node j;
          end if;
        end loop;
      end;
      or
      when ORC = 0 =>
        accept WAIT;
      or
      accept RELEASE loop
        RCS := FALSE;
        for j in 1 . . N loop
          if RD(j) then
            send reply message to j;
            RD(j) := FALSE;
          end if;
        end loop;
      end;
      or
accept RECEIVE_REQUEST
    (k, j: in INTEGER) do
declare
    DEFER: BOOLEAN;
begins
    HSN := max(HSN, k);
    DEFER := RCS and ((k > OSN)
        or (k = OSN and j > me));
if DEFER then RD(j) := TRUE;
else send reply(j) message;
end if;
end;
end;
or
accept RECEIVE_REPLY do
    ORC := ORC – 1;
end;
end select;
end loop;
end MUTUAL_EXCLUSION;

What is the purpose of the compound condition

\(((k > OSN) \text{ or } (k = OSN \text{ and } j > me))\)\

Notice that requests are totally ordered.

- Our process cannot enter the critical section until it gets permission from everyone else.
- Anyone else who has been waiting “longer” (= lower sequence #) than our process will not give its permission.
- Ties are resolved on the basis of node numbers.

Starvation cannot occur since ordering is FCFS.

Deadlock cannot occur because no cycle of waiting nodes can occur.

Control is fully distributed. All nodes execute exactly the same algorithm.
Maekawa’s square-root algorithm: Ricart and Agrawala’s algorithm achieves mutual exclusion with fully distributed control, but it requires a lot of messages.

Is it necessary for a node to query all other nodes before it is allowed to enter the critical section?

No! It can be allowed to query a subset of the nodes as long as it can be guaranteed that no two nodes can enter the critical section at the same time.

Idea: In an N–node system, create N different subsets of the nodes, such that each subset overlaps every other subset!

[That is, for each node i, define a subset S_i such that

\[ S_i \cap S_j \neq \emptyset \]

for any combination of i and j, 1 ≤ i, j ≤ N.]

Then, if a node succeeds in “locking” all the nodes in its set, it can be guaranteed that no other node has locked all the nodes in its set.

Here is an example of mutually overlapping subsets of the integers 1 to 7:

- \( S_1 = \{1, 2, 3\} \)
- \( S_2 = \{2, 4, 6\} \)
- \( S_3 = \{3, 5, 6\} \)
- \( S_4 = \{1, 4, 5\} \)
- \( S_5 = \{2, 5, 7\} \)
- \( S_6 = \{1, 6, 7\} \)
- \( S_7 = \{3, 4, 7\} \)

If we want to send the minimum number of messages, we don’t want to have—

- The set size \( K \) be too large, or then

  (In the extreme, if \( K = N \), then we are back to the previous algorithm.)

- The set size \( K \) be too small, or then

What is the smallest set size we can tolerate? The smallest number that will make equal-size sets overlap!

What is that?
Each of the \( K = \sqrt{N} \) elements is in \( \sqrt{N} \) sets. Each element overlaps \( \sqrt{N} \) sets; so the \( \sqrt{N} \) elements overall overlap approximately \( N \) sets.

Now we see why each set above had three members for \( N = 7 \).

*The algorithm in overview:* A node tries to lock all the other nodes in its set.

- If it succeeds, it can enter the critical section.
- If it fails, it waits for an already locked member to be freed and then locks it (repeatedly, till they are all locked).
- Deadlock is prevented by having a node yield whenever the timestamp of its request > the timestamp of another request.

*The algorithm in more detail:*

1. Before entering the critical section, node \( i \) sends a *request* message to all nodes in its set (including itself).

2. When it receives a *request* message, each member of the set checks to see if it (the receiving node) is locked for some other request.

   - If not, it—
     - sets a lock, and
     - returns a *locked* message to node \( i \) (which succeeded in getting the lock).

     The current request is now called the *locked_request*.

   - If it is locked, it—
     - places the *request* on its *queue* (ordered by timestamp), and
     - checks to see if the *locked_request* or any request in the queue has an earlier timestamp than this *request*.

     » if so, a *failed* message is returned to node \( i \).
» if not, an inquire message is sent to the node that has this node locked (i.e., the one that sent the locked_request).

The inquire message asks whether the originating node has successfully locked all the nodes in its set.

(An inquire does not need to be sent if the last inquire has not been responded to.)

3. When a node receives an inquire message, it—
   • returns a relinquish message if it knows it won’t succeed in entering the c.s. (i.e., if it has received a failed message from \( \geq 1 \) of the nodes in its set.)
   This frees the inquiring node to service a request with an earlier timestamp.
   • defers a reply if it doesn’t know if it will succeed in entering the c.s.
   • if it has succeeded in entering the c.s., waits until it is done, then returns a release message. (If a release message has already been sent, another one is not sent.)

4. When a node receives a relinquish message, it—
   • dequeues the request with the earliest timestamp and sends a locked message to its sender, and
   • queues the old locked_request.

5. When a node has received locked messages from all the nodes in its set, it enters the c.s.

6. When the node exits the c.s., it sends release messages to each member of its set.

7. When a node receives a release message, it—
   • deletes the locked_request, and
   • dequeues the earliest request on the queue (if any) and—
     ○ makes it the locked request, and
     ○ returns a locked message to its sender.
Adding and deleting nodes: When a new node is added to the network, it must—

- Interrogate some active node to get a list of participating nodes.
- Assign itself a unique node number.
- Have its node number placed on all other nodes’ list of participating nodes.
- Assign itself an initial timestamp > current values of all other timestamps.

Must defer all replies until it acquires a timestamp.

When a node leaves the network, it must notify all other nodes of its intention.

In the meantime, it cannot participate in any communication that will affect mutual exclusion.

A time-out mechanism should be used to remove a node from the network if it doesn’t reply to a request within a certain amount of time.

Distributed concurrency control: We have been concerned with synchronizing access to data at different sites.

We can consider the problem as one of assuring correct accesses to a distributed database.

A database

- consists of a collection of objects (e.g., files, records),
- each viewed as a single-unit resource
- on which a basic read or write access can be performed.

A database computation is a sequence of such actions.
Consistency constraints are a set of assertions that a database must satisfy. E.g.—

• In a financial database, the balance of each account = sum of deposits – sum of withdrawals.

A transaction is a sequence of actions that

• constitutes a logical unit of work, and
• takes the database from one consistent state to another.

For performance reasons, concurrent transactions are desirable.

However, arbitrary interleaving of transactions can lead to inconsistent states. Here is an example:

Initial balance in accounts:
A – $400 B – $200 C – $100

Transactions

$T_1$: begin { A pays B $50 }
read acct. A to obtain A’s balance
read acct. B to obtain B’s balance
write A’s balance – $50 to acct. A
write B’s balance + $50 to acct. B
end

$T_2$: begin { B pays C $100 }
read acct. B to obtain B’s balance
read acct. C to obtain C’s balance
write B’s balance – $100 to acct. B
write C’s balance + $100 to acct. C
end

$T_3$: begin { Print balances }
read acct. A to obtain A’s balance
read acct. B to obtain B’s balance
print A’s balance
print B’s balance
end

What should B’s final balance be?

If the transactions run concurrently, what could it be?

This is called the lost update anomaly.
Also, if transaction $T_3$ reads accounts $A$ and $B$ during the execution of $T_1$, it could conclude that money had “disappeared” from the accounts.

This is called the *inconsistent retrieval* anomaly. Temporary inconsistency is unavoidable. It is only possible to enforce consistency at the end of a transaction.

*Database model:* The database resides in a network composed of a set of nodes called *sites*.

Sites communicate via messages, which are assumed to arrive eventually, in the same order that they were sent.

The database consists of a set of *entities* (e.g., records, files) that serve as indivisible units of access.

- An entity is composed of one or more data objects,
- each of which is uniquely identified by an *(entity name, site_id)* pair.

*Replication* may be used: an entity may be represented by $> 1$ object.

- This may allow improved performance.
- It also improves availability (reliability).

The figure below shows the relationship between entities and objects.

![Diagram showing entities and objects]

- **Entity level:**
  - $E_1$
  - $E_2$
  - $E_3$

- **Object level:**
  - $<E_1, S_1>$
  - $<E_2, S_1>$
  - $<E_1, S_2>$
  - $<E_2, S_2>$
  - $<E_3, S_3>$

- **Sites:**
  - Site $S_1$
  - Site $S_2$
  - Site $S_3$
An entity \( E_i \) replicated at sites \( S_{i1}, \ldots, S_{ik} \) is represented by the objects
\[
\langle E_i, S_{i1} \rangle, \ldots, \langle E_i, S_{ik} \rangle
\]

**Operations of the transaction manager (TM):**

- \( \text{read}(E_i) \) returns the value of entity \( E_i \).
- \( \text{write}(E_i, \text{value}) \) sets \( E_i \) to the specified value.
- **begin** and **end** are used to bracket a transaction.

Each read request in a transaction \( T \) is translated by the TM into a read action
\[
(T, \text{read}_\text{object}, \langle E, S \rangle, V)
\]
where \( V \) is the variable that will receive the value and \( S \) is one of the sites where the object resides.

Each write request is translated to a write action
\[
(T, \text{write}_\text{object}, \langle E, S \rangle, V),
\]
one for each site where the object resides, and where \( V \) is the value to be written.

Also, the TM generates a
\[
(T, \text{commit}_\text{object}, \langle -, S \rangle, -)
\]
for each site \( S \) where any action was performed, to indicate successful completion of a transaction at site \( S \).

A transaction in a distributed database should “look” just like a transaction in a centralized database. This \( \Rightarrow \) 4 requirements:

1. **Location transparency:** Programmer need not be concerned with the location of data (even though data may move around).

2. **Replication transparency:** Programmer may treat each entity as a single item, regardless of whether it has been replicated.

3. **Concurrency transparency:** Programmer may view this transaction as being the only activity in the system.

4. **Failure transparency:** It must appear as if either all or none of the actions within a transaction have completed.
The results of a transaction should survive future hardware and software failures.

*Concurrency control based on locking:* There are two main methods for assuring that the above requirements are met by distributed transactions:

- locks and timestamps.

(We will not have time to consider timestamp-based schemes.)

In a lock-based scheme, objects are locked to assure that conflicting accesses do not occur.

There are three lock-related actions on objects:

- \((T, \text{lock}_X\langle E, S \rangle)\) denotes a request by transaction \(T\) for an exclusive lock on object \(\langle E, S \rangle\).
  - This is granted only when no other object is holding a lock on \(\langle E, S \rangle\).
- \((T, \text{lock}_S\langle E, S \rangle)\) denotes a request by transaction \(T\) for a shared lock on object \(\langle E, S \rangle\).
  - This is granted only when no other object is holding an exclusive lock on \(\langle E, S \rangle\).
- \((T, \text{unlock}\langle E, S \rangle)\) denotes a release by \(T\) of a lock on \(\langle E, S \rangle\).

If a requested lock cannot be granted, the transaction is blocked until the lock becomes available.

*Structural properties of transactions:* A transaction is well formed if it—

- reads an object only while it is holding a lock on the object, and
- writes an object only while holding an exclusive lock on the object.

A transaction is called two-phase if it does not issue any lock requests after it has performed an unlock action.

- The first phase is the growing phase, during which locks may be requested.
- The second phase is the shrinking phase, beginning with the first unlock action.
A transaction is *strongly two-phase* if all unlocks are done at the very end of the transaction.

On the left below is a transaction that is well formed but not two-phase. On the right is a transaction that is well formed and two-phase.

\[
\begin{align*}
&T_1, \text{lock}_S(A, S), \\
&T_1, \text{read}_\text{object}(A, S), A_{-bal}, \\
&T_1, \text{lock}_S(B, S), \\
&T_1, \text{read}_\text{object}(B, S), B_{-bal}, \\
&T_1, \text{unlock}(B, S), \\
&T_1, \text{lock}_X(A, S), \\
&T_1, \text{write}_\text{object}(A, S), A_{-bal} - 50, \\
&T_1, \text{unlock}(A, S), \\
&T_1, \text{lock}_X(B, S), \\
&T_1, \text{write}_\text{object}(B, S), B_{-bal} + 50, \\
&T_1, \text{unlock}(B, S)
\end{align*}
\]

\[
\begin{align*}
&T_1, \text{lock}_X(A, S), \\
&T_1, \text{lock}_X(B, S), \\
&T_1, \text{read}_\text{object}(A, S), A_{-bal}, \\
&T_1, \text{write}_\text{object}(A, S), A_{-bal} - 50, \\
&T_1, \text{read}_\text{object}(B, S), B_{-bal}, \\
&T_1, \text{write}_\text{object}(B, S), B_{-bal} + 50, \\
&T_1, \text{unlock}(A, S), \\
&T_1, \text{unlock}(B, S)
\end{align*}
\]

For conciseness, the *commit_object* operations have not been shown.

It is a necessary and sufficient condition for consistency in a concurrent environment that all transactions be well formed and two-phase.

For recoverability, it is also advantageous for—

- all transactions to be strongly two-phase, and
- *commit_object* operations to immediately precede unlock operations.

To avoid executing a lock action on every copy of an entity, *primary copy locking* can be used.

- Each entity has one site designated as the lock manager.
- All requests for locks on the entity must be directed to this site.
- After a lock has been granted, subsequent reads or writes may take place on any copy, but
- all copies must be updated before the lock is released.

If we assume strongly two-phase transactions, we can assume that—
• A read_object action carries with it a request for a shared lock, if the object has not already been locked.
• A write_object action carries a request for an exclusive lock.
• A commit_object immediately precedes an unlock.

(This makes our code more concise.)

**Consistent schedules:** A consistent schedule is one that preserves database consistency. How can such a schedule be defined?

First, we need to define some preliminaries.

A legal schedule is a schedule in which transactions do not simultaneously hold conflicting locks.

Here is a legal schedule for our three-transaction example.

\[
\langle (T_1, \text{lock}_X\langle A, S \rangle), \\
(T_2, \text{lock}_X\langle B, S \rangle), \\
(T_1, \text{read}_\text{object}\langle A, S \rangle, T_1's \ A \_\text{bal}), \\
(T_2, \text{read}_\text{object}\langle B, S \rangle, T_2's \ B \_\text{bal}), \\
(T_2, \text{lock}_X\langle C, S \rangle), \\
(T_2, \text{read}_\text{object}\langle C, S \rangle, T_2's \ C \_\text{bal}), \\
(T_2, \text{write}_\text{object}\langle B, S \rangle, T_2's \ B \_\text{bal} - 100), \\
(T_2, \text{write}_\text{object}\langle C, S \rangle, T_2's \ C \_\text{bal} + 100), \\
(T_2, \text{commit}_\text{object}\langle -, S \rangle), \\
(T_2, \text{unlock}\langle B, S \rangle), \\
(T_1, \text{lock}_X\langle B, S \rangle), \\
(T_1, \text{read}_\text{object}\langle B, S \rangle, T_1's \ B \_\text{bal}), \\
(T_1, \text{write}_\text{object}\langle A, S \rangle, T_1's \ A \_\text{bal} - 50), \\
(T_1, \text{write}_\text{object}\langle B, S \rangle, T_1's \ B \_\text{bal} + 50), \\
(T_1, \text{commit}_\text{object}\langle -, S \rangle), \\
(T_1, \text{unlock}\langle A, S \rangle), \\
(T_3, \text{lock}_S\langle A, S \rangle), \\
(T_3, \text{read}_\text{object}\langle A, S \rangle, T_3's \ A \_\text{bal}), \\
(T_1, \text{unlock}\langle B, S \rangle), \\
(T_3, \text{lock}_S\langle B, S \rangle), \\
(T_3, \text{print} \ T_3's \ A \_\text{bal}), \\
(T_3, \text{read}_\text{object}\langle B, S \rangle, T_3's \ B \_\text{bal}), \\
(T_3, \text{print} \ T_3's \ B \_\text{bal}), \\
(T_2, \text{unlock}\langle C, S \rangle), \\
(T_3, \text{commit}_\text{object}\langle -, S \rangle), \\
(T_3, \text{unlock}\langle A, S \rangle), \\
(T_3, \text{unlock}\langle B, S \rangle) \rangle
\]
What would make this schedule illegal?

A *serial schedule* is a schedule in which one transaction completes all of its actions before the next transaction begins.

How could we turn the schedule above into a serial schedule?

Clearly, a serial schedule preserves consistency.

Schedules are *equivalent* if they have the same *dependency relation*. What is a dependency relation?

A dependency relation is a relation that consists of pairs of transactions $T_i$ and $T_j$ such that—

- $T_j$ uses a value that was most recently written by $T_i$,
- $T_j$ overwrites a value that was most recently written by $T_i$, or
- $T_j$ overwrites a value that was most recently read by $T_i$.

If two schedules have the same dependency relation, they are said to be *equivalent*.

Think of what that means:

Given a set of transactions, is it possible to have two legal schedules that are not equivalent?

If a schedule is equivalent to some serial schedule, then it is said to be *consistent*.

(I.e., updates could have occurred in a serial schedule the same way that they did in this parallel schedule.)

It can be shown that,

- in a centralized system,
- any legal schedule
- composed of well formed two-phase transactions
  is consistent.
How can this result be extended to a distributed system?

First, define a *site schedule* $Q_j$ for site $j$ as the sequence of actions that take place at that site.

**Theorem:** Let $Q_1, \ldots, Q_n$ be the site schedules for a set of transactions $\{T_1, \ldots, T_m\}$.

Then there is a global schedule $Q$ such that—

- the action sequences of $T_j$ and $Q_i$ are subsequences of $Q$, and
- if each $Q_i$ is legal, then $Q$ is legal.

**Proof:** Assume each site $j$ has a local clock $C_j$ similar to the one described last lecture.

Assume that the transaction manager maintains an imaginary transaction clock whose value is denoted $C_T$. If the TM is at site $i$, $C_T = C_i$ at the outset.

When the TM requests an action $A$ on a remote object at site $j$, it—

- increments $C_i$, and
- sends a request message to site $j$.

When site $j$ receives the message, it adjusts its clock:

$$C_j := \max(C_i, C_j)$$

When site $j$ executes action $A$ for this transaction, it—

- increments $C_j$, and
- sends an acknowledgment to the TM.

Upon receipt of the acknowledgment, the TM sets

$$C_i := \max(C_i, C_j) \quad \text{and} \quad C_T := C_i.$$

To form an equivalent global schedule,

- combine the site schedules, and
- sort the actions in ascending order on $(C_T, \text{site number})$.

This schedule $Q$ satisfies both properties of the theorem.
On the next page are two site schedules for concurrent execution of our example transactions.

In this example,

- Accounts A, B, and C are replicated at sites S1 and S2.
  - Therefore, each transaction must do a write_object at two sites.
  - It is necessary to lock objects at each site, since primary-copy locking is not used.
- The transaction managers for T1 and T3 are located at S1.
- The TM for T2 is located at S2.

Schedule at Site S1:

\[
\langle 
  (T_1, \text{lock}_X(A, S_1)), \\
  (T_2, \text{lock}_X(B, S_1)), \\
  (T_1, \text{read}_object(A, S_1), T_1's \ A_{bal}), \\
  (T_2, \text{lock}_X(C, S_1)), \\
  (T_2, \text{write}_object(B, S_1), T_2's \ B_{bal} - 100), \\
  (T_2, \text{write}_object(C, S_1), T_2's \ C_{bal} + 100), \\
  (T_2, \text{commit}_object('\ - ', S_1)), \\
  (T_2, \text{unlock}(B, S_1)), \\
  (T_1, \text{lock}_X(B, S_1)), \\
  (T_1, \text{read}_object(B, S_1), T_1's \ B_{bal}), \\
  (T_1, \text{write}_object(A, S_1), T_1's \ A_{bal} - 50), \\
  (T_1, \text{write}_object(B, S_1), T_1's \ B_{bal} + 50), \\
  (T_1, \text{commit}_object('\ - ', S_1)), \\
  (T_1, \text{unlock}(B, S_1)), \\
  (T_3, \text{lock}_S(A, S_1)), \\
  (T_3, \text{read}_object(A, S_1), T_3's \ A_{bal}), \\
  (T_1, \text{unlock}(B, S_1)), \\
  (T_3, \text{lock}_S(B, S_1)), \\
  (T_3, \text{print} \ T_3's \ A_{bal}), \\
  (T_3, \text{read}_object(B, S_1), T_3's \ B_{bal}), \\
  (T_3, \text{print} \ T_3's \ B_{bal}), \\
  (T_2, \text{unlock}(C, S_1)), \\
  (T_3, \text{commit}_object('\ - ', S_1)), \\
  (T_3, \text{unlock}(A, S_1)), \\
  (T_3, \text{unlock}(B, S_1)) \rangle
\]
Schedule at Site $S_2$:

\[
\langle 
(T_1, \text{lock}_X(A, S_2)), \\
(T_2, \text{lock}_X(B, S_2)), \\
(T_2, \text{read}_\text{object}<B, S_2), T_2\text{’s B}_{\text{bal}}), \\
(T_2, \text{lock}_X(C, S_2)), \\
(T_2, \text{read}_\text{object}(C, S_2), T_2\text{’s C}_{\text{bal}}), \\
(T_2, \text{write}_\text{object}(B, S_2), T_2\text{’s B}_{\text{bal}} - 100), \\
(T_2, \text{write}_\text{object}(C, S_2), T_2\text{’s C}_{\text{bal}} + 100), \\
(T_2, \text{commit}_\text{object}(\text{–}, S_2)), \\
(T_2, \text{unlock}(B, S_2)), \\
(T_1, \text{lock}_X(B, S_2)), \\
(T_1, \text{write}_\text{object}(A, S_2), T_1\text{’s A}_{\text{bal}} - 50), \\
(T_1, \text{write}_\text{object}(B, S_2), T_1\text{’s B}_{\text{bal}} + 50), \\
(T_1, \text{commit}_\text{object}(\text{–}, S_2)), \\
(T_1, \text{unlock}(A, S_2)), \\
(T_1, \text{unlock}(B, S_2)), \\
(T_2, \text{unlock}(C, S_2)) \rangle
\]

**Distributed deadlock detection** (S&G §18.5.2): A database may be viewed as a collection of single-unit reusable resources.

Also, the states of a database system are expedient.

Therefore, it is possible to use wait-for graphs (p. 124) to detect deadlock.

When used in a transaction-oriented system, the graphs are called *transaction wait-for graphs* (TWFGs).

These graphs are drawn just the same as any other wait-for graphs, except for the connections between nodes.

**Outline for Lecture 27**

I. Distributed deadlock detection
   A. Transaction wait-for graphs
   B. Centralized approach
   C. Hierarchical approach
   D. Fully distributed approach

II. Distributed deadlock prevention
   A. Timestamp ordering approach
   B. Wait/die
   C. Wound/wait

III. Byzantine generals

We assume that all computation at remote sites is done by “slave” processes at those sites, called *remote agents*.

- When a transaction requests access to resources at a remote site, a *remote agent* is created.
• The remote agent performs the accesses,
• and disappears when the remote access is complete.

The original agent communicates with a remote agent (represented by a box) via input and output ports, which connect the sites.

A communication link, represented by a dashed line, connects the input and output ports.

This model can be applied to both—

• a system of transactions, which are relatively short lived, and
• a system of processes, which are relatively long lived.

There are three approaches to maintaining a global (union) wait-for graph:

• The centralized approach. The global wait–for graph is maintained in a single process, called the deadlock-detection coordinator.

• The hierarchical approach. The global wait–for graph is maintained distributed over a tree-structured set of controllers.

• The fully distributed approach. Each site maintains a part of the global wait-for graph.

The centralized approach: The global wait-for graph may be constructed in various ways, including the following one:

A process notifies the coordinator whenever it inserts or deletes an edge from its local graph. The coordinator uses the changes to update the global wait-for graph.
But—

- False cycles may occur if messages arrive at the coordinator out of sequence.

The diagram below shows local wait-for graphs at sites A and B. Suppose that—
- $p_2$ releases the resource it is holding at site A.
- $p_2$ requests a resource held by $p_3$ at site B.

If the second message arrives at the coordinator before the first, a false deadlock is sensed.

Note that the problem occurs because transaction $p_2$ requests one resource after having released another resource.

Thus, this problem cannot arise if two-phase locking is used.

But, in an operating system, two-phase locking cannot always be used.

So, some attempt needs to be made to distinguish real deadlocks from phantom deadlocks. (Judging by how long the deadlock has persisted may work, but not all the time.)

- With centralized detection, there is a high volume of messages between the local sites and the central site.

One approach is to report the existence of a path only after it has persisted for some time (but report deletion right away).
Why is this helpful?

- 
- 
- What is a third problem with centralized detection? 

*The hierarchical approach (S&G §18.5.2.2):* Instead of a single controller, there is a tree of controllers.

Each non-leaf controller maintains “relevant” information from the controllers below it.

Let $A$, $B$, and $C$ be controllers such that $C$ is the lowest common ancestor of $A$ and $B$.

If $p_i$ appears in the local wait-for graphs of controllers $A$ and $B$, it must also appear in the wait-for graph of

- controller $C$,
- every controller on the path from $C$ to $A$, and
- every controller on the path from $C$ to $B$.

In other words, if $p_i$ and $p_j$ are in the wait-for graph of a controller $D$ and there is a path from $p_i$ to $p_j$ at any controller, then the edge $(p_i, p_j)$ must also be in the graph of controller $D$.

(In the diagram below, we omit the input and output ports and instead show the state of the controller.)
Hierarchical detection greatly reduces the cost of constructing the wait-for graph.

However, phantom deadlocks can still be detected.

**Fully distributed approach:** The algorithm described is Maekawa’s periodic (not continuous) deadlock-detection algorithm.

Properties of the algorithm:
- Do not detect phantom deadlocks (except where a transaction may be aborted for some reason; e.g., resolving another deadlock).
- Only one site will attempt to resolve any given deadlock.
- Deadlock is detected on average with $O(n \log n)$ messages.

The algorithm makes use of sequence numbers, which are assigned by some sort of distributed clock mechanism, like that covered in Lecture 24.

Upon creation, each transaction is assigned an ID number of the form

$$(Q_s, S)$$

where $S$ is the site name and $Q_s$ is the value of the local clock.

- Every time a transaction is assigned a number, the clock is incremented.
- Also, whenever site $S$ receives a message from site $T$, it updates its counter to—

$$Q_s := \max(Q_s, Q_T) + 1$$

We will call the $Q_s$ in the $(Q_s, S)$ pair the *sequence number* of the transaction.

As shown in the diagram two pages hence, original and remote agents and input/output ports have identifiers of specific types.

- The original agent for a transaction is represented by a circle node.
  Its identifier has the form

$$((i, S), 0)$$
• $S$ is the site name.
  • $i$ is set equal to $Q_s$ when the transaction is created.
  • “0” denotes an original agent.

• A remote agent for a transaction is represented by a square node.
  Its identifier has the form
  \[(i, S), g, R\]
  • $S$ is the name of the site where the original agent is located.
  • $g$ is the copy number of the remote agent. (The first remote agent created gets $g = 1$, the second, $g = 2$, etc.)
  • $R$ is the name of the site where the remote agent is located.

Also, we assume that the agent only exists as long as the requested access is in progress.
If another agent is later created by the same transaction at the same remote site, it will get a different copy number.

• An output port created by transaction \[(i, S), 0\] for an access at site $T$ is denoted
  \[\langle (i, S), g, S, T \rangle\]
  and the corresponding input port at site $T$ is denoted
  \[\langle (i, S), g, T, S \rangle\]

  In both cases, $g$ is the copy number of the remote agent.

Each path $p$ in the network has an associated sequence number $\text{highest}(p)$ = the ID of the transaction with the highest sequence number in path $p$ (excluding agents for remote transactions).

In the example on the next page,

• if $p$ is the path from $\langle (6, C), 1, C, A \rangle$ to $\langle (2, A), 1, A, B \rangle$, then
  \[\text{highest}(p) = (2, A)\].

• if $r$ is the path from $\langle (4, B), 1, C, B \rangle$ to $\langle (6, C), 1, C, A \rangle$, then
  \[\text{highest}(r) = (6, C)\].
Sites communicate via deadlock-detection messages, which contain—

- the name of the site that originated the message,
- the message sequence number (set equal to $\text{highest}(\text{local path at originating node})$), and
- a description of the global path from the originating site to the current site.

This description is extended by each site before it is passed on.

Since detection messages are identical only when all three items match, messages that traverse different paths will be different.

The deadlock-detection algorithm:
Periodically, each site determines if it has a local I/O path and, if so, sends a detection message to the site named at the output port.

If a deadlock-detection message is received from another site,

- the receiving site extends the information, if possible, with local I/O paths, and
- sends the detection message to the next site.

Rules 1 and 2, below, tell how the algorithm works.

Each rule is obeyed while no agents are executing, so the state of the wait-for graph does not change while it is being analyzed.
**Rule 1:** Deadlock detection is initiated at site A.

- The site first resolves any local deadlock.
- For each path $p$ that leads through an output port (an I/O path),
  - A detection message
    $$ (A, \text{highest}(p), p) $$
    is generated and sent to the site specified by the output port on $p$.

**Rule 2:** A detection message is received at site B.

**Rule 2.1:** The node at site B is not the originating node of the message.

- If site B has already contributed to the I/O path in this message, it discards it.
- For each path that will extend the path in the message to another site,
  - if message’s sequence number $> \text{highest}(p)$,
    » global path info is extended with $p$ and message is passed on to another node
  - else, if no such path exists, the message is discarded.

**Rule 2.2:** The node at site B is the originating node of the message.

- Site B decides: Does the local I/O path that was used to originate the message still exist?
  - No $\Rightarrow$ message is discarded.
  - Yes $\Rightarrow$ global deadlock is detected. The local deadlock detector selects the highest sequence-numbered transaction on the local path as victim.

**Proof that the algorithm works:** We will show that—

- All real deadlocks are detected by this algorithm, and
- any deadlock that this algorithm detects is real.
**Theorem:** Using this algorithm, any global deadlock will be detected by some site.

**Proof:** A site generates a deadlock-detection message for each local I/O path.

Messages are passed on to other sites whenever possible, and their global path information is extended.

Detection messages are totally ordered according to their unique sequence numbers (message numbers are unique ← transaction sequence numbers are unique).

Only message whose sequence # = highest sequence # of any transaction on its cycle will return to its originating site.

This message identifies the global cycle.

**Theorem:** Any global cycle detected by this algorithm identifies a real global deadlock.

**Proof:** If a detected deadlock were phantom, then a message returned to its originating site despite the fact that the detected cycle never existed.

Hence, the message must have been passed over at least 1 I/O path that no longer exists.

Two cases are possible:

*Case 1:* A completion signal was returned, eliminating an I/O path.

Assume the path was eliminated by removing the arc from node $S$ to node $T$.

- The detection message is sent from $S$ to $T$ (forward).
- The completion signal is sent from $T$ to $S$ (backward).

Three subcases are possible.

a. Before the detection message is sent, the completion signal is received at site $S$.

Then (since agents don’t interfere with the detection algorithm), the I/O path is broken at site $S$ before the detection message is sent.

Thus, the detection message would be discarded at site $S$.

b. Before the completion signal is sent, the detection message is received at site $T$. 
In this case, the detection message would be discarded at the site where the input port was destroyed.

c. The completion signal and the detection message are simultaneously in transit.

In this case, the remote agent (and the input port) that the detection message is heading for have been destroyed.

Thus, the message will be discarded at site $T$.

Even if a new port is created by a new request from $T$ to $S$, the existing message will have the wrong copy number, and thus will not match.

*Case 2:* The cycle is broken by the resolution of another deadlock.

**Distributed deadlock prevention** (S&G §18.5.1): The deadlock-prevention algorithms for a uniprocessor system can be modified for a distributed system.

*The timestamp ordering approach:* One deadlock-prevention technique is the following:

- Each process is assigned a unique priority number.
- A higher-priority process can preempt resources held by a lower-priority process.

What is wrong with this scheme?

Suppose that priorities are based on the *age* of a process. An older process (with a smaller time-stamp value) has a higher priority.
Two deadlock-prevention schemes are possible:

- *Wait/die* scheme:
  - If an older process requests resources held by a younger process, it (the older) waits.
  - If a younger process requests resources held by an older process, it (the younger) is rolled back.

- *Wound/wait* scheme:
  - If an older process requests a resource held by a younger, it preempts the resource.
  - If a younger process requests a resource held by an older, it waits.

It is easy to see that both these schemes avoid starvation. Why?

*Differences between the schemes:*  
Wait/die → as a process ages, it waits more, gets rolled back less.

Wound/wait → fewer rollbacks.

- In the wound/wait scheme, if an older process requests a resource held by a younger process, the younger process is rolled back once; next time it waits.
- In the wait/die scheme, it may be rolled back several times, if the older process continues to hold the resource.

**Byzantine generals** (*S&G §18.7.2*): Suppose we need to have a transaction commit in more than one location.

How can we be sure that the commit “takes” at all sites? (The sites may be widely dispersed, and errors can occur in transmission.)
The Byzantine Generals problem: Several divisions of the Byzantine army surround an enemy camp. The Byzantine generals must reach an agreement on whether to attack the enemy at dawn.

However, the generals can only communicate with each other by using messengers who run from camp to camp. There are two problems:

- Messengers may get caught by the enemy.
- Generals may be traitors, trying to prevent the loyal generals from reaching agreement.

Detection of unreliable communications: A time-out scheme can be used. If another process does not respond to a message in a certain amount of time, it is assumed to have failed.

Detection of faulty processes: Consider a system composed of $n$ processes, of which $m$ are faulty.

Each non-faulty process $p_i$ has some private value $V_i$.

Each non-faulty process $p_i$ needs to construct a vector $X_i$ such that—

- if $p_i$ is a non-faulty process, then $X_i(i) = V_i$.
- if $p_i$ and $p_j$ are both non-faulty processes, then $X_i = X_j$.

A correct solution to this problem requires that $n \geq 3m + 1$.

Example: Consider the case of $m = 1$ and $n = 4$.

1. Each process sends its private value $V_i$ to the other three processes.

2. Each process sends the messages it has received in the first round to all other processes.

3. Each non-faulty process constructs its vectors $X_i$ as follows:

- $X_i(i) = V_i$.
- For $j \neq i$, $X_i(j) = $ the majority of the values received for $p_j$ (if at least two processes agree). If no majority exists, $X_i(j) = \text{NIL}$. 
Problem 1. (25 points) Consider the following process arrival list:

<table>
<thead>
<tr>
<th>Name</th>
<th>Arrival Time</th>
<th>Service Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>D</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>23</td>
<td>6</td>
</tr>
</tbody>
</table>

Consider these scheduling disciplines:

(a) First-come first-served (FCFS)  
(b) Shortest-job next (SJN)  
(c) Shortest remaining time (SRT)  
(d) Round-robin (RR), quantum = 1  
(e) Round-robin (RR), quantum = 5

Draw a Gantt chart (time line) showing which process is executing over time and calculate the turnaround time and waiting time for each process.

In SRT, if a job arrives whose service time is equal to the remaining service time of the job currently being served, the current job is not interrupted.

In RR, if the queue is empty, and a process arrives at the same time a quantum finishes, the running process is preempted and the new arrival executes. When a process gets interrupted by a new arrival, it goes to the end of the ready queue. For example—

```
<table>
<thead>
<tr>
<th></th>
<th>B arrives</th>
<th>C arrives</th>
</tr>
</thead>
<tbody>
<tr>
<td>A/B</td>
<td></td>
<td>A/B</td>
</tr>
<tr>
<td>A/B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A/B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Solution 1. In the Gantt charts below, I have shown which processes are in the queue, and in which order. If there are processes in the queue for a particular time period, they are shown above the process that executes during that time period, in the order that they occupy in the queue, with the next process to execute at the bottom. In the selfish round-robin diagram, only the round-robin queue of accepted jobs is shown.
(a) FCFS.

<table>
<thead>
<tr>
<th>Name</th>
<th>Response time</th>
<th>Waiting time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>E</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>F</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>G</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Avg.</td>
<td>9</td>
<td>4.86</td>
</tr>
</tbody>
</table>

(b) Shortest-job next.

<table>
<thead>
<tr>
<th>Name</th>
<th>Response time</th>
<th>Waiting time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>19</td>
<td>11</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Avg.</td>
<td>6.43</td>
<td>2.29</td>
</tr>
</tbody>
</table>

(c) SRT.
<table>
<thead>
<tr>
<th>Name</th>
<th>Response time</th>
<th>Waiting time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>19</td>
<td>11</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Avg.</td>
<td>6.29</td>
<td>2.14</td>
</tr>
</tbody>
</table>

In the Gantt charts below, I have shown which processes are in the round-robin queue, and in which order. If there are processes in the queue for a particular time period, they are shown above the process that executes during that time period, in the order that they occupy in the queue, with the next process to execute at the bottom. In the selfish round-robin diagram, only the round-robin queue of accepted jobs is shown.

(d) RR with quantum = 1.

![Gantt chart](image)

<table>
<thead>
<tr>
<th>Name</th>
<th>Response time</th>
<th>Waiting time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>C</td>
<td>19</td>
<td>11</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>F</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>G</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Avg.</td>
<td>9</td>
<td>4.86</td>
</tr>
</tbody>
</table>

(e) RR, with quantum = 5.

![Gantt chart](image)

<table>
<thead>
<tr>
<th>Name</th>
<th>Response time</th>
<th>Waiting time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>19</td>
<td>11</td>
</tr>
<tr>
<td>D</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>E</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>F</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>G</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Avg.</td>
<td>8.57</td>
<td>4.43</td>
</tr>
</tbody>
</table>
Problem 2 [15 points] (Silberschatz & Galvin, 5.4, slightly modified) Suppose the following jobs arrive for processing at the times indicated. Each job will run the listed amount of time. What is the average turnaround time for these jobs? Use non-preemptive scheduling and base all decisions on the information you have at the time the decision must be made.

<table>
<thead>
<tr>
<th>Job</th>
<th>Arrival Time</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.0</td>
<td>8</td>
</tr>
<tr>
<td>B</td>
<td>0.5</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>1.0</td>
<td>1</td>
</tr>
</tbody>
</table>

(a) FCFS

Solution:

```
<table>
<thead>
<tr>
<th></th>
<th>Arr</th>
<th>Compl</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>C</td>
<td>10</td>
<td>13</td>
</tr>
</tbody>
</table>
```

Average turnaround time = \( \frac{8 - 0 + 12 - 0.5 + 13 - 1}{3} = \frac{31.5}{3} = 10.5 \)

(b) SJF

Solution:

```
<table>
<thead>
<tr>
<th></th>
<th>Arr</th>
<th>Compl</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
<td>9</td>
</tr>
</tbody>
</table>
```

Average turnaround time = \( \frac{8 - 0 + 13 - 0.5 + 9 - 1}{3} = \frac{28.5}{3} = 9.5 \)

(c) SJF is supposed to improve performance, but notice that we choose to run job 1 at time 0 because we did not know that two shorter jobs would arrive shortly. Compute average turnaround time if the CPU is left idle for the first 1 time unit and then SJF scheduling is used. Remember that job 1 and job 2 are waiting during this idle time so their waiting time may increase. This algorithm could be known as future-knowledge scheduling.

Solution:

```
<table>
<thead>
<tr>
<th></th>
<th>Arr</th>
<th>Compl</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>9</td>
</tr>
</tbody>
</table>
```
Average turnaround time = \[
\begin{align*}
14 & - 0 \\
6 & - 0.5 \\
2 & - 1 \\
\end{align*}
\]
\[
20.5 \div 3 = 6.67
\]

**Problem 3.** *(30 points)* Use the following process information for parts (a) – (c).

<table>
<thead>
<tr>
<th>Process number</th>
<th>Arrival time</th>
<th>Execution time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 ms.</td>
<td>5 ms.</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3 ms.</td>
<td>7 ms.</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4 ms.</td>
<td>3 ms.</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4 ms.</td>
<td>8 ms.</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5 ms.</td>
<td>6 ms.</td>
<td>4</td>
</tr>
</tbody>
</table>

(a) Would a preemptive algorithm produce a larger, smaller, or same average turnaround time versus a non-preemptive algorithm over these processes?

**Solution:** Here is the schedule for a non-preemptive priority discipline.

Completion times are 5, 13, 19, 26, and 29. These times sum to 92.

Here is the schedule for a preemptive priority discipline.

Completion times are 12, 18, 24, 26, and 29. These times sum to 109. Thus, a non-preemptive discipline finishes the average job sooner, and hence produces a lower average turnaround time.

(b) Using preemptive priority scheduling, what would the average response time be over these processes?

**Solution:** Response times are 3, 1, 9, 14, and 25, for an average response time of 52 ÷ 5 = 10.4 ms.

(c) Would any scheduling discipline (FCFS, LCFS, SJN, SRT, RR quantum = 8) produce a better average waiting time over this set of processes? If so, what is the scheduling discipline, and what would the average waiting time be?
Solution: Yes, SRT produces the lowest average waiting time.

Waiting times are 0, 1, 3, 11, and 17, for an average waiting time of $32 \div 8 = 4.0$ ms. As can be seen from the above Gantt chart, in this case SRT = SJF, so SJF also produces an optimal waiting time.

(d) Which scheduling algorithms (FCFS, LCFS, SJF, SRT, RR, preemptive priority, non-preemptive priority) could produce process starvation (a process waits forever without getting a chance to execute)?

Solution: SJF, SRT, preemptive priority, and non-preemptive priority can produce starvation if new jobs keep arriving fast enough.

Problem 4. (15 points) For the following processes with given execution times—

(a) draw the Gantt charts for FCFS, RR with $q=1$, RR with $q=3$, and SJF.

(b) give the total execution times for a context switching time of $s = 0.05$ units and for $s = 0.1$ units. Assume there’s no context switch before the first process or after the last.

(c) give the average waiting time for a context switching time of $s = 0$, $s = 0.05$ units and for $s = 0.1$ units. Assume there’s no context switch before the first process or after the last.

<table>
<thead>
<tr>
<th>Process</th>
<th>Exec. time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
</tr>
</tbody>
</table>

Solution 4.

FCFS:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Total execution time $s = 0.05$: 13.15
Total execution time $s = 0.1$: 13.3
Average waiting time $s = 0$: $(0 + 4 + 7 + 12) / 4 = 5.75$
Average waiting time $s = 0.05$: $(0 + 4.05 + 7.1 + 12.15) / 4 = 5.83$
Average waiting time $s = 0.1$: $(0 + 4.1 + 7.2 + 12.3) / 4 = 5.90$

RR ($q = 1$):

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total execution time $s = 0.05$: 13.7
Total execution time $s = 0.1$: 14.2
Average waiting time $s = 0$: $(7 + 6 + 8 + 3) / 4 = 6.00$
Average waiting time $s = 0.05$: $(7.5 + 6.4 + 8.6 + 3.15) / 4 = 6.41$
Average waiting time $s = 0.1$: $(8.0 + 6.8 + 9.2 + 3.3) / 4 = 6.82$
Problem 5. (15 points) Consider a variant of the RR scheduling algorithm where the entries in the ready queue are pointers to the PCBs, and PCBs may contain priority values.

(a) What would be the effect of putting two pointers to the same process in the ready queue?

Solution: If two pointers to the same process are inserted in the ready queue, then the process concerned will be given twice as much CPU time as the other processes. This means that in this scheme, a process can be given a priority over other processes by inserting two pointers to its process control block in the ready queue.

For example, let there be three processes $p_1$, $p_2$, and $p_3$ in the ready queue. For RR with $q = 1$, the schedule is

![Diagram of RR with $q = 1$]

and so on. ($p_1$, $p_2$, and $p_3$ are pointers to the PCB's of the respective processes).

If process $p_1$ has two pointers in the ready queue, before the pointer to $p_2$ and before the pointer to $p_3$, then the schedule looks like this:

![Diagram of RR with two pointers]

(b) How would you modify the basic RR algorithm to achieve the same effect without the duplicate pointers?

Solution: By allowing insertion of more than one pointer to the PCB in the ready queue we are able to allocate more CPU time to processes within a single cycle. This can be achieved without having to introduce duplicate pointers as mentioned above.
A process control block contains information regarding resources required by a process for execution on the CPU. To allow more time to be given to a process on the CPU, additional information regarding the priority of the process should be stored in the PCB. This can be in the form of a priority number. The higher the priority number, the greater the frequency with which the process will be scheduled in one cycle. So, based on the priority number in the PCB the particular process segment is re-positioned in the ready queue repeatedly to satisfy its priority requirements in the PCB.
This was a 90-minute open-book test. You were to answer five of the six questions. Each question is worth 20 points. If you answer all six questions, your five highest scores counted.

Question 1. (5 points each) The following question asks you to determine which CPU scheduling strategies produce the exact same schedule under certain conditions. As your answer, you should list the set(s) of scheduling strategies that are guaranteed to produce the same results under the given conditions. You do not need to provide a reason, but if you do so, it may help your score if you do not get the whole answer right.

The scheduling strategies you should consider are — FCFS  RR
LCFS  PS
(For multilevel priority queues, assume that all queues SJN Priority disciplines are FCFS, except for the last, which is RR.) SRT  Multilevel priority queues

Example 1. Condition: Each job finishes before the next arrives, so there is never more than one job in the system at a time. Answer: In this case, all disciplines perform the same (because each job starts as soon as it arrives, and executes continuously till it finishes).

Example 2. Condition: Jobs arrive in the order of increasing execution time, so that each job that arrives is longer than all that have preceded it. Answer: FCFS = SJN (because both these strategies are guaranteed to produce the same schedule under these conditions).

(a) Condition: Jobs arrive in order of decreasing execution time.
Answer: LCFS = SJN.

(b) Condition: Besides the job that is executing, no more than one job at a time is ever waiting.
Answer: FCFS = LCFS = SJN = Priority.

(c) Condition: All jobs take less than one RR quantum to execute. (Assume 1 quantum for RR = the smallest quantum in multilevel priority queues.)
Answer: FCFS = RR = MPQ.

(d) Condition: Jobs arrive in order of decreasing priority.
Answer: FCFS = Priority.

A common error on several parts of this question was assuming more about the jobs than was given, e.g., assuming that no more than one job will arrive during the execution of another job.
Question 2. (a) (5 points) The message operations send and receive may be used for interprocess synchronization instead of P and V operations on semaphores. Explain how this can be done, and tell how the mailboxes should be initialized.

Answer: A mailbox can be used for synchronization if blocking (or busy-waiting) is done when a receive is performed on an empty mailbox. In this case, P operations are replaced by receives, and V operations are replaced by sends. The mailbox should be initialized to contain one message, so that the first process performing a receive can go ahead and enter the critical section, while subsequent processes are forced to wait until the first process performs a send.

(b) (12 points) Consider the code for resource allocation from p. 178 of the lecture notes. It is shown below. Modify the code (by crossing out and rewriting) to replace all P, V, enqueue, and dequeue operations by send and receive. Assume indirect naming is used, and that all mailboxes are large enough that they never become full. For each receive operation, explicitly state what kind of “special action” you are assuming in case of an empty mailbox. Hint: Should the receiver be blocked?

Answer: Code that has been changed is underlined in the program below. The “special action” is given as the third parameter to each receive operation; either the receiver is blocked, or a “mailbox empty” code is returned.

```plaintext
procedure request(i : process; j : ResType);
begin
    receive(mutex, dummy, block_receiver);
    if avail[j] = 0 then
        begin
            send(res[j], i); { Enqueue process i to wait for resource j}
            blocked := true;
        end
    else
        avail[j] := avail[j] – 1;
        send(mutex, dummy);
        if blocked then receive(priv_sem[i], dummy, block_receiver);
    end;
end;

procedure release(i : process; j : ResType);
begin
    receive(mutex, dummy, block_receiver);
    avail[j] := avail[j] + 1;
    if receive(res[j], i, special_code) ≠ empty
        {i.e., if a process k is waiting for resource j} then
        begin
            avail[j] := avail[j] – 1;
            send(priv_sem[i], dummy);
        end;
    send(mutex, dummy);
end;
```

(c) (3 points) Should the mailboxes be “initialized” to contain any messages?

Answer: The mutex mailbox should be initialized to contain one message, so that the first process to receive from it can proceed into the critical section. The other mailboxes should be empty after initialization.
Question 3. (4 points each) Here is an Ada implementation of readers and writers:

```ada
task readers_and_writers is
  procedure reader (readvalue: out integer);
  entry writer (writevalue: in integer);
end readers_and_writers;

task body readers_and_writers is
  sharedvariable: integer := 0;
  entry beginreading;
  entry finishedreading;

  procedure reader (readvalue: out integer) is
    beginreading;
    readvalue := sharedvariable;
    finishedreading;
  end;

  begin
    accept writer (writevalue: integer) do
      sharedvariable := writevalue;
    end;

    loop
      select
        accept beginreading;
        readers := readers + 1;
      or
        accept finishedreading;
        readers := readers - 1;
      or
        when readers = 0 =>
          †
          accept writer(writevalue: integer) do
            sharedvariable := writevalue;
          end;
      end select;
    end loop;
  end readers_and_writers;
```

(a) Why are there two "accept writer"s in the code, and only one accept for the read procedures?

**Answer:** The shared information needs to be written before it is read. That is the reason for the first one. The second one is used for all requests except the first.

(b) Under what precise conditions (involving this process and perhaps other processes) is the daggered (†) line executed?

**Answer:** If readers = 0 and another process executes `readers_and_writers.writer(writevalue)`.

(c) Notice that reader is a `procedure`, rather than an `entry`. What unnecessary restriction would be imposed if it were an `entry`?

**Answer:** Only one reader could execute at a time (since there is only one `readers_and_writers` task and it can only accept one `entry` at a time).

(d) Why is this code subject to starvation?

**Answer:** There is no provision to keep readers from continuing to arrive, which would prevent readers from ever becoming 0. Then the writer entry could never be accepted.

(e) What, if anything, would be the impact of adding another alternative

```
o
  or
  delay(10);
```

right before the `end select`?

**Answer:** It would cause a slight amount of busy-waiting. The task would cycle around the loop every 10 seconds even if there were no work to do. Without the `delay`, there would be at least one open alternative (because neither `beginreading` nor `finishedreading` have an associated `when` clause). If no rendezvous were possible, the task would wait until one became available (without any busy waiting).
Question 4. Here is a solution to the producer/consumer problem using sequencers and eventcounts:

Shared Variables

\textbf{var} \quad \text{Pticket}, \text{Cticket}: \text{sequencer};
                  \quad \text{ln, Out}: \text{eventcount};
                  \quad \text{buffer: array}[0..N-1]\ of \text{message};

\textbf{Producer} i \quad \text{Consumer} j

-- A variable \( t \), local to each producer. -- A variable \( u \), local to each consumer.
\textbf{var} \quad \text{t: integer}; \quad \textbf{var} \quad \text{u: integer};

\textbf{begin} \quad \textbf{begin}
  \textbf{loop} \quad \textbf{loop}
    \ldots \quad \ldots
    \text{Create a new message} \( m \);
    \text{One producer at a time}
    \text{\( t := \text{ticket}(\text{Pticket}) \);} \quad \text{\( u := \text{ticket}(\text{Cticket}) \);}
    \text{Await an empty cell}
    \text{\( \text{await}(\text{ln}, t); \quad \text{await}(\text{Out}, u); \)}
    \text{Signal a full buffer}
    \text{\( \text{await}(\text{Out}, t + N + 1); \quad \text{await}(\text{ln}, u + 1); \)}
    \text{and allow other producers}
    \text{\( \text{buffer}[t \mod N] := m; \quad \text{buffer}[u \mod N]; \)}
    \text{\( \text{advance}(\text{ln}); \quad \text{advance}(\text{Out}); \)}
    \ldots \quad \ldots
  \textbf{end} \{\text{loop}\}; \quad \textbf{end} \{\text{loop}\};
\textbf{end}; \quad \textbf{end};

(a) (4 points) Suppose a consumer is the first process to execute (before any producers). Where will it wait, and when will it continue?

\textbf{Answer:} It will wait at the \textbf{await}(\text{ln}, u + 1) until the first producer finishes, \textbf{advancing} \text{ln}.

(b) (4 points) Explain how it is possible for both producers and consumers to execute concurrently, given that they are both updating the same queue. Why couldn't this cause a race condition?

\textbf{Answer:} The producer changes only the \text{in} pointer, and the consumer changes only the \text{out} pointer.

(c) (3 points) Must \textbf{advance} be executed indivisibly in order for this program to work?

\textbf{Answer:} No, because only one producer and one consumer is allowed to execute at a time.

(d) (3 points) The following is stated in an operating system text: “If concurrent executions of a producer and a consumer are not necessary, we can totally order all the producers and consumers by using just one sequencer, and each process needs to execute only a single \textbf{await}.”

Suppose we simply replace \text{Cticket} and \text{Pticket} by a sequencer \text{OneTicket} and eliminate the first \textbf{await} in each process. Why is this not a correct solution?

\textbf{Answer:} The first \textbf{await} is there to enforce mutual exclusion, so mutual exclusion is not preserved. However, that is not what really causes the problem in this case. The producer and consumer are both taking tickets from the same sequencer. Consequently, they can never get the same ticket numbers. But they use their tickets to decide what buffer portions to operate on.
Consequently, the consumer will only consume garbage that has never really been “produced,” and the information produced by the producer will never be consumed by the consumer.

(e) (6 points) Assuming that a producer in the code above is executing a procedure called *produce* and a consumer is executing a procedure called *consume*, give a path expression that imposes the same constraints on the execution of producers and consumers as the code above.

*Answer:* path $N: (\text{produce}; \text{consume}), (1: \text{produce}, 1: \text{consume})$ end;

**Question 5.** Consider this system of processes,

$$P = \{ P_1, \ldots, P_6 \}$$

$$\Rightarrow = \{ (P_1, P_2), (P_1, P_3), (P_2, P_4), (P_3, P_4), (P_4, P_5),$$

$$\quad (P_4, P_6) \}$$

(a) (4 points) Draw the precedence graph of this system.

*Answer:*

```
   P_1
    |   |
    |   |
  P_2  P_3
    |   |
  P_4
    |   |
  P_5  P_6
```

(b) (16 points for parts (b) and (c))

Decide if the system is determinate. Explain how you can tell.

*Answer:* Given the above precedence constraints, the only opportunities for parallel execution are $P_2$ and $P_3$, and $P_5$ and $P_6$. Both of these pairs satisfy Bernstein’s conditions, since

$$R(P_2) = \{ a \} \text{ does not intersect } W(P_3) = \{ b, c \}$$

$$R(P_3) = \{ c \} \text{ does not intersect } W(P_2) = \{ e \}$$

$$W(P_2) = \{ a \} \text{ does not intersect } W(P_3) = \{ b, c \}$$

$$R(P_5) = \{ a, c \} \text{ does not intersect } W(P_6) = \{ b \}$$

$$R(P_6) = \{ c, d \} \text{ does not intersect } W(P_5) = \{ e \}$$

$$W(P_5) = \{ e \} \text{ does not intersect } W(P_6) = \{ b \}$$

For all other pairs of processes, one is a predecessor of the other. Therefore, the system is determinate.

(c) If the system is determinate, find the maximally parallel system equivalent to the original. (Drawing the precedence graph will be sufficient.)

*Answer:* We need to determine which pairs of processes satisfy Bernstein’s conditions. The following sets do:

$$\{P_1, P_2\}$$

$$\{P_2, P_3\}, \{P_2, P_4\}, \{P_2, P_6\}, \{P_4, P_5\}, \{P_5, P_6\}$$

*Answer:*

```
   P_1
    |
    |
    |
  P_2
    |
    |
  P_3
    |
  P_4
    |
  P_5
    |
  P_6
```
Question 6. (a) (4 points) When or-synchronization is used in a program, it may be necessary to check more than one event queue when an event occurs. Explain why.

Answer: Because a process waiting for an event may be queued on either the queue for the specific event, or a shared queue for multiple events involved in a wait-or statement.

(b) (4 points) Suppose that the following were the only wait statements in a program. (Here, wait means the same thing as \( P \) for a semaphore.) How many queues would be needed? For partial credit, explain your answer!

\[
\begin{align*}
\text{wait}(a); & \quad \text{wait}(d); & \quad \text{wait-or}(a, b, D); \\
\text{wait}(b); & \quad \text{wait}(a); & \quad \text{wait-or}(d, e, E); \\
\text{wait-or}(a, b, A); & \quad \text{wait-or}(a, b, c, B); \\
\text{wait}(c); & \quad \text{wait-or}(a, c, C);
\end{align*}
\]

Answer: A separate queue is needed for each set of condition variables waited on by a wait or wait-or statement. We need—

- one queue for event \( a \)
- one queue for event \( b \)
- one queue for event \( c \)
- one queue for event \( d \)
- one queue for events \( a \) and \( b \)
- one queue for events \( a, b, \) and \( c \)
- one queue for events \( a \) and \( c \)
- one queue for events \( d \) and \( c \)

for a total of eight queues.

(c) (5 points) How is this wait-or approach better than the “naïve implementation” (p. 82 of lecture notes) that also uses multiple queues for each event?

Answer: In the naïve approach, it may be necessary to remove a process from the middle of several queues when an event occurs. In the wait-or approach, it is only necessary to remove a process from the end of one queue. Sometimes it will be necessary to check more than one queue (if the first queue checked is empty), but this is even more true of the naïve approach, which has to check the event queue for each of the events being waited for whenever an event occurs which might require a process to be removed from its queue.

(d) (4 points) The lecture notes give one possible implementation of wait-and (which they call \( SP \)). \( SP \) can’t be implemented by just decrementing all of the semaphores in a critical section. Explain why the implementation at the right will not work.

\[
\begin{align*}
\text{SP}(s_1, s_2, \ldots, s_n); & \quad \text{begin} \\
& \quad \text{P} (\text{mutex}); \\
& \quad \text{P} (s_1); \\
& \quad \text{P} (s_2); \\
& \quad \vdots \\
& \quad \text{P} (s_n); \\
& \quad \text{V} (\text{mutex}); \\
\text{end};
\end{align*}
\]

Answer: If one of the semaphores already has the value 0, a process will get hung up in the middle of the critical section, and not allow any other process to get a crack at decrementing the other variables. No other process can even get past the mutex, so no other \( SP \) anywhere in the program can proceed.

(e) (3 points) How about implementing \( SV \) in the analogous way to the \( SP \) of part (d)—as a sequence of \( V \) operations. Would this work?

Answer: Yes. It would work even without the mutex.

<table>
<thead>
<tr>
<th>Grade summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>High 96</td>
</tr>
<tr>
<td>Mean 70.46</td>
</tr>
<tr>
<td>Low 30</td>
</tr>
<tr>
<td>Std. Dev. 15.65</td>
</tr>
<tr>
<td>Quartiles (61, 76, 83)</td>
</tr>
</tbody>
</table>