Virtual Memory Systems

- Virtual memory is another level in the memory hierarchy.
- All general-purpose processors today employ virtual memory.

In our quest to make the fastest possible computer system, we sometimes must acquiesce to the programmers and add function that slows the system down.

- We add cache memory to a computer system solely for the purpose of speeding up the processing rate.
- We add virtual memory to a computer system as a convenience for the programmer. Virtual memory slows down the processing rate.
Multiple address spaces

Virtual memory allows *multiple large linear address spaces* to concurrently exist in a single real memory by breaking each address space into fixed sized *pages*, and keeping only a subset of the pages in real memory. The remaining pages of each virtual memory space may or may not be defined and allocated space in the system. If they are defined, they are allocated space in a *backing store*, usually fixed disk.

The job of the computer system architect is to *minimize the overhead* imposed by the virtual memory design.

The subject of virtual memory is covered in detail in CSC 501, *Operating System Principles*. We will consider here only those aspects of virtual memory that affect system performance and the physical memory system design.

What’s the problem with Virtual Memory?

♦ Cache memory deals with *physical memory addresses*.

♦ The processor actually generates *virtual memory addresses*.

♦ Before we can begin our search of the cache, we must *translate the virtual address into a physical address*. 
Example Virtual Memory System

- 16 Megabytes of real memory (24-bit address)
- 4 Gigabytes of virtual memory per address space (32-bit address).
- 4K page size.
Mapping of the virtual addresses to physical addresses

Most virtual memory systems use a two-level lookup to translate virtual addresses into real addresses.

The virtual address is divided into:

♦ A segment table entry (STE) number
♦ A page table entry (PTE) number
♦ A byte displacement into the page

The system uses a **single segment table pointer** to locate (in real memory) the segment table that is currently in use.

Each segment table entry contains:

♦ Whether or not the corresponding page table is valid (defined).
♦ Whether or not it is resident in real memory or paged out.
♦ The address of the page table (real or disk) if it is valid.

Each page table entry contains:

♦ Whether or not the page is valid (defined).
♦ Whether or not it is resident in real memory or paged out.
♦ The address of the page (real or disk) if it is valid.
Example virtual address translation

The following diagram shows the translation of a 32-bit virtual address into a 24-bit real address using a two-level lookup system.

- 1K possible page tables
- 1K pages per page table
- 4K page size
Steps to translate virtual to real

♦ Index into the segment table using the segment table address and the STE number in the virtual address.

♦ If the page table is not valid, generate an address exception interrupt.

♦ If the page table is valid but paged out, generate a page fault interrupt.

♦ Index into the corresponding page table using the page table address in the STE and the PTE number in the virtual address.

♦ If the page is not valid, generate an address exception interrupt.

♦ If the page is valid but paged out, generate a page fault interrupt.

♦ Concatenate the page frame address in the PTE and the byte displacement in the virtual address to form the real address.

We are now ready to see if the data at the real address is in the cache!

BUT, we just had to access the system memory TWICE in order to translate the virtual address into the real address so we can get the data the processor really wants.

Every access to memory is now three accesses to memory:

♦ Access memory to get the segment table entry.

♦ Access memory to get the page table entry.

♦ Access memory to read or write the data requested by the processor.

This does not bode well for the performance of our computer system.
The Translation Lookaside Buffer

We can minimize the performance effects of this two-level lookup with the use of a **translation lookaside buffer** (TLB).

The TLB is simply a **cache of translated addresses**. Whenever we translate a virtual address to a real address, we save the translation in the TLB.

<table>
<thead>
<tr>
<th>Virtual Page</th>
<th>Real Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>23EF0</td>
<td>013</td>
</tr>
<tr>
<td>602D2</td>
<td>2E3</td>
</tr>
<tr>
<td>9A517</td>
<td>800</td>
</tr>
<tr>
<td>01B69</td>
<td>014</td>
</tr>
<tr>
<td>C3459</td>
<td>16B</td>
</tr>
</tbody>
</table>

**We can manage the TLB in the same way as a data cache:**

- Direct map, $n$-way set associative, Fully associative.
- LRU replacement algorithm.
- Split TLB (separate instruction and data TLB’s).

Virtually **ALL** TLB’s today are implemented on-chip with the processor.

- Performance is critical.
- Very few bits (relative to data cache) are needed to implement it.
- Sizes are on the order of 64 to 256 entries.
A flowchart of memory access operation:

Dr. Ed Gehringer shows this flowchart to give an overall idea of the complexity of what goes on in a modern computer system to retrieve a word from memory. Note that this chart assumes that the desired page is resident in main memory – there is no provision for a page fault, and that we do not have a multi-level cache.