Demand Paging

PA 3
Intel System Programming

- Outer Page Table (Page Directory) = 1024 page directory entries in a page directory
  Page Table = 1024 page table entries in a page table
  Page - 4-KB flat address space

- PDBR = Page Directory Base Register (CR3)
  points to the start address of Page Directory (Outer Page Table)

- TLB - lookup in page tables in memory are performed only when the TLBs do not contain the translation information for a requested page.
  invalidate - automatically invalidated any time the CR3 register is loaded.

(Read Intel System Programming API - Chapter 3 (Paging), Chapter 5 (Page Fault interrupt))
From Boot

1. Initialize (zero out the values)
   - backing store - (create data structures)
   - frames - (create data structures)
   - install page fault handler

2. Create new page table for null process:
   - create page directory (outer page table)
   - initialize 1:1 mapping for the first 4096 pages
     - allocate 4 page tables (4x1024 pages)
     - assign each page table entry to the address starting from page number 0 to 1023
   - this page tables should be shared between processes
3. Enable paging
   - set bit 31st of the CR0 register
   - take care that PDBR is set, because subsequent memory address access will be virtual memory addresses

4. Creating new process (eg. main):
   - create page directory (same as with null process)
   - share the first 4096 pages with null process

5. Context switch:
   - every process has separate page directory
   - before ctxsw() load CR3 with the process's PDBR
Using Virtual Memory

1. Allocate pages in backing store
2. Map it to virtual page using xmmmap()
   - for example if you do xmmmap(A, backingstore, 10)
   - then the mapping would be made to consecutive locations in backingstore for
     virtual pages: A, A+1, A+2, ..., A+9
3. Then try accessing the virtual address
4. If the page is not present a Page Fault is generated:
Page Fault

1. Address that caused page fault
   - content of CR2 register

2. Search for the page table entry. Two cases:
   - a). second level page table does not exist
   - b). second level page table exists but the page table entry does not exist
   - How do we know? Use the P flag for page directory/table entry
Page Fault - 2

3. Case a)
   - allocate a frame -> initialize (zero out the page table frame)
   - update the page directory entry with base address of the page table frame
   - Now this case becomes Case (b)
4. Case b)
   - Locate backing store id of the faulted page, the page number in the backing store.
   - Find a free frame to store the page from backing store
     - if found: use the free frame
     - if not found: evict a page frame (Page Replacement Algorithm)
   - Update the page table entry for the page and possibly for evicted page frame
5. Finally: Flush TLB content, by reloading CR3 with page directory address
Virtual address has page table offset as well as page directory offset.

PageTableNumber(31-22) PageNumber(21-12) Offset(11-0)

Page Directory/Table Entry Format
31-12   PFA  page frame address
11-9    Avail available to OS
8       0    must be 0
7       L    PTE -- Must be 0. Dir Entry -- 4MB page
6       D    dirty (PTE only -- documented as undefined in directory entry)
5       A    accessed
4       PCD page cache disable (can't cache data on this page)
3       PWT page write transparent (tell external cache to use write-through strategy for this page)
2       U    user accessible
1       W    writeable
0       P    present
/* user.c - main */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <stdio.h>
#include <paging.h>

void halt();

/*------------------------------------------------------------------------
  main  -- user main program
 *------------------------------------------------------------------------*/
int main() {
  char *addr = (char*)0x40000000; // 1G
  bsd_t bs=1;

  int i = ((unsigned long)addr)>>12;  // the ith page

  kprintf("\nHello World, Xinu lives\n\n");
  kprintf("Calling get_bs for bs: %d npages:200\n",bs);
  get_bs(bs, 200);

  kprintf("Calling xmmap for vpage:%x(%d) bs:%d npages:200\n",i,i,bs);
  if (xmmap(i, bs, 200) == SYSERR) {
    kprintf("xmmap call failed\n");
    return 0;
  }
}
for(i=0;i<2;i++){
    kprintf("Write Operation\n===============\n");
    *addr = 'A'+i;
    addr+=NBPG;
}

kprintf("\nRead Operation\n===============\n"); addr = (char*)0x40000000; //1G
    kprintf("0x%08x: %c\n", addr, *addr);

xmunmap(0x40000000>>12);

return 0;"}
connection 'backend-pc', class 'POWERCYCLE', host 'blade1g3-3'

system running up!
alloc_frame return frame 1024
initialize page tables for null process
alloc_frame return frame 1025
pd[0].base = 1025
alloc_frame return frame 1026
pd[1].base = 1026
alloc_frame return frame 1027
pd[2].base = 1027
alloc_frame return frame 1028
pd[3].base = 1028

PC Xinu (CSC501 1-2009 base) #3 (nshah3@bn17-109.dcs.mcnc.org) Wed Oct 27 16:10:51 EDT 2010

4194304 bytes real mem
113115 bytes Xinu code
clock enabled
alloc_frame return frame 1029
load cr3
enable paging
cr0: 80000011, cr3 400000

Hello World, Xinu lives

Calling get bs for bs: 1 npages:200
200 pages of 1 bs allocated

Calling xmmap for vpage:40000(262144) bs:1 npages:200
map bs1 into process 49:262144
add mapping bs1 to process 49:262144
add mapping bs-1 to process 49: 40000(262144)

add pid 49's mapping of bs 1
Write Operation
========================
#PF in main, cr2:40000000, ec:2
alloc_frame return frame 1030
alloc_frame return frame 1031
map bs1/page: 0 to frame 1031
Write Operation
=================
#PF in main, cr2:40001000, ec:2
alloc_frame return frame 1032
map bs1/page: 1 to frame 1032

Read Operation
===============
0x40000000: A
unmap process 49:262144
put frame mapping bs 1/0
remove frame 1031 from bs 1
put frame mapping bs 1/1
remove frame 1032 from bs 1
delete pid 49's mapping of bs1@262144
delete pid 49's mapping of bs1

All user processes have completed.
purge all the mapping for pid 49
free page tables for 49