CS 201
Computer Systems Programming II – Memory Management

Notes based on robboy’s notes and instructors notes from B&O’s web site: http://csapp.cs.cmu.edu

Building the Address Space

- Load time:
  - Allocate primary memory
  - Adjust addresses in address space
  - Copy address space from secondary to primary memory

Going way back in history

- Limited versatility.

With multiple processes

Issue: Where do you load p_i’s address space into primary memory?
Dynamic Memory Allocation

- Process wants to change the size of its address space
  - Malloc/sbrk
  - Stack growth – temporary variables
- May have to dynamically relocate the program

A system with physical memory only

- Examples:
  - most Cray machines, early PCs, nearly all embedded systems

Some problems with physical memory only

- With multi-tasking, you have to dynamically relocate programs when loading them
- If the stack overflows the area allocated for it, we're in trouble
- The same with the heap
- With swapping, you have to dynamically relocate programs each time they are swapped in
  - Is that even possible? How would you handle locally declared pointers?

VM address translation

- Remember: all processes have their own virtual address space
  - In addition to: memory protection, relocation, swapping, …
A system with virtual memory

- Examples:
  - Workstations, servers, modern PCs, etc.

Address Translation: Hardware converts virtual addresses to physical addresses via a lookup table (page table)

Example

- 32 bit addresses, page size is 4096
- How many bits is the offset into a page?
- A page-aligned address has how many low order zero bits?

Example

- 32 bit addresses, page size is 4096 = 0x1000
- Consider some address: 0x3e80a123
- Low order 12 bits: offset within the page: 0x123
- Address with low order 12 bits masked out: address of the page: 0x3e80a000
- High order 20 bits alone are the page number: 0x3e80a

Previous example continued

- 32 bit addresses, 4K pages
- Page table = 1 page
  - 1 page = 4096 bytes/4 byte entry = 1024 entries
  - 1024 entries x 4096 bytes = 4 MB of virtual memory per page table
- 4 MB X 1024 page tables = 4GB memory space
Clarification

- Page tables and page directories are data structures in memory.
- The O.S. kernel (software) sets them up and manages them.
- The format of the contents is defined by hardware.
- The hardware uses them on every memory reference, to convert a virtual address to a physical address.
- The PDBR tells the hardware where to look.

Implications for memory management

- To allocate memory for a process, now the O.S. doesn't have to manage contiguous blocks of memory.
- All it has to do is find a set of available pages.
  - The pages can be scattered all over the place.
  - The pages are mapped into contiguous virtual memory regions.
  - No more fragmentation.
- Analogous to allocating files on a disk – fixed size blocks.

What this means for linking/loading

- The linker binds programs to absolute addresses.
  - No relocation at load time.
  - No allocation of memory segments at load time.

How ia32 maps virtual to physical addr

- Virtual address
- Physical address

- PDBR
- PDE
- PTE
- PPN
- PPO
Exercise

- Suppose a computer has 16-bit virtual addresses, 16-bit physical addresses, a page size of 64 bytes, and two-level page tables, like a pentium.
  - How many bits is the VPO?
  - How many entries in a page table?
  - How many bits is VPN0?
  - How many bits is VPN1?

Some arithmetic

- VPO of 12 bits → A page is 4096 bytes
- Each PD and PT occupies one page.
- Each PDE and PTE is 32 bits (4 bytes) →
  - Each page directory contains 1024 PDEs
  - Each page table contains 1024 PTEs
  - Each page table points to 1024 pages
- 1024 pages * 4K bytes = 4 MB covered by a page table
- 4 MB * 1024 PDEs = 4 GB memory space covered by a page directory.

ia32 page table structure

- Page directory
  - 1024 4-byte page directory entries (PDEs) that point to page tables
  - one page directory per process
  - page directory must be in memory when its process is running
  - always pointed to by PDBR
- Page tables:
  - 1024 4-byte page table entries (PTEs) that point to pages.
  - page tables can be paged in and out.
ia32 page table structure (continued)

- Page directory
  - A page directory defines the virtual memory mapping for a process
  - Not all PDEs point to a valid PT
    - That is, a process does not necessarily use its entire 4GB memory space
    - The valid PDEs may be sparse in the PD
- Page tables:
  - Valid PTEs may be sparse in the PT also
  - Some pages may be valid but not present

Exercise

- Suppose we had a paging scheme on a computer with 8 bit addressing, a page size of 16 bytes, and 8-bit page table entries.
  - How big a memory space can be expressed with 8 bits?
  - How many bits is a VPO?
  - How many bits is a VPN?
  - How many pages does it take to cover a memory space?
  - How many page tables do you need?
  - How many entries are there in a page directory?

P6 page directory entry (PDE)

<table>
<thead>
<tr>
<th>31 12 11 9 8 7 6 5 4 3 2 1 0</th>
<th>Page table physical base addr</th>
<th>Available</th>
<th>G</th>
<th>PS</th>
<th>A</th>
<th>CD</th>
<th>WT</th>
<th>US</th>
<th>RW</th>
<th>P=1</th>
</tr>
</thead>
</table>

Page table physical base address: 20 most significant bits of physical page table address (forces page tables to be 4KB aligned)
- Available: These bits available for system programmers
- G: global page (don’t evict from TLB on task switch)
- PS: page size 4K (0) or 4M (1)
- A: accessed (set by MMU on reads and writes, cleared by software)
- CD: cache disabled (1) or enabled (0)
- WT: write-through or write-back cache policy for this page table
- US: user or supervisor mode access
- RW: read-only or read-write access
- P: page table is present in memory (1) or not (0)

<table>
<thead>
<tr>
<th>31 0</th>
<th>Available for OS (page table location in secondary storage)</th>
<th>P=0</th>
</tr>
</thead>
</table>

P6 page table entry (PTE)

<table>
<thead>
<tr>
<th>31 12 11 9 8 7 6 5 4 3 2 1 0</th>
<th>Page physical base address</th>
<th>Available</th>
<th>G</th>
<th>D</th>
<th>A</th>
<th>CD</th>
<th>WT</th>
<th>US</th>
<th>RW</th>
<th>P=1</th>
</tr>
</thead>
</table>

Page base address: 20 most significant bits of physical page address (forces pages to be 4 KB aligned)
- Available: available for system programmers
- G: global page (don’t evict from TLB on task switch)
- D: dirty (set by MMU on writes)
- A: accessed (set by MMU on reads and writes)
- CD: cache disabled or enabled
- WT: write-through or write-back cache policy for this page
- US: user/supervisor
- RW: read/write
- P: page is present in physical memory (1) or not (0)

<table>
<thead>
<tr>
<th>31 0</th>
<th>Available for OS (page location in secondary storage)</th>
<th>P=0</th>
</tr>
</thead>
</table>
Questions

- If for every memory reference, we had to do a two-level table lookup, then every memory reference would actually involve three memory references:
  - Page directory, page table, and memory containing data

- Would this be good for performance?
- How can it be optimized?

What is the TLB

- The TLB is an on-chip cache of page mappings
- The TLB converts a virtual page address to a physical page address
- On a TLB hit, the TLB delivers a physical page address
  - Bypasses the page tables entirely
- Even on a TLB miss, the page table entries may be in the memory cache.
P6 TLB

- TLB entry (not all documented, so this is speculative):
  - V: indicates a valid (1) or invalid (0) TLB entry
  - PD: is this entry a PDE (1) or a PTE (0)?
  - tag: disambiguates entries cached in the same set
  - PDE/PTE: page directory or page table entry

- Structure of the data TLB:
  - 16 sets, 4 entries/set

Translating with the P6 TLB

1. Partition VPN into TLBT and TLBI.
2. Is the PTE for VPN cached in set TLBI?
   - Yes: then build physical address.
   - No: then read PTE (and PDE if not cached) from memory and build physical address.

Exercise

- The operating system kernel manages page tables in memory.
- When the kernel modifies a page table entry, what has to happen with the TLB?
- With 32 bit addressing and a 4K page size, why did they decide to use 2-level page translation?
- What size page table would you need with single level page translation?
The Pentium processor family has another paging mode, with 4 megabyte pages.

- How many pages are there in a 4G memory space?
- How many bits are there in a virtual page offset?
- How many bits are there in the page number?

With 4 meg. pages, what kind of scheme would work for address translation?

A disadvantage: Internal fragmentation if you don’t use the whole 4M page.

What’s the advantage?