Announcements

- Project #2 is available on the web
Managing Memory

• **Main memory is big, but what if we run out**
  - use virtual memory
  - keep part of memory on disk
    • bigger than main memory
    • slower than main memory

• **Want to have several programs in memory at once**
  - keeps processor busy while one process waits for I/O
  - need to protect processes from each other
  - have several tasks running at once
    • compiler, editor, debugger
    • word processing, spreadsheet, drawing program

• **Use virtual addresses**
  - look like normal addresses
  - hardware translates them to physical addresses
Advantages of Virtual Addressing

- Can assign non-contiguous regions of physical memory to programs
- A program can only gain access to its mapped pages
- Can have more virtual pages than the size of physical memory
  - pages that are not in memory can be stored on disk
- Every program can start at (virtual) address 0
Paging

- Divide physical memory into fixed sized chunks called *pages*
  - typical pages are 512 bytes to 64k bytes
  - When a process is to be executed, load the pages that *are actually used* into memory
- Have a table to map virtual pages to physical pages
- Consider a 32 bit addresses
  - 4096 byte pages (12 bits for the page)
  - 20 bits for the page number

![Diagram of paging system](Diagram.png)
Problems with Page Tables

- One page table can get very big
  - $2^{20}$ entries (for most programs, most items are empty)
- solution1: use a hierarchy of page tables
Inverted Page Tables

- Solution to the page table size problem
- One entry per page frame of physical memory
  <process-id, page-number>
  - each entry lists process associated with the page and the page number
  - when a memory reference:
    - <process-id, page-number, offset> occurs, the inverted page table is searched (usually with the help of a hashing mechanism)
    - if a match is found in entry \( i \) in the inverted page table, the physical address \( <i, offset> \) is generated
  - The inverted page table does not store information about pages that are not in memory
    - page tables are used to maintain this information
    - page table need only be consulted when a page is brought in from disk
Inverted Page Table Example (PPC)

- Virtual Address
  - Seg: 4
  - Page #: 16
  - Byte: 12

- Segment Registers (per process)
  - 16

- Virtual Segment ID
  - 24

- Hash Function
  - 40

- Page Table Group
  - 8 page table entries

- Page Table
  - (variable size)
  - one per system

- Status bits
  - VS ID (40)
  - Physical page (20)

- Page Table Entry (PTE)

- Main Memory
Faster Mapping from Virtual to Physical Addresses

- need hardware to map between physical and virtual addresses
  - can require multiple memory references
  - this can be slow
- answer: build a cache of these mappings
  - called a translation look-aside buffer (TLB)
  - associative table of virtual to physical mappings
  - typically 16-64 entries

<table>
<thead>
<tr>
<th>Virtual Page</th>
<th>Physical Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>20 bits</td>
<td>20 bits</td>
</tr>
</tbody>
</table>

For Intel x86
Sharing Memory

• Pages can be shared
  – several processes may share the same code or data
  – several pages can be associated with the same page frame
  – given read-only data, sharing is always safe

• when writes occur, decide if processes share data
  – operating systems often implement “copy on write” - pages are shared until a process carries out a write
    • when a shared page is written, a new page frame is allocated
    • writing process owns the modified page
    • all other sharing processes own the original page
  – page could be shared
    • processes use semaphores or other means to coordinate access