Lecture Topics

- **Today**: Memory Management  
  (Stallings, chapter 7.1-7.4)

- **Next**: Virtual Memory  
  (Stallings, chapter 8.1-8.4)

Announcements

- Self-Study Exercise #6
- Project #5 (due 10/18)
- Project #6 (due 10/25)
Memory Management

- Processes must be resident in RAM in order to be executed:
  - Process control block
  - Space for machine language instructions
  - Space for data objects

- OS also must be resident in RAM (set of kernel functions which are called by user processes)
Simplistic Memory Management

- Assume a process must be loaded entirely into main memory in order to execute

- Simple strategies (without virtual memory):
  - fixed partitioning
  - dynamic partitioning
  - simple paging
  - simple segmentation

Simple Paging

- The address space of a process is viewed as a sequence of pages

- Each page is placed in a page frame in physical memory
Simple Paging

- Physical memory is managed as a sequence of page frames (all the same size)
- Each process is viewed as an integer number of pages (same size as page frames)
- Advantage: a process no longer needs a large contiguous block of physical memory; instead it is given a set page frames (which might be scattered around RAM)

OS manages a page table for each process

Page table maps logical addresses to physical addresses
Simple Paging

- A logical address in a process is viewed as a page number and byte offset
  - Assume addresses are 16 bits
  - Assume pages are 1 K (1024 bytes)
  - Addresses: \texttt{PPPPPPPxxxxxxxxxxx}

- OS maintains a page table for each process to map page numbers to frame numbers

Simple Paging

Every logical address must be mapped to a physical address (page number is the index into the page table)
Example

- System uses 1 Kbyte pages
- Process is 2.7 Kbytes long
- OS views process as 3 pages
- Last page is not fully used (internal fragmentation)
- Logical address viewed as page number and byte offset

- OS allocates 3 page frames (one for each page), initializes page table
Simple Paging

OS can manage RAM efficiently

• unnecessary to find large contiguous blocks of RAM – each process subdivided into pages

• all page frames are identical and can be used interchangeably

• processes can be swapped out to disk, swapped back in later – and placed in different page frames

Example

- Assume 15 page frames allocated in RAM for user processes (numbered 0 to 14)

- OS tracks which page frames have not been allocated to any process (free frame list)

- OS allocates frames when a process is created

- OS reclaims frames when process terminates
Example (2)

(a) Fifteen Available Frames

(b) Load Process A

(c) Load Process B

Example (3)

(d) Load Process C

(e) Swap out B

(f) Load Process D
Example (4)

Page tables after Process D is loaded

Example (5)

Configuration after Process D is loaded
Paging: Address Translation

- Pages and page frames must be same size
- Use page size which is a power of 2 (allows logical address to be subdivided easily)
- Logical addresses and physical addresses don't have to be the same size (page number and frame number can have different number of bits)

Assume 15-bit logical addresses and 20-bit physical addresses

Assume 12-bit page displacements
Paging Example

- Logical address has 16 bits:
  - Page: 4 bits to identify page
  - Offset: 12 bits to identify byte within page

- Physical address has 32 bits:
  - Frame: 20 bits to identify frame
  - Offset: 12 bits to identify byte within frame

Assume page 5 is in frame 20CE7

Logical address 5408 maps to physical address 20CE7408:

```
0101010000001000
00100000110011100111010000001000
```

Paging Example (2)
Paging Example (3)

- How many entries in the page table?
  - page number is 4 bits
  - $2^4 = 16$ entries

- How many bytes in one page (and one frame)?
  - page offset is 12 bits
  - $2^{12} = 4096$ bytes

Paging Example (4)

- How many frames in RAM?
  - frame number is 20 bits
  - $2^{20} = 1,048,576$ frames

- How large is RAM?
  - physical address is 32 bits
  - $2^{32} = 4,294,967,296$ bytes = 4 GB
### Paging Example (5)

<table>
<thead>
<tr>
<th>I</th>
<th>V</th>
<th>Frame</th>
<th>I</th>
<th>V</th>
<th>Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>FF641</td>
<td>8</td>
<td>0</td>
<td>0004A</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00014</td>
<td>9</td>
<td>0</td>
<td>00028</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0003A</td>
<td>A</td>
<td>0</td>
<td>00028</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>FF593</td>
<td>B</td>
<td>0</td>
<td>FFF7C</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>FFF7C</td>
<td>C</td>
<td>0</td>
<td>00EA1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>20EC7</td>
<td>D</td>
<td>0</td>
<td>00028</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>00014</td>
<td>E</td>
<td>0</td>
<td>0003A</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>00014</td>
<td>F</td>
<td>0</td>
<td>0003A</td>
</tr>
</tbody>
</table>

### Paging Example (6)

- Consider the entry at index 2:

1  0003A

First and last logical addresses?

- first address: 2000
- last address: 2FFF

First and last physical addresses?

- first address: 0003A000
- last address: 0003AFFF
Paging Example (7)

- Request for logical address **3A14**
  
  Page 3, so check entry at index 3:
  
  1  **FF593**
  
  V = 1, so mapping is valid
  
  physical address **FF593A14**

---

Paging Example (8)

- Request for logical address **92E8**
  
  Page 9, so check entry at index 9:
  
  0  **00028**
  
  V = 0, so mapping is not valid
  
  invalid logical address
Page Table Implementation

- OS manages page table in RAM
- Direct mapped: page number is index
- Problem: page table can be large:
  - P bits in page number, $2^P$ page table entries
  - All page table entries allocated (even if unused)
- Solution: use multi-level page tables or similar technique to reduce table size

Page Table Implementation

- Problem: page table in RAM, so every address translation requires a memory access (doubles the number of accesses)
- Solution: cache of page table entries (TLB – translation look-aside buffer)
- Typical configuration:
  - Instruction cache (instructions)
  - Data cache (data values)
  - TLB (page table entries)
Page Table and TLB

TLB Processing

- Hit in TLB: map page number to frame (within 1 clock cycle)
- Miss in TLB: load page table entry into the TLB, then restart instruction (10-100 clock cycles)
Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (text editors, compilers, etc)
  - Shared code must appear in same location in the logical address space of all processes

- **Private code and data**
  - Each process has a separate copy of the private code and data (can be anywhere in logical address space)
Summary: Simple Paging

- Transparent to user: system manages paging without user having to be aware
- Simple for the OS to manage
- Supports swapping: page frames copied to and from disk
- No external fragmentation (but there is internal fragmentation)

Simple Segmentation

- Similar to dynamic partitioning: user program divided into segments
- Segments within a program do not need to be the same size, but there is a maximum size for all segments
- Each segment loaded into a partition in main memory; the partitions do not need to be adjacent
Simple Segmentation

- Operating system maintains a segment table for each process
  - Contains the starting physical address for each segment in the process and the length of that segment
  - Logical address consists of a segment number and offset within the segment
Simple Segmentation

Example

- OS views process as 2 segments (different sizes)
- Both segments fully used (no internal fragmentation)
- Logical address viewed as segment number and byte offset
- OS allocates 2 partitions (one for each segment), initializes segment table

**Summary: Simple Segmentation**

- Matches user's view of memory: program broken up into distinct areas of memory
- More complex for the OS to manage than paging
- Supports swapping, but not as well as paging
- No internal fragmentation (but there is external fragmentation)