Operating Systems

Memory Management
(Ch 4: 4.1-4.2)

Overview

- Provide Services (done)
  - processes (done)
  - files (briefly here, more in cs4513)
- Manage Devices
  - processor (done)
  - memory (next!)
  - disk (done after files)

Simple Memory Management

- One process in memory, using it all
  - each program needs I/O drivers
  - until 1960

Simple Memory Management

- Small, protected OS, drivers
  - DOS

Multiprocessing w/Fixed Partitions

- Unequal queues
- Waste large partition
- Skip small jobs

Hey, processes can be in different memory locations!

Address Binding

- Compile Time
  - maybe absolute binding (.com)
- Link Time
  - dynamic or static libraries
- Load Time
  - relocatable code
- Run Time
  - relocatable memory segments
  - overlays
  - paging
Normal Linking and Loading

**Printer.c**

```plaintext
gcc
```

**Main.c**

```plaintext
gcc
```

Static Library

```plaintext
ar
```

```plaintext
.a.out
```

Loader

Memory

Load Time Dynamic Linking

**Printer.c**

```plaintext
gcc
```

**Main.c**

```plaintext
gcc
```

Dynamic Library

```plaintext
ar
```

```plaintext
.a.out
```

Memory

Load Time Dynamic Linking Performance Comparisons

<table>
<thead>
<tr>
<th>Linking Method</th>
<th>Disk Space</th>
<th>Load Time</th>
<th>Run Time (4 used)</th>
<th>Run Time (2 used)</th>
<th>Run Time (0 used)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>3Mb</td>
<td>3.1s</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Load Time</td>
<td>1Mb</td>
<td>3.1s</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Run Time</td>
<td>1Mb</td>
<td>1.1s</td>
<td>2.4s</td>
<td>1.2s</td>
<td>0</td>
</tr>
</tbody>
</table>

Run-Time Dynamic Linking

**Printer.c**

```plaintext
gcc
```

**Main.c**

```plaintext
gcc
```

Dynamic Library

```plaintext
ar
```

```plaintext
.a.out
```

Run-time Loader

Memory

Save disk space.

Startup fast.

Might not need all.

Design Technique: Static vs. Dynamic

- Static solutions
  - compute ahead of time
  - for predictable situations
- Dynamic solutions
  - compute when needed
  - for unpredictable situations
- Some situations use dynamic because static too restrictive (malloc)
- ex: memory allocation, type checking

Logical vs. Physical Addresses

- Compile-Time + Load Time addresses same
- Run time addresses different

User goes from 0 to max

Physical goes from R+0 to R+max
Relocatable Code Basics

- Allow logical addresses
- Protect other processes
- Addresses must be contiguous!

Variable-Sized Partitions

- Idea: want to remove “wasted” memory that is not needed in each partition
- Definition:
  - Hole - a block of available memory
  - scattered throughout physical memory
- New process allocated memory from hole large enough to fit it

Variable-Sized Partitions

OS
process 5
process 8
process 2
OS
process 5
process 9
process 2
OS
process 9
process 10
process 2
OS
process 2
process 3
process 8
malloc(20k)?

Memory Request?

- What if a request for additional memory?

Internal Fragmentation

- Have some “empty” space for each processes
- Internal Fragmentation - allocated memory may be slightly larger than requested memory and not being used.

External Fragmentation

- External Fragmentation - total memory space exists to satisfy request but it is not contiguous

“But, how much does this matter?”
Analysis of External Fragmentation

- Assume:
  - system at equilibrium
  - process in middle
  - if \( N \) processes, \( 1/2 \) time process, \( 1/2 \) hole
    \[ \Rightarrow 1/2 N \text{ holes!} \]
  - Fifty-percent rule
  - Fundamental:
    - adjacent holes combined
    - adjacent processes not combined

Compaction

- Shuffle memory contents to place all free memory together in one large block
- Only if relocation dynamic!
- Same I/O DMA problem

Cost of Compaction

- 128 MB RAM, 100 nsec/access
  \[ \Rightarrow 1.5 \text{ seconds to compact!} \]
- Disk much slower!

Solution?

- Want to minimize external fragmentation
  - Large Blocks
  - But internal fragmentation!
- Tradeoff
  - Sacrifice some internal fragmentation for reduced external fragmentation
  - Paging

Where Are We?

- Memory Management
  - fixed partitions (done)
  - linking and loading (done)
  - variable partitions (done)
- Paging
- Misc

Paging

- Logical address space noncontiguous; process gets memory wherever available
  - Divide physical memory into fixed-size blocks
    - size is a power of 2, between 512 and 8192 bytes
    - called Frames
  - Divide logical memory into blocks of same size
    - called Pages
Paging

- Address generated by CPU divided into:
  - Page number (p) - index to page table
    - Page table contains base address of each page in physical memory (frame)
  - Page offset (d) - offset into page/frame

Paging Example

- Page size 4 bytes
- Memory size 32 bytes (8 pages)

Paging Hardware

- address space $2^m$
- page offset $2^n$
- page number $2^{m-n}$

Note: Not losing any bytes!

Paging Example

- Consider:
  - Physical memory = 128 bytes
  - Physical address space = 8 frames
- How many bits in an address?
- How many bits for page number?
- How many bits for page offset?
- Can a logical address space have only one page? How big would the page table be?

Another Paging Example

- Consider:
  - 8 bits in an address
  - 3 bits for the frame/page number
- How many bytes (words) of physical memory?
- How many frames are there?
- How many bytes is a page?
- How many bits for page offset?
- If a process' page table is 12 bits, how many logical pages does it have?
**Paging Tradeoffs**

- **Advantages**
  - No external fragmentation (no compaction)
  - Relocation (now pages, before were processes)

- **Disadvantages**
  - Internal fragmentation
    - Consider: 2048 byte pages, 72,766 byte proc
      - 35 pages + 1086 bytes = 962 bytes
      - Avg: 1/2 page per process
      - Small pages!
  - Overhead
    - Page table/process (context switch + space)
    - Lookup (especially if page to disk)

**Implementation of Page Table**

- Page table kept in registers
- Fast!
- Only good when number of frames is small
- Expensive!

**Associative Registers**

- **Hit Ratio** - Percentage of times that a page number is found in associative registers
  \[ \text{Effective access time} = \text{hit ratio} \times \text{hit time} + \text{miss ratio} \times \text{miss time} \]
  - Hit time = reg time + mem time
  - Miss time = reg time + mem time * 2

  **Example:**
  - 80% hit ratio, reg time = 20 nanosec, mem time = 100 nanosec
  - .80 * 120 + .20 * 220 = 140 nanoseconds
Protection
- Protection bits with each frame
- Store in page table
- Expand to more perms

Page Table

Protection Bit

Page 0
Page 1
Page 2

Logical Memory

Page Table

Physical Memory

Large Address Spaces
- Typical logical address spaces:
  - 4 Gbytes = $2^{32}$ address bits (4-byte address)
- Typical page size:
  - 4 Kbytes = $2^{12}$ bits
- Page table may have:
  - $2^{32}/2^{12} = 2^{20} = 1$ million entries
- Each entry 3 bytes => 3MB per process!
- Do not want that all in RAM
- Solution? Page the page table
  - Multilevel paging

Multilevel Paging

page number page offset
p1 p2 d
10 10 12

Logical Memory

Outer Page Table

Page Table

Memory View
- Paging lost users’ view of memory
- Need “logical” memory units that grow and contract
  - ex: stack, shared library
- Solution?
  - Segmentation!

Segmentation
- ex: stack, shared library
- Memory accesses longer! (search + swap)

Inverted Page Table
- Page table maps to physical addresses
- Still need page per process --> backing store
- Memory accesses longer! (search + swap)
Segmentation

• Logical address: <segment, offset>
• Segment table - maps two-dimensional user defined address into one-dimensional physical address
  – base - starting physical location
  – limit - length of segment
• Hardware support
  – Segment Table Base Register
  – Segment Table Length Register

Memory Management Outline

• Basic (done)
  – Fixed Partitions (done)
  – Variable Partitions (done)
• Paging (done)
  – Basic (done)
  – Enhanced (done)
• Specific
  – WinNT
  – Linux
• Linking and Loading

Memory Management in WinNT

• 32 bit addresses \(2^{32} = 4 \text{ GB address space}\)
  – Upper 2GB shared by all processes (kernel mode)
  – Lower 2GB private per process
• Page size is 4 KB \(2^{12}\), so offset is 12 bits
• Multilevel paging (2 levels)
  – 10 bits for outer page table (page directory)
  – 10 bits for inner page table
  – 12 bits for offset

Memory Management in Linux

• Page size:
  – Alpha AXP has 8 Kbyte page
  – Intel x86 has 4 Kbyte page
• Multilevel paging (3 levels)
  – Makes code more portable
  – Even though no hardware support on x86
    + “middle-layer” defined to be 1