Operating Systems

Memory Management

Overview

• Provide Services (done)
  – processes (done)
  – files (after memory management)
• 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

Multiprocessing w/Fixed Partitions

Simple!

• Unequal queues
• Waste large partition
• Skip small jobs

Hey, processes can be in different memory locations!

Simple Memory Management

• Small, protected OS, drivers
  – DOS

RAM
User Prog
I/O Drivers
OS
RAM
User Prog
ROM
Device Drivers
ROM

• “Mono-programming” – No multiprocessing!
  – Early efforts used “Swapping”, but slooow

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

```
PrintC
  gcc
  Linker
  Static Library
  a.out
  Loader
  Memory
```

Load Time Dynamic Linking

```
PrintC
  gcc
  Main.o
  Linker
  Dynamic Library
  a.out
  Loader
  Memory
```

X Window code:
- 500K minimum
- 450K libraries

• Save disk space.
• Libraries move?
• Moving code?
• Library versions?
• Load time all the same.

Run-Time Dynamic Linking

```
PrintC
  gcc
  Main.o
  Linker
  Dynamic Library
  a.out
  Loader
  Memory
```

Save disk space.
Startup fast.
Might not need all.

Memory 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 Load</td>
<td>3Mb</td>
<td>3.1s</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Static Run</td>
<td>1Mb</td>
<td>3.1s</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Dynamic Load</td>
<td>1Mb</td>
<td>1.1s</td>
<td>2.4s</td>
<td>1.2s</td>
<td>0</td>
</tr>
<tr>
<td>Dynamic Run</td>
<td>1Mb</td>
<td>1.1s</td>
<td>2.4s</td>
<td>1.2s</td>
<td>0</td>
</tr>
</tbody>
</table>

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

```
CPU
  Logical Address 346
  Relocation Register 14346
  Physical Address 14346
  Memory
```

• User goes from 0 to max
• Physical goes from R+0 to R+max
Relocatable Code Basics
- Allow logical addresses
- Protect other processes

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 keeps track of:
  - allocated partitions
  - free partitions (holes)
  - queues!

Memory Request?
- What if a request for additional memory?

Internal Fragmentation
- Have some “empty” space for each processes
- Allocated to

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 holes!
  - Fifty-percent rule
  - Fundamental:
    - Adjacent holes combined
    - Adjacent processes not combined

Cost of Compaction

- 128 MB RAM, 100 nsec/access
  \( \Rightarrow \) 1.5 seconds to compact!
- Disk much slower!

Compaction

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

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:
  - \textit{Page number} ($p$) - index to page table
  - page table contains base address of each page in physical memory (frame)
  - \textit{Page offset} ($d$) - offset into page/frame

```
CPU \rightarrow p \rightarrow \text{Page} \rightarrow \text{Frame} \rightarrow \text{Logical Memory} \rightarrow \text{Page Table} \rightarrow \text{Physical Memory} \rightarrow f \rightarrow \text{Page}
```

Paging Example

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

```
<table>
<thead>
<tr>
<th>Logical Memory</th>
<th>Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>0</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
</tr>
<tr>
<td>011</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Paging Hardware

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

```
p \quad \text{Page number} \quad d \quad \text{Page offset}
```

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 2 pages? 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?
Page Table Example

<table>
<thead>
<tr>
<th>Page Table</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process A</td>
<td>Page Table</td>
</tr>
<tr>
<td>Page 0</td>
<td>0</td>
</tr>
<tr>
<td>Page 0</td>
<td>1</td>
</tr>
<tr>
<td>Page 1</td>
<td>0</td>
</tr>
<tr>
<td>Page 1</td>
<td>1</td>
</tr>
</tbody>
</table>

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
    - 25 pages = 816 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!

Registers
Memory
Disk

Implementation of Page Table Base Register (PTBR)

Page Table Length

- Two memory accesses per data/inst access.
- Solution? Associative Registers

Associative Registers

- Hit Ratio - percentage of times that a page number is found in associative registers

  Effective access time =
  
  hit ratio \times \text{hit time} + \text{miss ratio} \times \text{miss time}

- hit time = \text{reg time} + \text{mem time}
- miss time = \text{reg time} + \text{mem time} \times 2

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

Large Address Spaces
- Typical logical address spaces:
  - 4 Bytes $\Rightarrow 2^{20}$ address bits (+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 $\Rightarrow 3$MB per process!
- Do not want that all in RAM
- Solution? Page the page table
  - Multilevel paging

Multilevel Paging

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

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