Overview

- Provide Services (done)
  - processes (done)
  - files (done after memory)
- 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 partitions
- 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
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!

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

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

- Given a list of free holes:
  - How do you satisfy a request of sizes?
    - 20k, 130k, 70k

Variable-Sized Partitions

- OS keeps track of:
  - allocated partitions
  - free partitions (holes)
  - queues!
Variable-Sized Partitions

- Requests: 20k, 130k, 70k
  - First-fit: allocate first hole that is big enough
  - Best-fit: allocate smallest hole that is big enough
  - Worst-fit: allocate largest hole (say, 120k)

Memory Request?

- What if a request for additional memory?
  - First-fit: might not search the entire list
  - Best-fit: must search the entire list
  - Worst-fit: must search the entire list
  - First-fit and Best-fit better than Worst-fit in terms of speed and storage utilization

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

Analysis of External Fragmentation

- Assume:
  - system at equilibrium
  - process in middle
  - if N processes, 1/2 time process, 1/2 hole
  - + => 1/2 N 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, 100nsec/access
  ➞ 1.5 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

Analysis of External Fragmentation
• Assume:
  – system at equilibrium
  – process in middle
  – if N processes, 1/2 time process, 1/2 hole
  ➔ 1/2 N holes!
  – Fifty-percent rule
  – Fundamental:
    ➔ adjacent holes combined
    ➔ adjacent processes not combined
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 Example

- Page size 4 bytes
- Memory size 32 bytes (8 pages)
**Paging Hardware**

- Address space $2^n$
- Page size $2^m$
- Page offset $2^{m-n}$

*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 pages? How big would the page table be?

**Page Table Example**

- Process A
  - Page 0
  - Page 1
- Process B
  - Page 0
  - Page 1

**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)

**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?

**Implementation of Page Table**

- Page table kept in registers
- Fast!
- Only good when number of frames is small
- Expensive!
Implementation of Page Table
- Page table kept in main memory
- **Page Table Base Register (PTBR)**
  
  ![Page Table Base Register Diagram]

- Page Table Length
- Two memory accesses per data/instruction access
  - Solution: **Associative Registers**

Associative Register Performance
- **Hit Ratio** - percentage of times that a page number is found in associative registers

**Effective access 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
  - 0.80 * 120 + 0.20 * 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 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
Multilevel Paging Translation

- Page number
- Page offset
- Inner page table
- Outer page table

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
- Solution? • Segmentation!
- Ex: stack, shared library
- Symbol table

Segmentation

- Logical address: \(<\text{segment, offset}>\)
- 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

Segmentation

- Logical address
  - Limit
  - Base
  - Physical address

("Er, what have we gained?")

Memory Management Outline

- Basic (done)
  - Fixed Partitions (done)
  - Variable Partitions (done)
- Paging (done)
  - Basic (done)
  - Enhanced (done)
- Specific
  - WinNT
  - Linux
- Virtual Memory

("Er, what have we gained?")

\(\Rightarrow\) Paged segments!
Memory Management in WinNT

- 32 bit addresses ($2^{32} = 4$ 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
- Each page-table entry has 32 bits
  - only 20 needed for address translation
  - 12 bits “left-over”
- Characteristics
  - Access: read-only, read-write
  - States: valid, zeroed, free …
- Inverted page table
  - Points to page table entries
  - List of free frames

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
- Buddy-heap
- Buddy-blocks are combined to larger block
- Linked list of free blocks at each size
- If not small enough, broken down

Object Module

- Information required to “load” into memory
- Header Information
- Machine Code
- Initialized Data
- Symbol Table
- Relocation Information
- (see SOS sample)

Linking an Object Module

- Combines several object modules into load module
- Resolve external references
- Relocation - each object module assumes starts at 0. Must change.
- Linking - modify addresses where one object refers to another (example - external)
Loading an Object
- Resolve references of object module

On Disk
- Header
- Executable Code
- Initialized Data
- Uninitialized Data

In Memory
- Executable Code
- Initialized Data
- Uninitialized Data

Normal Linking and Loading
- Printf .c
- Printf .o
- Static Library
- gcc
- ar
- a.out
- Linker
- Loader
- Memory

Load Time Dynamic Linking
- Printf .c
- Printf .o
- Dynamic Library
- gcc
- ar
- a.out
- Loader
- Memory

Run-Time Dynamic Linking
- Printf .c
- Printf .o
- Dynamic Library
- gcc
- ar
- a.out
- Loader
- Memory

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

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

- Save disk space.
- Libraries move?
- Moving code?
- Library versions?
- Load time still the same.

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