Operating Systems I
Memory Management

Overview
- Provide Services
  - processes ✓
  - files ×
- Manage Devices
  - processor ✓
  - memory ✓
  - disk ×

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
  - “Mono-programming” -- No multiprocessing

Multiprocessing w/Fixed Partitions
- Simple!
- Unequal queues
- Waste large partition
- Skip small jobs

Address Binding
- Compile Time
  - maybe absolute binding (.com)
- Link Time
  - dynamic vs. 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

CPU
 Logical Address
 Relocation Reg
 346
 MMU
 Physical Address
 14346
 Memory

• User goes from 0 to max
• Physical goes from R+0 to R+max

Relocatable Code
- Allow logical addresses
- Protect other processes

CPU
 Limit Reg
 Reloc Reg
 physical address

• Addresses must be contiguous!

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

Loading an Object
- Resolve references of object module

On Disk
Header
Executable Code
Initialized Data

In Memory
Executable Code
Initialized Data
Uninitialized Data

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!
Variable-Sized Partitions

- Given a list of free holes:
  - 100k
  - 140k
  - 110k
  - 25k
- How do you satisfy a request of sizes?
  - 20k, 130k, 70k

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)

- 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

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

OS
process 3
process 8
process 2
malloc(20k)?

Internal Fragmentation
- Have some “empty” space for each processes
  - A stack
  - A data
  - A program
  - Room for growth

- 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

OS
process 3
process 8
process 2
50k
100k
125k
Process 9

?
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

Compaction

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

<table>
<thead>
<tr>
<th>OS</th>
<th>50k</th>
<th>90k</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>125k</td>
<td>Process 9</td>
<td>process 3</td>
<td>process 8</td>
</tr>
<tr>
<td>process 8</td>
<td>50k</td>
<td>process 2</td>
<td>process 3</td>
</tr>
<tr>
<td>process 2</td>
<td>100k</td>
<td>process 2</td>
<td>process 3</td>
</tr>
</tbody>
</table>

Cost of Compaction

- Compaction of Memory vs. Swap (Disk)

| process 1 | 50k | process 1 |
| process 3 | 90k | process 8 |
| process 8 | 50k | process 2 |
| process 2 | 100k | |

- Disk much slower!

Solution?

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

Paging

- Logical address space of a process can be noncontiguous; process is allocated memory wherever latter is 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 size $2^n$
- page offset $2^{m-n}$

Page Table 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?
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!

Implementation of Page Table

- Page table kept in main memory
- **Page Table Base Register (PTBR)**
- Page Table Length
- Two memory accesses per data/inst access
  - Solution? Associative Registers

Associative Register Performance

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

**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
- 0.80 * 120 + 0.20 * 220 = 140 nanoseconds

Protection

- Protection bits with each frame
- Store in page table
- Expand to more perms

Diagram showing registers, memory, and disk.
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 Translation

- 2 memory access if miss
- Effective access time?
  - 90% hit rate
  - $.80 \times 120 + .20 \times 320 = 160$ nanoseconds

Inverted Page Table

- Page table maps to physical addresses
- Still need page per process --> 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!
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
  - Fixed Partitions
  - Variable Partitions
- Paging
  - Basic
  - Enhanced
- Specific
  - WinNT
  - Linux
- Virtual Memory

Memory Management in WinNT

- 32 bit addressess \(2^{32} = 4 \text{ GB} \text{ 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)
  - Even though hardware support on x86!
Memory Management in Linux

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

<table>
<thead>
<tr>
<th>16 KB</th>
<th>8 KB</th>
<th>8 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4 KB</td>
<td>4 KB</td>
</tr>
</tbody>
</table>