Lecture 7

Memory Management

April 25, 2006

Prof. Joe Pasquale
Department of Computer Science and Engineering
University of California, San Diego
Before We Begin ...

Read Chapter 8 (on Main Memory)

Programming Assignment #3
  • Due Sunday, May 7, midnight

Midterm Exam
  • May 11
  • Will cover all material up to that point
Process Memory

Each process requires memory to store

- **Text**: code of program
- **Data**: static variables, heap
- **Stack**: automatic variables, activation records
- **Other**: shared memory regions

**Memory characteristics**

- **Size**: fixed or variable (max size)
- **Permissions**: R, W, X
Process's Memory Address Space

Address space

- Set of addresses to access memory
- Typically, linear and sequential
- 0 to N-1 (for size N)

For process memory of size N

- Text (of size X) at 0 to X-1
- Data (of size Y) at X to X+Y-1
- Stack (of size Z) at N-Z to N-1
Compiler’s View of Memory

Compiler needs to generate memory addresses

- Needs empty region for text, data, stack
- Ideally, very large to allow data and stack to grow
- Another possibility: three empty regions

Compiler needs to know, but doesn’t at compile time

- Physical memory size, to place stack at high end
  - Could locate stack relative to run-time value in register
- Must avoid allocated regions in memory
Goal: Support Multiple Processes

To support multiple programs running “simultaneously”

- Support process abstraction
- Multiplex CPU time over all runnable processes

But, process requires more than CPU time: memory
Multiple Processes: CPU + Memory

Multiple Processes

Multiplexed CPU and Memory
Sharing the Physical Memory

When process is given CPU, it must also be in memory

Problem

- Context-switching time (CST): 10 $\mu$sec
- Loading from disk: 10 MB/s
- To load 1 MB process: 100 msec = 10,000 x CST
- Too much overhead! Breaks illusion of simultaneity

Solution: keep multiple processes in memory

- Context switch only between processes in memory
Issues and Topics

Where should process memories be placed?
  - Topic: Memory management

How does the compiler model memory?
  - Topics: Logical memory model, segmentation

How to deal with limited physical memory size?
  - Topics: Virtual memory, paging
Memory Management

Physical memory starts as one big empty space, or “hole”

When creating process, allocate memory

• Find a hole that can contain process
• Allocate region within hole
• Typically, leaves a (smaller) hole

When process exits, deallocate its memory

• Creates a hole
• If next to another hole, coalesce
Memory Management

Physical memory starts as one big empty space, or “hole”

When creating process, allocate memory
  • Find a hole that can contain process
  • Allocate region within hole
  • Typically, leaves a (smaller) hole

When process exits, deallocate its memory
  • Creates a hole
  • If next to another hole, coalesce
Memory Management

Physical memory starts as one big empty space, or “hole”

When creating process, allocate memory
• Find a hole that can contain process
• Allocate region within hole
• Typically, leaves a (smaller) hole

When process exits, deallocate its memory
• Creates a hole
• If next to another hole, coalesce
Selecting the Best Hole

When searching for a hole, what if there are multiple?

Algorithms
- First fit
- Next fit
- Best fit
- Worst fit

Complication
- Is region fixed or variable size?
Selecting the Best Hole

When searching for a hole, what if there are multiple?

Algorithms

• First fit
• Next fit
• Best fit
• Worst fit

Complication

• Is region fixed or variable size?
Selecting the Best Hole

When searching for a hole, what if there are multiple?

Algorithms

- First fit
- Next fit
- Best fit (leaves small fragments)
- Worst fit

Complication

- Is region fixed or variable size?
Selecting the Best Hole

When searching for a hole, what if there are multiple?

Algorithms

• First fit
• Next fit
• Best fit
• Worst fit

Complication

• Is region fixed or variable size?
Selecting the Best Hole

When searching for a hole, what if there are multiple?

Algorithms

• First fit
• Next fit
• Best fit
• Worst fit (leaves large fragments)

Complication

• Is region fixed or variable size?
Fragmentation

Eventually, memory becomes fragmented
- After repeated allocations/deallocations

Internal fragmentation
- Unused space within process
- Cannot be allocated to others
- Can come in handy for growth

External fragmentation
- Unused space outside any process (holes)
- Can be allocated (but often too small to be useful)
What If No Holes?

There may still be significant unused space

- External fragments
- Internal fragments

Approaches

- Compaction
- Break process memory into pieces (easier to fit)
What If No Holes?

There may still be significant unused space

• External fragments
• Internal fragments

Approaches

• Compaction
• Break process memory into pieces (easier to fit)
What If No Holes?

There may still be significant unused space

- External fragments
- Internal fragments

Approaches

- Compaction
- Break process memory into pieces (easier to fit)
The Buddy System

Dynamically partition in powers-of-2 size chunks

Allocation

Given request for size $r$, find smallest chunk
while ($r \leq \text{sizeof(chunk)}/2$)
divide chunk into 2 buddies (each of 1/2 size)

Deallocation

Free the chunk
while (buddy is also free)
coalesce
Example of Buddy System

<table>
<thead>
<tr>
<th>Alloc A</th>
<th>Alloc B</th>
<th>Alloc C</th>
<th>Free B</th>
<th>Free A</th>
</tr>
</thead>
<tbody>
<tr>
<td>900 KB</td>
<td>1.2 MB</td>
<td>1.5 MB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8 MB

<table>
<thead>
<tr>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MB</td>
<td>1 MB</td>
<td>1 MB</td>
<td>1 MB</td>
</tr>
<tr>
<td>2 MB</td>
<td>B</td>
<td>B</td>
<td>2 MB</td>
</tr>
<tr>
<td>4 MB</td>
<td>4 MB</td>
<td>C</td>
<td>2 MB</td>
</tr>
<tr>
<td>2 MB</td>
<td></td>
<td>C</td>
<td>2 MB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C</td>
</tr>
</tbody>
</table>

4 MB
Data Structure for Buddy System

Alloc A: 900 KB

8 MB
4 MB
4 MB
2 MB
2 MB
4 MB
4 MB
2 MB
2 MB
Data Structure for Buddy System

Alloc A: 900 KB

Alloc B: 1.2 MB
Data Structure for Buddy System

Alloc C: 1.5 MB
Data Structure for Buddy System

Free B
Data Structure for Buddy System

Free A
Data Structure for Buddy System

Coalesce

2 MB

2 MB

C

2 MB

Coalesce

4 MB

C

2 MB
Problems with Sharing Memory

The Addressing Problem
- Compiler generates memory references
- Unknown where process will be located

The Protection Problem
- Modifying another process’s memory

The Space Problem
- The more processes there are, the less memory each individually can have
Address Spaces

Address space
- set of addresses for memory

Usually linear: 0 to N-1 (size N)

Physical Address Space
- 0 to N-1, N = size
- kernel occupies lowest addresses (typically)
Logical vs. Physical Addressing

Logical addresses

- Assumes separate memory starting at 0
- Compiler generated
- Independent of location in physical memory

Converting logical to physical

- S/W: at load time
- H/W: at access time
Hardware for Logical Addressing

Base register filled with start address
To translate logical address, add base
Achieves relocation
To move process: change base
Protection

Bound register works with base register.

Is address < bound
  • Yes: add to base
  • No: invalid address, TRAP

Achieves protection
Memory Registers Part of Context

On Every Context Switch
  • Load base/bound registers for selected process
  • Only kernel does loading of these registers
  • Kernel must be protected from all processes

Benefit
  • Allows each process to be separately located
  • Protects each process from all others
Loading Programs into Memory

To create a process, must load it into memory

What to load (the load module) is based on program

- Text (code)
- Data (initialized and uninitialized)
- Stack (keeps track of pending calls, starts empty)