Lecture 12: Memory Management Overview

Yiying Zhang
Announcements

• Midterm to be graded within ~1 week
  ♦ Solutions will not be released, individual questions will be answered (tag zhang_section and exam on Piazza)
  ♦ Once graded, we will publish stats

• Start working on Project 2 now!
  ♦ It will be harder than project 1 and need more time
Memory Management

Next few lectures are going to cover memory management

• Goals of memory management
  ♦ To provide a convenient abstraction for programming
  ♦ To allocate scarce memory resources among competing processes to maximize performance with minimal overhead

• Mechanisms
  ♦ Physical and virtual addressing
  ♦ Techniques: partitioning, paging, segmentation
  ♦ Page table management, TLBs, VM tricks

• Policies
  ♦ Page replacement algorithms
Virtual Memory

- The abstraction that the OS provides for managing memory is virtual memory (VM)
  - Virtual memory enables a program to execute with less than its complete data in physical memory
    - A program can run on a machine with less memory than it “needs”
    - Can also run on a machine with “too much” physical memory
  - Many programs do not need all of their code and data at once (or ever) – no need to allocate memory for it
  - OS will adjust amount of memory allocated to a process based upon its behavior
  - VM requires hardware support and OS management algorithms to pull it off
The Big Picture

- main.c
- math.c
- main.o
- math.o
- main.o
- a.out
- loader
- memory management
- Instruction execution
- arch
- compiler
- linker

Load a.out to mem
Manage mem for proc
Let’s go back to the beginning

- Rewind to the very old days
  - Programs use physical addresses directly
  - OS loads job, runs it, unloads it
1. Simple uniprogramming: Single segment per process
Simple uniprogramming: Single segment per process

- The single segment contains code, data, stack, heap
- No virtual memory, directly using physical memory
  - When loading a process, just bring it in at 0
- Can only run one program at a time

Examples:
- early batch monitor which ran only one job at a time
  - if the job wrecks the OS, reboot OS
- 1st generation PCs operated in a similar fashion
Multiprogramming

- Want to let several processes coexist in main memory
Issues in sharing main memory

• Transparency:
  ♦ Processes should not know memory is shared
  ♦ Run regardless of the number/locations of processes

• Safety:
  ♦ Processes cannot corrupt each other

• Efficiency:
  ♦ Both CPU and memory utilization shouldn’t be degraded badly by sharing
2. Simple multiprogramming

With static software memory relocation, no protection, 1 segment per process:

• Highest memory holds OS
• Processes allocated memory starting at 0, up to the OS area
• When a process is loaded, relocate it so that it can run in its allocated memory area
2. Simple multiprogramming:
Single segment per process, static relocation

- OS
- Segment 2
- Segment 1
Simple multiprogramming:
Single segment per process, static relocation

- Segment 1 completed
- Segment 2
- Segment 3
- OS
- Segment 4?
Simple multiprogramming: Simple multiprogramming:  
Single segment per process, static relocation

- four drawbacks
  1. No protection
  2. Low utilization -- Cannot relocate dynamically
     » Addresses in binary is fixed (after loading)
     » Cannot do anything about holes
  3. No sharing -- Single segment per process
     » Cannot share part of process address space (e.g. text)
  4. Entire address space needs to fit in mem
     » Need to swap whole, very expensive!
What else can we do?

• Already tried
  ♦ Compile time / linking time
  ♦ Loading time

• Let us try execution time!
3. Dynamic memory relocation

- Instead of changing the address of a program before it’s loaded, change the address dynamically *during every reference*

Can this be done in software?
Virtual Addresses

- To make it easier to manage the memory of processes running in the system, we’re going to make them use virtual addresses
  - Virtual addresses are independent of the actual physical location of the data referenced
  - OS determines location of data in physical memory
  - Compiler+linker determines virtual memory. OS also allocates virtual memory (heap memory)
  - CPU executes instructions with virtual addresses
  - Virtual addresses are translated by hardware into physical addresses (with help from OS)

- The set of virtual addresses that can be used by a process comprises its virtual address space (VAS)
  - VAS often larger than physical memory (64-bit addresses)
  - But can also be smaller (32-bit VAS with 8 GB of memory)
The Big Picture

main.c
math.c

main.o
math.o

Virt Mem

Load a.out to mem
Manage mem for proc

linker

memory management

memory management

Translate and access phys mem

Execute inst w/ virt mem

Translate and access phys mem

Set up and manage virt->phys mem mapping

Instruction execution

Instruction execution

arch

Load a.out to mem
Manage mem for proc

compiler

arch

Set up and manage virt->phys mem mapping

memory management

Execute inst w/ virt mem

Translate and access phys mem

Memory Management

Load a.out to mem
Manage mem for proc

Instruction execution

Translate and access phys mem

Execute inst w/ virt mem
Translation overview

- Actual translation process is usually performed by hardware
- Translation table is set up by software
- CPU view
  - what program sees, virtual addresses
- Memory view
  - physical memory addresses
3.1 Base and bound

- Built in Cray-1 (1976)
- A program can only access physical memory in 
  [base, base+bound]
- On a context switch: save/restore base, bound registers

Pros:
- simple, fast translation, cheap
- Can relocate segment at execution time
3.1 Base and bound

- The essence:
  - A level of indirection
  - Phy. Addr = Vir. Addr + base

- Why do we need the limit register? Protection
  - If (physical address > base + limit) then an exception will happen
3.1 Base and bound

- Cons:
  - Relocation requires moving the entire address space
  - Only one segment per process
  - How can two processes share code while keeping private data areas?  
    » Can it be done safely with a single-segment scheme?
3.2 Multiple Segments

- Separate a virtual memory address space into multiple “segments”
- A hardware segment table of (seg base, size), each entry also has an associated permission (nil, read, write, exec)
- On a context switch: save/restore the table (or a pointer to the table) in kernel memory
Segmentation

- Segmentation is a technique that partitions memory into logically related data units
  - Module, procedure, stack, data, file, etc.
- Natural extension of base-and-bound
  - Base-and-bound: 1 segment/process
  - Segmentation: many segments/process
Segmented Address Space

Segment Table

<table>
<thead>
<tr>
<th>Seg base</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stack

Seg1, 0x00000000

Heap

Seg2, 0x00000000

Static Data (Data Segment)

Seg3, 0x00000000

Code (Text Segment)

Seg4, 0x00000000
Pros/cons of segmentation

• Pros:
  ♦ Process can be split among several segments
    » Allows sharing
  ♦ Segments can be assigned, moved, or swapped independently

• Cons:
  ♦ External fragmentation: many holes in physical memory
    » Also happens in base and bound scheme
External fragmentation with segmentation

OS

Segment 2

Segment 3

Segment 4?

External fragmentation
What fundamentally causes external fragmentation?

1. Segments of many different sizes

2. Each has to be allocated contiguously

• “Million-dollar” question:

  Physical memory is precious.
  Can we limit the waste to a single hole of $X$ bytes?
Next time...

- Chapters 18, 19, 20