Announcements

• Homework 2 graded, midterm being graded

• Grade distribution change
  ♦ Max((midterm∗0.21+final∗0.23), (midterm∗0.18+final∗0.26))

• Project 1 due this Thur, 5/7, 23:59pm
  ♦ No deadline extension!!!
  ♦ No consideration of late submissions. We will take a snapshot of your master github repo at 23:59pm if you do not submit in time.
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 (1)
  ♦ Techniques: partitioning, paging, segmentation (1)
  ♦ Page table management, TLBs, VM tricks (2)

• **Policies**
  ♦ Page replacement algorithms (3)
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
- Let’s go back to the beginning…
In 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

Physical memory

OS

Segment 1

address 0
Simple uniprogramming: Single segment per process

- Highest memory holds OS
- Process is allocated memory starting at 0, up to the OS area
- The single segment contains code, data, stack, heap
- When loading a process, just bring it in at 0
  - Directly using physical addresses

- 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

- Pros / Cons?
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
null
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
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:
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** (logical 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
- 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  →  main.o
math.c  →  math.o
compiler  →  linker

Virt Mem

a.out

Load a.out to mem
Manage mem for proc

Set up and manage virt->phys mem mapping

Instruction execution

Execute inst w/ virt mem

Translate and access phys 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 $[\text{base}, \text{base}+\text{bound}]$
- On a context switch: save/restore base, bound registers

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

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

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

- Pros:
  ♦ No internal fragmentation

- Cons:
  ♦ Only one segment per process
  ♦ How can two processes share code while keeping private data areas (shared editors)?
    » Can it be done safely with a single-segment scheme?
What have we solved?

• four drawbacks
  1. No protection
  2. Low utilization -- Cannot relocate dynamically
     » 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!
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

CSE 120 – Lecture 9 – Memory Management Overview
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

CSE 120 – Lecture 9 – Memory Management Overview
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?
Paging

- Paging solves the external fragmentation problem by using fixed sized units in both physical and virtual memory.
Paging

- Translating addresses
  - Virtual address has two parts: virtual page number and offset
  - Virtual page number (VPN) is an index into a page table
  - Page table determines page frame number (PFN)
  - Physical address is PFN::offset ("::" means concatenate)

- Page tables
  - Map virtual page number (VPN) to page frame number (PFN)
    - VPN is the index into the table that determines PFN
  - One page table entry (PTE) per page in virtual address space
    - Or, one PTE per VPN
Paging

- Context switch
  - similar to the segmentation scheme

- Pros:
  - easy to allocate memory
  - easy to swap
  - easy to share
**Paging Example**

- Pages are 4K
  - 4K → offset is 12 bits → VPN is 20 bits ($2^{20}$ VPNs), assuming 32-bit system

- Virtual address is 0x7468
  - Virtual page is 0x7, offset is 0x468 (lowest 12 bits of address)

- Page table entry 0x7 contains 0x2
  - Page frame number is 0x2
  - Seventh virtual page is at address 0x2000 (physical page 2)

- Physical address = 0x2000 :: 0x468 = 0x2468
Deep thinking: Paging implementation

- Translation: table lookup and bit substitution

- Why is this possible?

- Why can’t we do the same in segmentation?
Page Tables

- Page tables completely define the mapping between virtual pages and physical pages for an address space.
- Each process has an address space, so each process has a page table.
- Page tables are data structures maintained by the OS.
Page Table Entries (PTEs)

1 1 1 3 20  
M R V Prot Page Frame Number

- Page table entries control mapping
  - The **Modify** bit says whether or not the page has been written
    » It is set when a write to the page occurs
  - The **Reference** bit says whether the page has been accessed
    » It is set when a read or write to the page occurs
  - The **Valid** bit says whether or not the PTE can be used
    » It is checked each time the virtual address is used
  - The **Protection** bits say what operations are allowed on page
    » Read, write, execute
  - The **page frame number** (PFN) determines physical page
How many PTEs do we need? (assume page size is 4096 bytes)

• Worst case for 32-bit address machine?

• What about 64-bit address machine?

• Page size?
  ♦ Small page -> big table
    » 32-bit with 4k pages
  ♦ Large page -> small table but large internal fragmentation
Paging implementation – how does it really work?

- Where to store page table?
- How to use MMU?
  - Even small page tables too large to load into MMU
  - Page tables kept in mem and MMU only has their base addresses
    » What does MMU have to do?
Paging Advantages

• Easy to allocate memory
  ♦ Memory comes from a free list of fixed-sized chunks
  ♦ Allocating a page is just removing it from the list
  ♦ External fragmentation not a problem

• Easy to swap out chunks of a program
  ♦ All chunks are the same size
  ♦ Use valid bit to detect references to swapped pages
  ♦ Pages are a convenient multiple of the disk block size
Paging Limitations

- Can still have internal fragmentation
  - Process may not use memory in multiples of a page
- Memory reference overhead
  - 2 references per address lookup (page table, then memory)
  - Solution – use a hardware cache of lookups (next lec)
- Memory required to hold page table can be significant
  - Solution – hierarchical page tables (next lec)
Deep thinking

• Why does the page table we talked about so far have to be contiguous in the physical memory?
  ♦ Why did a segment have to be contiguous in memory?

• For a 4GB virtual address space, we just need 1M PTE (~4MB), what is the big deal?

• My PC has 2GB, why do we need PTEs for the entire 4GB address space?
Virtual memory
- Processes use virtual addresses
- Hardware translates virtual address into physical addresses with OS support

Evolution of techniques
- Single, fixed physical segment per process (no virt mem)
- Single segment per process, static relocation (no virt mem)
- Base-and-bound – dynamic relocating whole process
- Segmentation – multiple (variable-size) segments with dynamic relocation
- Paging – small, fixed size pages
Next time...

- Chapters 18, 19, 20