Lecture 9: Memory Management Overview

Yiying Zhang
Announcements

- Homework 2 due tonight
- Start working on Project 2!
  - It will be harder than project 1 and need more time
Non-Preemptive Scheduling

- The OS only has a chance to schedule threads on a core when the current running thread leaves its running state:
  - Yield, terminate, blocked by I/O, etc.

- How can we force a thread off its running state?

![Diagram showing thread states: Ready, Running, Blocked, with transitions for Create process, Scheduler dispatch, Yield, Terminate, Block for resource, Resource becomes available.](image-url)
A running process is interrupted by the timer, and CPU is switched to run another process.
[lec8] Goals and Assumptions

- **Goals (Performance metrics)**
  - **Minimize turnaround time**
    - avg time to complete a job
    - \( T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}} \)
  - **Maximize throughput**
    - operations (jobs) per second
    - Minimize overhead of context switches: large quanta
    - Efficient utilization (CPU, memory, disk etc)
  - **Short response time**
    - \( T_{\text{response}} = T_{\text{firstrun}} - T_{\text{arrival}} \)
    - type on a keyboard
    - Small quanta
  - **Fairness**
    - fair, no starvation, no deadlock
[lec8] Scheduling policies

- FIFO
  - Response time

- RR
  - Throughput
  - Avg. turnaround time

- SJF
  - Fairness
[lec8] Multiple Queue Scheduling

• Motivation: processes may be of different nature and can be easily classified
  ♦ e.g. foreground jobs vs. background jobs

• The method:
  ♦ Processes permanently assigned to one queue, based on processes priority / type
    » Preference to jobs with higher priorities
  ♦ Each queue can have its own scheduling algorithm
    » e.g. RR for foreground queue, FCFS for background queue

♦ Need a scheduling among the queues
  » e.g. fixed priority preemptive scheduling (high-pri queue trumps other)
  » e.g. time-slice between queues
Multilevel Feedback Queue (MLFQ)

- Problem: how to change priority?
- Jobs start at highest priority queue
- Feedback
  - If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).
  - If a job gives up the CPU before the time slice is up, it stays at the same priority level.
  - After a long time period, move all the jobs in the system to the topmost queue (aging)
MLFQ Example – a long job + short jobs in between

Time Slice

Leave I/O bound and interactive processes in higher-priority queue

Potential problem?
Starvation

Q2
A B

Q1
B

Q0
C C C
[lec8] MLFQ Example – a long job+short jobs, with boost

Boost Time

Time Slice

Q2
A C B

Q1
C B

Q0
C C C
Scheduling Overhead

- Operating systems aim to minimize overhead
  - Context switching is not doing any useful work and is pure overhead
  - Overhead includes context switch + making a scheduling decision

- Modern time-sharing OSes (Unix, Windows, …) time-slice processes in ready list
  - A process runs for its quantum, OS context switches to another, next process runs, etc.
  - A CPU-bound process will use its entire quantum (e.g., 10ms)
  - An IO-bound process will use part (e.g., 1ms), then issue IO
  - The IO-bound process goes on a wait queue, the OS switches to the next process to run, the IO-bound process goes back on the ready list when the IO completes
CPU utilization is the fraction of time the system is doing useful work (e.g., not context switching).

If the system has:
- Quantum of 10ms + context-switch and decision making overhead of 0.1ms
- 3 CPU-bound processes + round-robin scheduling

In steady-state, time is spent as follows:
- 10ms + 0.1ms + 10ms + 0.1ms + 10ms + 0.1ms
- CPU utilization = time doing useful work / total time
  - CPU utilization = (3*10ms) / (3*10ms + 3*0.1ms) = 30/30.3

If one process is IO-bound, it will not use full quantum:
- 10ms + 0.1ms + 10ms + 0.1ms + 1ms + 0.1ms
- CPU util = (2*10 + 1) / (2*10 + 1 + 3*0.1) = 21/21.3
Scheduler (dispatcher) is the module that gets invoked when a context switch needs to happen.

Scheduling algorithm determines which process runs, where processes are placed on queues.

Many potential goals of scheduling algorithms:
- Utilization, throughput, wait time, response time, etc.

Various algorithms to meet these goals:
- FCFS/FIFO, SJF, Priority, RR

Can combine algorithms:
- Multiple-level feedback queues
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
• 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
CSE 120 – Lecture 9 – Memory Management Overview
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
Simple multiprogramming:
Single segment per process, static relocation
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. 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  →  main.o

math.c  →  math.o

compiler  →  linker

a.out

Virt Mem

a.out to mem

Manage mem for proc

Loader

Instruction execution

Translate and access phys mem

Execute inst w/ virt mem

Set up and manage virt->phys mem mapping

memory management

Load a.out to mem

arch
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 \([\text{base}, \text{base+bound}]\)
- 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?
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

Heap

Static Data (Data Segment)

Code (Text Segment)

Seg1, 0x00000000

Seg2, 0x00000000

Seg3, 0x00000000

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?*
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

**Virtual address**

- VPN
- offset

**Page table**

- PFN
- offset

**Physical address**

-PFN
- offset

- page table size
- error
Paging Example

• Pages are 4K
  ♦ 4K → offset is 12 bits → VPN is 20 bits \(2^{20}\) VPNs), assuming 32bit 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?
Summary

- **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