CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Fall 2021 L16
Deadlocks, Main Memory

Slides based on
• Text by Silberschatz, Galvin, Gagne
• Various sources
Where we are: Deadlocks

• System Model
• Deadlock Characterization
• Methods for Handling Deadlocks
  – Deadlock Prevention
  – Deadlock Avoidance resource-allocation
  – Deadlock Detection
  – Recovery from Deadlock
• Livelock

Help Session this Wed: Discussion of Midterm. TAs available using Microsoft Teams, Piazza, email
• How do critical systems like (those in an aircraft) deal with the issue of deadlocks?
  – specialized real-time operating systems
• **Safe state** is definitely not deadlocked.
• **Banker’s algorithm**: When a process requests a resource, it may have to wait (**resource request algorithm**), and request not granted if the resulting system state is unsafe (**safety algorithm**)
  – Need \([i,j] = \text{Max}[i,j] – \text{Allocation}[i,j]\)
• **Work**: currently available resources of each type
• **Midterm**: raw and adjusted scores.
Example A: Banker’s Algorithm

- Is it a safe state?
- Yes, since the sequence < P1, P3, P4, P2, P0> satisfies safety criteria

<table>
<thead>
<tr>
<th>Process</th>
<th>Max</th>
<th>Allocation</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>7 5 3</td>
<td>0 1 0</td>
<td>7 4 3</td>
</tr>
<tr>
<td>P1</td>
<td>3 2 2</td>
<td>2 0 0</td>
<td>1 2 2</td>
</tr>
<tr>
<td>P2</td>
<td>9 0 2</td>
<td>3 0 2</td>
<td>6 0 0</td>
</tr>
<tr>
<td>P3</td>
<td>2 2 2</td>
<td>2 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>P4</td>
<td>4 3 3</td>
<td>0 0 2</td>
<td>4 3 1</td>
</tr>
</tbody>
</table>

P1 run to completion. Available becomes [3 3 2]+[2 0 0] = [5 3 2]
P3 run to completion. Available becomes [5 3 2]+[2 1 1] = [7 4 3]
P4 run to completion. Available becomes [7 4 3]+[0 0 2] = [7 4 5]
P2 run to completion. Available becomes [7 4 5]+[3 0 2] = [10 4 7]
P0 run to completion. Available becomes [10 4 7]+[0 1 0] = [10 5 7]

Hence state above is safe.
Deadlock Detection

• Allow system to enter deadlock state
• Detection algorithm
  – Single instance of each resource:
    • wait-for graph
  – Multiple instances:
    • detection algorithm (based on Banker’s algorithm)
• Recovery scheme
Example of Detection Algorithm

- Five processes $P_0$ through $P_4$; three resource types A (7 instances), B (2 instances), and C (6 instances)
- Sequence $<P_0, P_2, P_3, P_1, P_4>$ will result in $\text{Finish}[i] = \text{true}$ for all $i$. \textbf{No deadlock}

<table>
<thead>
<tr>
<th>Process</th>
<th>Allocation</th>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>available</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>After</th>
<th>available</th>
</tr>
</thead>
<tbody>
<tr>
<td>ini</td>
<td>0 0 0</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
</tr>
<tr>
<td>P2</td>
<td>3 1 3</td>
</tr>
<tr>
<td>P3</td>
<td>5 2 4</td>
</tr>
<tr>
<td>P1</td>
<td>7 2 4</td>
</tr>
<tr>
<td>P4</td>
<td>7 2 6</td>
</tr>
</tbody>
</table>
Recovery from Deadlock: Process Termination

Choices

• Abort all deadlocked processes

• Abort one process at a time until the deadlock cycle is eliminated
  
  – Selecting a victim – minimize cost

  – Rollback – return to some safe state, restart process for that state

  – Starvation – same process may always be picked as victim, include number of rollbacks in cost factor
Welcome to CS370 Second Half

- Topics: Memory, Storage, File System, Virtualization
- Class rules: See Syllabus
  - Class, Canvas, Teams
  - participation
  - Final
    - Sec 001, local 801: in class.
    - Sec 801 non-local: on-line.
    - SDC: Sec 001, Sec 801: must be taken at SDC
  - Project, deadlines, Plagiarism
Some OS History Lessons 1

- History in Unix-like OSs
Some OS History Lessons 2

- 1974: CP/M Intel 8080, Gary Kildall, Digital Research
  - 8-bit, min 16 kB RAM, floppy
  - Inspired by CP/M?
- 1981: PC DOS, Bill Gates, Microsoft
  - 86-DOS licensed for $25,000, hired Paterson
- 1985: Windows, Bill Gates, Microsoft
  - GUI inspired by MAC? Xerox PARC Star?

Gary Kildall net worth $1.9 Million at death
Tim Paterson Net Worth: $250,000
Main Memory

Slides based on
- Text by Silberschatz, Galvin, Gagne
- Various sources
Main Memory

Slides based on
- Text by Silberschatz, Galvin, Gagne
- Various sources
Chapter 8: Main Memory

Objectives:

• Organizing memory for multiprogramming environment
  • Partitioned vs separate address spaces
• Memory-management techniques
  • Virtual vs physical addresses
  • Chunks
    • segmentation
    • Paging: page tables, caching ("TLBs")
• Examples: the Intel (old/new) and ARM architectures
What we want

• Memory capacities have been increasing
  – But programs are getting bigger faster
  – Parkinson’s Law*: Programs expand to fill the memory available to hold

• What we would like
  – Memory that is
    • infinitely large, infinitely fast
    • Non-volatile
    • Inexpensive too

• Unfortunately, no such memory exists as of now

*work expands so as to fill the time available for its completion. 1955
Background

• Program must be brought (from disk) into memory and run as a process
• Main memory and registers are only storage CPU can access directly
• Memory unit only sees a stream of
  – addresses + read requests, or
  – address + data and write requests
• \( n \)-bit address: address space of size \( 2^n \) bytes.
  – Ex: 32 bits: addresses 0 to \((2^{32} - 1)\) bytes
  – Addressable unit is always 1 byte.
• Access times:
  – Register access in one CPU clock (or less)
  – Main memory can take many cycles, causing a stall
  – Cache sits between main memory and CPU registers making main memory appear much faster
• Protection of memory required to ensure correct operation

\[ 2^{10} = 1,024 \approx K \]
\[ 2^{20} = 1,048,576 \approx M \]
\[ 2^{30} \approx G \]
Main memory and registers are only storage CPU can access directly. Register access in one CPU clock (or less). Main memory can take many cycles, causing a stall.

Cache sits between main memory and CPU registers making main memory appear much faster.

Ch 9

Ch 10

Ch 11,13,14,16: Disk, file system

Cache: CS470
The History of Memory

In 1940 I invented a bit of memory.

1953: I invented a byte of memory.

1966: 1k

1978: 32k

2011: Look, 100 terabytes

2038: What memory?
Protection: Making sure each process has separate memory spaces

• OS must be protected from accesses by user processes
• User processes must be protected from one another
  – Determine range of legal addresses for each process
  – Ensure that process can access only those

• Approaches:
  – Partitioning address space (early system)
  – Separate address spaces (modern practice)
Partitioning: Base and Limit Registers

- Base and Limit for a process
  - **Base**: Smallest legal physical address
  - **Limit**: Size of the range of physical address
- A pair of **base** and **limit registers** define the logical address space for a process
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user
- **Base**: **Smallest** legal physical address
- **Limit**: Size of the **range** of physical address
- Eg: **Base** = 300040 and **limit** = 120900
- Legal: 300040 to \((300040 + 120900 - 1) = 420939\)
Hardware Address Protection

Legal addresses: Base address to Base address + limit -1
Multistep Processing of a User Program
Address Binding Questions

- Programs on disk, ready to be brought into memory to execute form an input queue
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Addresses represented in different ways at different stages of a program’s life
  - Source code addresses are symbolic
  - Compiled code addresses bind to relocatable addresses
    - i.e., “14 bytes from beginning of this module”
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e., 74014
  - Each binding maps one address space to another
• Address binding of instructions and data to memory addresses can happen at three different stages
  – **Compile time**: If memory location known a priori, *absolute code* can be generated; must recompile code if starting location changes
  – **Load time**: Must generate *relocatable code* if memory location is not known at compile time
  – **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    • Need hardware support for address maps (e.g., base and limit registers)
Separate Address Spaces

- Each process has its own private address space.
  - **Logical address space** is the set of all logical addresses used by a process.

- However, the physical memory has just one address space.
  - **Physical address space** is the set of all physical addresses.

- Need to map one to the other.
Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses
Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
  - Many methods possible, we will see them soon
- Consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers
- The **user program** deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses
Dynamic relocation using a relocation register
Loading vs Linking

• Loading
  – Load executable into memory prior to execution

• Linking
  – Takes some smaller executables and joins them together as a single larger executable.
Linking: Static vs Dynamic

- **Static linking** – system libraries and program code combined by the loader into the binary image
  - Every program includes library: wastes memory
- **Dynamic linking** – linking postponed until execution time
  - Operating system checks if routine is in processes’ memory address
Dynamic Linking

- **Dynamic linking** – linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine and executes the routine
- Operating system checks if routine is in processes’ memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for
  - shared libraries
Dynamic loading of routines

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- OS can help by providing libraries to implement dynamic loading
  - Static library
    - Linux. .a (archive)
    - Windows .lib (Library)
  - Dynamic Library
    - Linux .so (Shared object)
    - Windows .dll (Dynamic link library)
Swapping a process

• A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  – Total physical memory space of processes can exceed physical memory
• **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
• Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
• System maintains a **ready queue** of ready-to-run processes which have memory images on disk
Do we really need to keep the entire process in the main memory? Stay tuned.
Context Switch Time including Swapping

• If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
• Context switch time can then be very high
• 100MB process swapping to hard disk with transfer rate of 50MB/sec
  – Swap out time of 100MB/50MB/s = 2 seconds
  – Plus swap in of same sized process
  – Total context switch swapping component time of 4 seconds + some latency
• Can reduce if reduce size of memory swapped – by knowing how much memory really being used by a process
• Standard swapping not used in modern operating systems
  – But modified version common
    • Swap only when free memory extremely low
Memory Allocation Approaches

- **Contiguous allocation**: entire memory for a program in a single contiguous memory block. Find where a program will “fit”. Earliest approach.

- **Segmentation**: program divided into logically divided “segments” such as main program, functions, stack etc.
  - Need table to track segments.

- **Paging**: program divided into fixed size “pages”, each placed in a fixed size “frame”.
  - Need table to track pages.
Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vectors
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory
• **Registers** used to protect user processes from each other, and from changing operating-system code and data
  – **Relocation (Base) register** contains value of smallest physical address
  – **Limit register** contains range of logical addresses – each logical address must be less than the limit register

• **MMU** maps logical address *dynamically*
Hardware Support for Relocation and Limit Registers

MMU maps logical address *dynamically*

*Physical address = relocation reg + valid logical address*
Multiple-partition allocation

- Degree of multiprogramming limited by number of partitions
- **Variable-partition** sizes for efficiency (sized to a given process’ needs)
- **Hole** – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
  a) allocated partitions  
  b) free partitions (hole)

![Diagram showing memory allocation and process scheduling with holes and allocated partitions](image-url)
Dynamic Storage-Allocation Problem

How to satisfy a request of size $n$ from a list of free holes?

- **First-fit**: Allocate the first hole that is big enough
- **Best-fit**: Allocate the smallest hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the largest hole; must also search entire list
  - Produces the largest leftover hole

**Simulation studies:**

- First-fit and best-fit better than worst-fit in terms of speed and storage utilization
- Best fit is **slower** than first fit. Surprisingly, it also results in more **wasted memory** than first fit
  - Tends to fill up memory with tiny, useless holes
Fragmentation

• **External Fragmentation** – External fragmentation: memory wasted due to small chunks of free memory interspersed among allocated regions

• **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

• Simulation analysis reveals that given \( N \) blocks allocated, \( 0.5 \ N \) blocks lost to fragmentation
  – 1/3 may be unusable → **50-percent rule**
Fragmentation (Cont.)

• Reduce external fragmentation by **compaction**
  – Shuffle memory contents to place all free memory together in one large block
  – Compaction is possible *only* if relocation is dynamic, and is done at execution time
  – I/O problem
    • Latch job in memory while it is involved in I/O
    • Do I/O only into OS buffers
**Segmentation:** program divided into logically divided “segments” such as main program, function, stack etc.
- Need table to track segments.
- Term “segmentation fault occurs”: improper attempt to access a memory location

**Paging:** program divided into fixed size “pages”, each placed in a fixed size “frame”.
- Need table to track pages.
- No external fragmentation
- Increasingly more common
Paging vs Segmentations

![Brick Wall](Image)

![Rock Wall](Image)
Pages

- Pages and frames
  - Addresses: page number, offset
- Page tables: mapping from page # to frame #
  - TLB: page table caching
- Memory protection and sharing
- Multilevel page tables