CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Spring 2019 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
- Liveloop
• How do critical systems like (those in an aircraft) deal with the issue of deadlocks?
  – Multiple systems that often run specialized real-time operating systems

• **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] = Max[i,j] – Allocation[i,j]

• **Work**: currently available resources of each type
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>type</td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>available</td>
<td></td>
<td>3 3 2</td>
<td></td>
</tr>
<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.
Bankers Algorithm: Practical Issues

- Processes rarely know in advance about their maximum resource needs
- Number of processes is not fixed
  - Varies dynamically
- Resources thought to be available can disappear
- Few systems use this algorithm
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
• Maintain **wait-for graph** *(based on resource allocation graph)*
  – Nodes are processes
  – $P_i \rightarrow P_j$ if $P_i$ is waiting for $P_j$

  – **Deadlock if cycles**

• Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

• An algorithm to detect a cycle in a graph requires an order of $n^2$ operations, where $n$ is the number of vertices in the graph
Resource-Allocation Graph and Wait-for Graph

(a) Resource-Allocation Graph

(b) Corresponding wait-for graph

Has cycles. Deadlock.
Several Instances of a Resource Type

Banker’s algorithm: Can requests by all process be satisfied?

• **Available:** A vector of length \( m \) indicates the number of available resources of each type initially.
• **Work:** A vector of length \( m \) indicates the number of available resources of each type at a given time.
• **Allocation:** An \( n \times m \) matrix defines the number of resources of each type currently allocated to each process.
• **Request:** An \( n \times m \) matrix indicates the current request of each process. If \( \text{Request} [i][j] = k \), then process \( P_i \) is requesting \( k \) more instances of resource type \( R_j \).
Detection Algorithm

1. Let $\textbf{Work}$ and $\textbf{Finish}$ be vectors of length $m$ and $n$, respectively. Initialize:
   (a) $\textbf{Work} = \text{Available}$
   (b) For $i = 1, 2, \ldots, n$, if $\text{Allocation}_i \neq 0$, then $\text{Finish}[i] = false$; otherwise, $\text{Finish}[i] = true$

2. Find an index $i$ such that both:
   (a) $\text{Finish}[i] == false$
   (b) $\text{Request}_i \leq \text{Work}$
   If no such $i$ exists, go to step 4

3. $\text{Work} = \text{Work} + \text{Allocation}_i$
   $\text{Finish}[i] = true$
   go to step 2 (find next process)

4. If $\text{Finish}[i] == false$, for some $i$, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if $\text{Finish}[i] == false$, then $P_i$ is deadlocked

Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state
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 $i$. **No deadlock**

<table>
<thead>
<tr>
<th>Process</th>
<th>Allocation</th>
<th>Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>A  B  C</td>
<td>A  B  C</td>
</tr>
<tr>
<td>available</td>
<td>0 0 0</td>
<td></td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>P1</td>
<td>2 0 0</td>
<td>2 0 2</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 3</td>
<td>0 0 0</td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>0 0 2</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>
Example of Detection Algorithm (cont)

• $P_2$ requests an additional instance of type $C$

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

• State of system?
  – Can reclaim resources held by process $P_0$, but insufficient resources to fulfill other processes; requests
  – Deadlock exists, consisting of processes $P_1$, $P_2$, $P_3$, and $P_4$
Detection-Algorithm Usage

• When, and how often, to invoke depends on:
  – How often a deadlock is likely to occur
  – How many processes will need to be rolled back
    • one for each disjoint cycle

• If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.
Recovery from Deadlock: Process Termination

Choices

• Abort all deadlocked processes

• Abort one process at a time until the deadlock cycle is eliminated

In which order should we choose to abort?

1. Priority of the process
2. How long process has computed, and how much longer to completion
3. Resources the process has used
4. Resources process needs to complete
5. How many processes will need to be terminated
6. Is process interactive or batch?
Recovery from Deadlock: Resource Preemption

- **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
Deadlock recovery through rollbacks

- **Checkpoint** process periodically
  - Contains memory image and resource state
- Deadlock detection tells us *which* resources are needed
- Process owning a needed resource
  - **Rolled back** to before it acquired needed resource
    - Work done since rolled back checkpoint discarded
  - **Assign** resource to deadlocked process
Livelocks

In a livelock two processes need each other’s resource
• Both run and make no progress, but neither process blocks
• Use CPU quantum over and over without making progress

Ex: If fork fails because process table is full
• Wait for some time and try again
• But there could be a collection of processes each trying to do the same thing
• Avoided by ensuring that only one process (chosen randomly or by priority) takes action

Two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass. But they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.
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
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
- 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
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 11, 13, 14, 16: Disk, file system

Cache: CS470
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
  – 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**
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)
THE HISTORY OF MEMORY

IN 1940 I INVENTED A BIT OF MEMORY

1953: I INVENTED A BYTE OF MEMORY

1966: 1 K

1978: 32 K
ONLY $999

2011: LOOK, 100 TERABYTES

2038: WHAT MEMORY?
Multistep Processing of a User Program
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.
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)
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.
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.
• 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
Contiguous Allocation