Memory Management

Notice: The slides for this lecture have been largely based on those accompanying the textbook Operating Systems Concepts with Java, by Silberschatz, Galvin, and Gagne (2003). Many, if not all, the illustrations contained in this presentation come from this source.
Last time:

Deadlock Detection & Recovery
Detection Algorithm

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

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

3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   Go to step 2.

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

- Five processes \( P_0 \) through \( P_4 \): three resource types
  A (7 instances), B (2 instances), and C (6 instances).

- Snapshot at time \( T_0 \):

<table>
<thead>
<tr>
<th></th>
<th>Allocation</th>
<th>Request</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_0 )</td>
<td>0 1 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>( P_1 )</td>
<td>2 0 0</td>
<td>2 0 2</td>
<td></td>
</tr>
<tr>
<td>( P_2 )</td>
<td>3 0 3</td>
<td>0 0 0</td>
<td></td>
</tr>
<tr>
<td>( P_3 )</td>
<td>2 1 1</td>
<td>1 0 0</td>
<td></td>
</tr>
<tr>
<td>( P_4 )</td>
<td>0 0 2</td>
<td>0 0 2</td>
<td></td>
</tr>
</tbody>
</table>

- Sequence \( \langle P_0, P_2, P_3, P_1, P_4 \rangle \) will result in \( \text{Finish}[i] = \text{true} \) for all \( i \).
Example (Cont.)

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

  \[
  \text{Request} \\
  A B C \\
  P_0 \quad 0 \quad 0 \quad 0 \\
  P_1 \quad 2 \quad 0 \quad 1 \\
  P_2 \quad 0 \quad 0 \quad 1 \\
  P_3 \quad 1 \quad 0 \quad 0 \\
  P_4 \quad 0 \quad 0 \quad 2 \\
  \]

• State of the system?
  – Can reclaim resources held by process $P_0$, but have insufficient resources to fulfill the requests of other processes.
  – 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

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - 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 rollback in cost factor.
Combined Approach to Deadlock Handling

- Combine the three basic approaches
  - prevention
  - avoidance
  - detection
  allowing the use of the optimal approach for each of resources in the system.

- Partition resources into hierarchically ordered classes.

- Use most appropriate technique for handling deadlocks within each class.
Memory Management
Background

- Program must be brought into memory and placed within a process for it to be run.

- **Input queue** – collection of processes on the disk that are waiting to be brought into memory to run the program.

- User programs go through several steps before being run.
Binding of Instructions and Data to Memory

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*).
Processing of a User Program

- Source program
  - Compiler or assembler
    - Object module
      - Linkage editor
        - Load module
          - Loader
            - In-memory binary image
              - System library
                - Dynamically loaded system library
                  - Dynamic linking

Compile time
Load time
Execution time
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 and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.
Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address.

- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.

- The user program deals with *logical* addresses; it never sees the *real* physical addresses.
Dynamic relocation using a relocation register

CPU \[ \text{logical address} \rightarrow \text{MMU} \rightarrow \text{physical address} \]

14000

\[ + \]

14346

memory

03/10/2004 CSCI 315 Operating Systems Design
## Dynamic Loading

- Routine is not loaded until it is called.
- Better memory-space utilization; unused routine is never loaded.
- Useful when large amounts of code are needed to handle infrequently occurring cases.
- No special support from the operating system is required implemented through program design.
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 needed to check if routine is in processes’ memory address.
- Dynamic linking is particularly useful for libraries.
Overlays

- Keep in memory only those instructions and data that are needed at any given time.
- Needed when process is larger than amount of memory allocated to it.
- **Implemented by user**, no special support needed from operating system, programming design of overlay structure is complex.
Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.

- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.

- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower priority process is swapped out so higher priority process can be loaded and executed.

- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.

- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows).
Schematic View of Swapping

Operating System

main memory

user space

swap out

process $P_1$

swap in

process $P_2$

backing storage
Contiguous Allocation

- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector.
  - User processes then held in high memory.

- Single-partition allocation
  - Relocation register scheme used to protect user processes from each other, and from changing operating system code and data.
  - Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register.
Hardware Support for Relocation and Limit Registers

[Diagram showing the flow of data from CPU to memory through logical and physical address processing, including limit and relocation registers.]

03/10/2004
CSCI 315 Operating Systems Design
Contiguous Allocation

- Multiple-partition allocation
  - **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.
  - Operating system maintains information about:
    a) allocated partitions  b) free partitions (hole)
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.

First-fit and best-fit better than worst-fit in terms of speed and storage utilization.
Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous.
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- 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.
Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.
- Divide physical memory into fixed sized blocks called *frames* (size is power of 2, between 512 bytes and 8192 bytes).
- Divide logical memory into blocks of same size called *pages*.
- Keep track of all free frames.
- To run a program of size $n$ pages, need to find $n$ free frames and load program.
- Set up a page table to translate logical to physical addresses.
- Internal fragmentation.
Address Translation Scheme

• Address generated by CPU is divided into:
  – *Page number* \((p)\) – used as an index into a
    *page table* which contains base address of
    each page in physical memory.

  – *Page offset* \((d)\) – combined with base address
    to define the physical memory address that is
    sent to the memory unit.
Address Translation Architecture
Paging Example

<table>
<thead>
<tr>
<th>Frame number</th>
<th>page 0</th>
<th>page 1</th>
<th>page 2</th>
<th>page 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Logical memory

Page table

Physical memory
## Paging Example

<table>
<thead>
<tr>
<th>Logical Memory</th>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 a 1 b 2 c 3 d 4 e 5 f 6 g 7 h 8 i 9 j 10 k 11 l 12 m 13 n 14 o 15 p</td>
<td>0 5 1 6 2 1 0 2</td>
</tr>
</tbody>
</table>

Page Table

03/10/2004 CSCI 315 Operating Systems Design
Free Frames

Before allocation

After allocation