Deadlock Prevention Techniques

- Prevent hold and wait: one shot allocation
  - Conservative and inefficient

- Prevent "no preemption": allow preemption, by releasing or grabbing resources
  - Need checkpointing/recovery; can lead to livelock

- Prevent circular wait
  - Impose a hierarchy on resources; acquire resources only in one direction

Deadlock Avoidance

- Three General Approaches:
  1. Use a protocol that will guarantee deadlock cannot occur
  - Deadlock prevention: ensure that one of the ingredients necessary for deadlock cannot happen
  - Deadlock avoidance: a smarter way of avoiding deadlock (later)
  2. Allow deadlock to happen but detect it if it happens and recover
  3. Do nothing (??); actually used by many OS’s including unix
  - If it is infrequent, why worry about it?
  - If user processes are deadlock, they will eventually kill them (a slow form of deadlock recovery?)

Handling Deadlock

- Prevent hold and wait: one shot allocation
  - Conservative and inefficient

- Prevent "no preemption": allow preemption, by releasing or grabbing resources
  - Need checkpointing/recovery; can lead to livelock

- Prevent circular wait
  - Impose a hierarchy on resources; acquire resources only in one direction

Administrivia

- Project discussion?
- Last time
  - Wrapped up deadlock
- Today:
  - Start memory management
Banker’s Algorithm

\( S = \text{set of processes} \);
while (S is not empty){
  1. Find a process \( p \) such that
     \[
     \text{foreach } i \quad \text{Need}[p,i] \leq V[i]
     \]
  2. If impossible -- fail; state is unsafe
  3. Remove \( p \) from S;
      add \( p \)'s resources Available pool
}

- A safe state is a realizable state where there exits at least one sequence for the processes to run to completion
- Allow a resource request if the resulting state is safe
- How do we determine if a state is safe?
- Algorithm above is \( O(n^2) \) – more efficient implementation exists (Habbermann’s algorithm)

Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Allocation ( A )</th>
<th>Claim ( B )</th>
<th>( C )</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

\( V = [3 \ 3 \ 2] \)

- Is the current state safe? Is it deadlocked?
  - Generate need matrix
  - Apply safety algorithm

- New request from \( P1 \) (1, 0, 2); allow it or not?

Deadlock Detection

- Never block a process if enough resources are available
- Detect deadlock and recover once it occurs
- How to detect deadlock? (think about detecting a deadlock-free state)
  - Add a matrix \( Q \) for each processes currently outstanding resource requests
  - Update banker’s algorithm to pick a process that “may” finish (given its currently held resources + outstanding requests)
    - Replace need matrix with the outstanding request matrix
    - Apply “safety” algorithm; if it succeeds, no deadlock – why?
- When to run the deadlock detection algorithm?
- What to do if there is deadlock? Kill processes or preempt resources

Deadlock Detection Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Allocation ( A )</th>
<th>Outstanding ( A )</th>
<th>( B )</th>
<th>( C )</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

\( V = [2 \ 3 \ 2] \)

- Deadlocked state?
Memory Management – Motivation

- Primary goal is to bring programs into memory for execution
- What is the physical organization of the memory?
- How can the operating system help manage it?
- Memory management is easy for uniprogramming systems; load and execute
  - What happens if the program is bigger than the memory?
- Multiprogramming makes life interesting:
  - How to share the memory between the processes?
  - What addition problems/side-effects occur because of this sharing; how do we solve them?
- Input queue: queue of processes on disk waiting to be loaded to memory

Compiling, Linking and Loading

Address Binding

- Absolute binding: bind memory references at compile (link) time; program has to go in specific place in memory
- Relocateable binding: Compiler produces relative addresses
  - At load time, these addresses are translated.
  - What happens if the process is swapped?
- Dynamic run-time binding: References are kept relative, final translation is done at run time (when the reference occurs)
  - Too expensive? In software – Yes; hardware mechanisms assists

Dynamic Loading

- Problem: what to do with memory references/branches; will the program go in the same place in memory every time?
Linking

- External references (to other object files, or libraries), are usually resolved at linking time.
- Some OS's allow dynamic linking.
- Reference to some libraries is postponed until execution time.
  - A small piece of code (called stub) is used to locate the appropriate memory resident library routine.
  - Stub replaces itself with the address of the library routine.
  - OS is needed to check if the required code is in the process memory address (e.g., if another process does, then the OS is the only entity that can allow sharing).
- What if the required memory is larger than the physical memory size?

Overlays

- Key Idea: Suppose we keep only the parts of the program we are currently using in memory.
- Overlays is the oldest incarnation of this idea.
- Two pass assembler; only 150Kbytes physical memory available.

Swapping

- With overlays a process loads the portion of its image that is needed to fit in memory.
- Swapping is similar (remember medium term scheduler); a process can be swapped temporarily from memory into a swap area on a disk.
- Major cost is swap time between the disk and memory – cost proportional to size of process.
- Swapping is used in most modern OS’s including Unix and Windows.

Memory Management Requirements

- Relocation: we do not know beforehand where the program will actually go in memory.
- Protection: firewalls are needed to protect programs from interfering with the OS or with each other.
- Sharing: the protection mechanism should be flexible enough to allow portions of the memory to be shared (instructions or data).
- Logical vs. Physical address
  - Each process has a logical address space that is bound to a separate physical address space.
  - Logical address
    - also called virtual address
    - generated by the CPU
    - this is the address in the program after linking
  - Physical address – address seen by the memory
  - Impossible/too expensive to translate at runtime using software. Memory Management Unit translates.
Address Translation – Logical to Physical

• For each contiguous chunk in memory
  – Base (or Relocation) register; add to logical address to produce physical address
  – Limit (or Bound) register; check if the address is within limit (logical or physical address?)
• Why not translate at load time (and after swaps)?
  – What about dynamic addresses (pointers); can you resolve those before the program starts execution? memory aliasing problem
  – Also, can be expensive to recompute all the addresses and update everyone

How to share the memory

• What if we share it in time? Single Partition
  – One process gets the full memory
  – When it is blocked (or preempted), swap it out to disk
  – Swap in the process your schedule wants to run next
  – OS (or at least the kernel) resides in memory always
  – Usually in low address area along with interrupt tables
• For now, we assume contiguous allocation

Fixed Partitions

• Memory is statically divided into partitions; these sizes are fixed in the OS (recompile OS to change)
• Partition sizes can be equal or not equal
• What happens if the partition is too big for a program? internal fragmentation
• What happens if partition is too small for a program?
  – Use overlays at a big cost to swap in/swap out
• Do these problems disappear if partition size was not equal?
• Problem: restricts the number of “ready” processes to the number of partitions
• What does the placement algorithm look like (queue-per-partition vs. single-queue)

Variable Size Partitions – Dynamic Partitioning

• Allocate memory on a need-to-basis:
  – find an available memory big enough to fit process
  – allocate exactly the needed space to the process
• Is internal fragmentation a problem?
• Is external fragmentation a problem?
• This better or worse than fixed partitions?
• Requires periodic compaction

SUNY-Binghamton – CS 350 Spring ’08 Lec. #13

SUNY-Binghamton – CS 350 Spring ’08 Lec. #13
A new process comes in, where should it be placed in memory?

Find a suitable area (free; and equal or bigger than what we need)

Some policies:
- First fit: The first suitable area where the process fits
- Next fit: The first suitable area starting from the placement
- Best fit: The smallest of the suitable areas

What are the tradeoffs in using these?

Best fit performs worst; why?

Example

<table>
<thead>
<tr>
<th>job queue</th>
<th>process</th>
<th>memory</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>600K</td>
<td>1000K</td>
<td>10</td>
</tr>
<tr>
<td>P2</td>
<td>1000K</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>300K</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>700K</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>500K</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

Example (cont'd)

Compaction

Example
Discussion

- Fixed Partitioning:
  - Limit the number of active processes statically
  - Internal Fragmentation
- Dynamic Partitioning:
  - Complex to maintain
  - External Fragmentation/overhead for compaction
- Can we strike a balance?
- What happens if we swap out or relocate (for example, due to compaction) memory while an I/O device is writing to it? Two options
  - force the job to stay in memory where it is if it has active I/O
  - Do I/O only to OS buffers, copy when done

Buddy System

gethole(i) {
    if (i > u)
        fail;
    if (list_of_i’s is empty) {
        hole = gethole(i+1);
        split hole into buddies;
        put buddies on list_of_i’s;
    }
    return first hole on list_of_i’s;
}

- Example: free memory is 1Meg ($2^{20}$); the following processes are executed 140k, 100k, 60k, 256k