**Outline**

- Announcements
  - Lab 4 due next Wednesday (March 30th)
    - demos on 30th, 31st
- Process Deadlocks (cont’d)
  - Methods for handling deadlocks
    - Deadlock prevention
    - Deadlock avoidance
    - Deadlock detection and recovery
- Memory Management
  - logical versus physical address space
  - swapping
  - allocation schemes

[Silberschatz/Galvin/Gagne: Sections 7.2 – 7.8, Chapter 8]

---

**(Review)**

**Deadlock Avoidance: Notion of a Safe State**

- A system is in a safe state iff there exists a safe sequence
- A sequence \(<P_1, P_2, \ldots, P_n>\) is a safe sequence for the current allocation if, for each \(P_i\), the resources that \(P_i\) can still request can be satisfied by the currently available resources plus resources held by all the \(P_j\), for \(j < i\)

---

**Safe Sequences/States and Deadlock Avoidance**

- A safe state is not a deadlock state
- An unsafe state may lead to deadlock
- It is possible to go from a safe state to an unsafe state
- Avoidance algorithms prevent the system from entering an unsafe state
  - By blocking requests

**Single resource instance case**

- Deadlock \(\equiv\) Cycle in the resource allocation graph
- A request is granted iff it does not result in a cycle
  - cycle detection: \(O(V + E)\) operations

**Multiple resource instance case**

- More complicated to determine if system would go into an unsafe state
Deadlock Avoidance: Multiple Resource Instances

- Banker’s Algorithm
  - upon entering the system, a process declares the maximum number of instances of each resource type that it may need
  - the algorithm decides, for each request, whether granting it would put the system in an unsafe state

resource availability
Available[1..m]
maximum demand
Max[1..n, 1..m]
current allocation
Allocation[1..n, 1..m]
potential need
Need[1..n, 1..m]

1. If Request i ≤ Need i
   goto Step 2, else flag error
2. If Request i ≤ Available
   goto Step 3, else wait
3. Allocate the resources
   Available := Available - Request i;
   Allocation i := Allocation i + Request i;
   Need i := Need i - Request i;
   Check if this is a safe state.
   If not: undo the allocation and wait

4. If Finish[i] = true for all i,
   then the system is in a safe state

Banker’s Algorithm: Example

- Three resource types and three processes (P 1, P 2, P 3)
  Capacity = [2, 4, 3]
  Max = [[1, 2, 2], [1, 2, 1], [1, 1, 1]]
  Allocation = [[1, 2, 0], [0, 1, 1], [1, 0, 1]]
  Available = [0, 1, 1]
  Need = [[0, 0, 2], [1, 1, 0], [0, 1, 0]]

1. Work := Available;
   Finish[i] := false, for all i;
2. If Request i ≤ Available
   goto Step 3, else wait
3. Allocate the resources
   Available := Available - Request i;
   Allocation i := Allocation i + Request i;
   Need i := Need i - Request i;
   Check if this is a safe state.
   If not: undo the allocation and wait
4. If Finish[i] = true for all i,
   then the system is in a safe state

Limitations of Deadlock Avoidance

- Deadlock avoidance vs. deadlock prevention
  - Prevention schemes work with local information
    - What does this process already have, what is it asking
  - Avoidance schemes work with global information
    - Therefore, are less conservative

- However, avoidance schemes require specification of future needs
  - not generally known for OS processes
  - more applicable to specialized situations
    - programming language constructs (e.g., transaction-based systems)
    - known OS components (e.g., Unix “exec”)

- More general solution: Deadlock detection and recovery

Deadlock Detection: Single Resource Instances

- Go back to using a resource allocation graph in which only
  - request and assignment edges are defined
  - future (potential) requests are not relevant to “is there deadlock now?”

- Deadlock ≡ Cycle in the RAG
  - need only look at the wait-for graph
    - obtained by removing resource nodes and collapsing the appropriate edges
Deadlock Detection: Multiple Resource Instances

- A cycle in the graph is a necessary but not sufficient condition for the existence of a deadlock
  - if a cycle does not exist: no deadlock
  - if a cycle exists: there may or may not be a deadlock

(Examples from earlier in the lecture)

Detection: Multiple Resource Instances (cont’d)

- A new use for the Bankers’ algorithm
  - detect if the current set of requests are such that satisfying any of them will put the system in an unsafe state

1. Work := Available;
   Finish[i] := false, for all i;

2. Find an i such that
   a. Finish[i] = false, and
   b. Request[i] ≤ Work
   if no such i, goto Step 4

3. Work := Work + Allocation[i];
   Finish[i] := true;
   goto Step 2;

4. If Finish[i] = false for some i,
   then the system is in a deadlock state

Detection: Multiple Resource Instances (Example)

- System with three resource types and five processes

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Request</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0 [0, 1, 0]</td>
<td>[0, 0, 0]</td>
<td>[3, 1, 3]</td>
</tr>
<tr>
<td>P1 [2, 0, 0]</td>
<td>[2, 0, 0]</td>
<td></td>
</tr>
<tr>
<td>P2 [0, 0, 0]</td>
<td>[0, 0, 0]</td>
<td></td>
</tr>
<tr>
<td>P3 [2, 0, 3]</td>
<td>[0, 0, 0]</td>
<td></td>
</tr>
<tr>
<td>P4 [1, 0, 1]</td>
<td>[1, 0, 0]</td>
<td></td>
</tr>
</tbody>
</table>

No deadlock!

- What about the following?

<table>
<thead>
<tr>
<th>Allocation</th>
<th>Request</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0 [0, 1, 0]</td>
<td>[0, 0, 0]</td>
<td>[0, 1, 0]</td>
</tr>
<tr>
<td>P1 [2, 0, 0]</td>
<td>[2, 0, 0]</td>
<td></td>
</tr>
<tr>
<td>P2 [0, 0, 0]</td>
<td>[0, 0, 0]</td>
<td></td>
</tr>
<tr>
<td>P3 [2, 0, 3]</td>
<td>[0, 0, 0]</td>
<td></td>
</tr>
<tr>
<td>P4 [1, 0, 1]</td>
<td>[1, 0, 0]</td>
<td></td>
</tr>
</tbody>
</table>

Deadlock!

Deadlock Recovery

- Only general principles known (read Section 7.7 for details)

Two options

- Break the cyclic waiting by terminating some of the processes
  - choice 1: abort all deadlocked processes
  - choice 2: abort one process at a time till deadlock resolved

- Enable at least one of the processes to make progress
  (by preempting resources from another)
  - issue 1: how is the victim process selected?
  - issue 2: can the process handle resource preemption?
    - in general, might require rollback and restart
    - issue 3: how does one prevent starvation?
      - bound the number of rollbacks/preemptions for a particular process
Combined Approaches

- Using only a single approach (prevention, avoidance, or detection + recovery) in isolation is not very effective

Combination is superior
- General idea: Classify resources, use different approach for each
- Example: Consider a system with four classes of resources
  - internal resources (e.g., PCBs)
  - main memory
  - job resources (e.g., tape drives, files)
  - swappable space
- A combined deadlock solution
  - process control blocks: use resource ordering (prevention) Why?
  - user process memory: use pre-emption (detection/recovery)
  - job resources: require prior claims (avoidance)
  - swappable space: preallocate; no hold and wait (prevention)

Outline

- Announcements
  - Lab 4 due next Wednesday (March 30th)
    • demos on 30th, 31st
- Process Deadlocks (cont’d)
  - Methods for handling deadlocks
    • Deadlock prevention
    • Deadlock avoidance
    • Deadlock detection and recovery
- Memory Management
  - logical versus physical address space
  - swapping
  - allocation schemes

Background

- Programs operate on data and instructions stored in memory (von Neumann model)
  - memory is shared by multiple processes and is limited in size
  - further, the actual programming prior to compilation uses symbolic representations of these locations which get translated into actual (or physical) memory locations
- Memory management: Providing efficient mechanisms for
  - binding: mapping program names into actual memory locations
  - mapping: utilizing the limited physical memory to bring logical memory objects (belonging to multiple processes) back and forth
    • Lectures 13 and 14: allocation of physical memory to processes
      - assume that the entire process fits in physical memory
    • Lectures 14 and 15: supporting virtual memory in allocated physical memory
      - process data and instructions need not all fit into physical memory

Binding Program Names: Logical to Physical

at compile-time
  - mapping of logical-to-physical addresses is done statically
  - changes in the physical address map require recompilation
    • rare for general programs, sometimes for OS components

at load-time
  - binding done by the loader when program is brought into memory for execution
    • change in the starting address only requires a reload

at run-time
  - binding is delayed until the program actually executes
    • special hardware support needed to accomplish this
    • more details in the rest of the lecture

[Silberschatz/Galvin/Gagne: Sections 7.2 – 7.8, Chapter 8]
Process Memory Requirements

- So far, we have assumed that the entire process and data need to fit into memory for the program to execute
  - Many techniques to reduce amount that needs to fit at any time

Explicit management by the programmer
- dynamic loading
  - load procedures “on demand”
- overlays
  - keep in memory only those instructions/data that are needed at any given time
  - rewrite portions of the address space with new instructions/data as required

Implicit management by the OS
- Dynamic linking
  - typically used with shared system libraries that are loaded on demand
    - calls resolved using an “import table”: initially point to the loading stub
- Large virtual address spaces
  - more about this in Lectures 14 and 15

Multiprogramming and Swapping

- Problem: Memory requirements of all the processes cannot be simultaneously met

Solution: Swapping
- “Dynamically” move a process out of memory into a backing store (and back in) as dictated by the medium-term scheduler
  - backing store is typically a fast disk
  - choice of which processes to swap out/in
    - can be influenced by short-term scheduling policy (e.g., priority-driven)
    - knowledge of process’ actual memory requirements
      - requires the process to reserve, commit, decommit, and release memory

Swapping: Issues

- High context-switch times
  - assume a user process of size 100 KB
  - backing store is a standard hard disk with transfer rate of 5 MB/s
  - actual transfer of 100 KB from and to memory takes
    \[2 \times \left(\frac{100 \text{ KB}}{5000 \text{ KB/s}}\right) = 2 \times \left(\frac{1}{50} \text{ second}\right)\]
    \[= 2 \times (20 \text{ ms}) = 40 \text{ ms} + \text{disk time}\]
  - helps to know exactly how much memory is being used
  - also, determines frequency

- Swapping out a process that is currently in the middle of I/O
  - I/O completion might store values in memory, now occupied by a new process
  - common solutions
    - never swap out a process while in a wait state induced by I/O requests
    - all I/O interactions are via a special set of buffers that are controlled by the OS and are part of its space; not swapped out
Memory Mapping Schemes

- **Goal:** Allocate physical memory to processes
  - translate process logical addresses into physical memory addresses

- **Objectives**
  - memory protection
    - users from other users, system from users
  - efficient use of memory
  - programmer convenience
    - large virtual memory space

- **Three schemes**
  - Partitioning
  - Paging
  - Segmentation (if time permits)

Memory Mapping (1): Partitioning

- **Idea:** Divide memory into partitions

- **Protection**
  - each partition protected with a "key"
  - at run time, process key (stored in a register) matched with partition key
    - on mismatch, generates a trap

- **Allocation**
  - fixed partitions
    - memory is divided into a number of fixed size partitions
    - each partition is allotted to a single process
    - used in the early IBM 360 models
    - no longer in use
  - variable partitions
    - contiguous memory is allocated on loading
    - released on termination
    - this is what you are using in Nachos Lab 4

Memory Mapping: Partitioning (cont’d)

- **Partitioning for statically-bound programs**
  - programs must execute in the same place
  - allocation is inefficient, and swapping is very constrained
  - no provision for changing memory requirements

- **Partitioning for dynamically-bound programs**
  - relocation registers
    - a CPU register keeps track of the starting address where the process is loaded
    - whenever a memory location is accessed:
      - the system computes physical-address = logical-address + relocation register
      - fetches the value from the resulting memory location
    - the stream of physical addresses are seen only by the MMU
  - how to prevent a process from accessing addresses outside its partition?

Memory Mapping: Partitioning (cont’d)

- **Protection and relocation for dynamically-bound programs**
  - Two registers keep info for each partition: limit, relocation

- **Other advantages**
  - relocation register can be changed on the fly
  - why is this useful?
Memory Allocation and Scheduling

4 Processes: P1 (320K), P2 (224K), P3 (288K), P4 (128K)

Partitioning Policies

- Memory is viewed as sequence of blocks and voids (holes)
  - blocks are in use
  - voids are available: neighboring voids are coalesced to satisfy request

- Question: Given a request for process memory and list of current voids, how to satisfy the request
  - First fit: allocate space from the first void in the list that is big enough
  - Best fit: allocate space from a void to leave minimum remaining space
  - Worst fit: allocate a void such that the remaining space is a maximum

Partitioning Policies (contd.)

- Criterion for evaluating a policy: Fragmentation
  - External fragmentation
    - void space between blocks that does not serve any useful purpose
    - statistical analysis of first-fit: ~0.5N blocks will be lost due to fragmentation
    - can be avoided by compaction
      - Swap out a partition
      - Swap it back into another part of memory: requires relocation
  - Internal fragmentation
    - it is not worth maintaining memory that leaves very small voids (e.g., a few bytes) between used regions
      - occurs more obviously when unit of allocation is large (e.g., disks)
    - Happens when memory request is smaller than the smallest partition size

Memory Compaction: Reducing Fragmentation

- Moving partitions around can group the voids together
  - increase likelihood of their being used to satisfy a future request

- Many ways of doing this:
Memory Mapping (2): Paging

- **Motivation:** Partitioning suffers from large external fragmentation

**Paging**
- view physical memory as composed of several fixed-size frames
  - a "frame" is a physical memory allocation unit
- view logical memory as consisting of blocks of the same size: pages
- allocation problem
  - put "pages" into "frames"
    - a page table maintains the mapping
    - allocation need not preserve the contiguity of logical memory
      - e.g., pages 1, 2, 3, 4 can be allocated to frames 3, 7, 9, 14
      - how does this avoid external fragmentation?
- paging played a major role in virtual memory design
  - separation between the meaning of a location in the user's virtual space and its actual physical storage

---

Paging (example)

**Mapping of pages to frames**
- the mapping is hidden from the user and is controlled via the OS

**Allocation of frames to processes (Nachos Lab 4)**
- the OS maintains a map of the available and allotted frames via a structure called a frame table
  - whether a frame is allocated or not
  - if allocated, to which page of which process

**Address translation**
- performed on every memory access
- must be performed extremely efficiently so as to not degrade performance
- typical scheme
  - frames (and pages) are of size \( 2^k \)
  - for each logical address of \( a = m + n \) bits
    - the higher order \( m \) bits indicate the page number \( p \)
    - the remaining \( n \) bits indicate the offset \( w \) into the page

---

Page Table Lookup

- Mapping between pages and frames is maintained by a page table
  - the page number \( p \) is used to index into the \( p \)th entry of the (process') page table where the corresponding frame number \( f \) is stored

- All of this requires hardware support
  - since performed on every memory access
Page Table Structure

- Page table typically stored in memory
  - a single page table base register that
    - points to the beginning of the page table
    - $p_i$ is now the offset into this table
  - problem
    - requires two accesses to memory for each value
    - even with caches, can become very slow

- Solution: Translation Lookaside Buffer (TLB)
  - a portion of the page table is cached in the TLB
    - little performance degradation if a value is a hit in the TLB
    - if not: a memory access is needed to load the value into the TLB
      - an existing value must be flushed if the TLB is full
  - E.g.: Average memory access time for a system with 90% hit rate in TLB
    $$= 0.9 \times (\text{Access}_{\text{TLB}} + \text{Access}_{\text{mem}}) + 0.1 \times (\text{Access}_{\text{mem}} + \text{Access}_{\text{mem}})$$
    $$= 1.1 \times (\text{Access}_{\text{mem}})$$

Multi-level Page Tables

- Rationale: Modern systems support a very large logical address space
  - page tables themselves become very large
    - e.g., for a system with 32-bit logical addresses and 4K pages
      - we need $2^{20}$ page table entries (4 bytes per PTE implies 4 MB of space)
- Solution: page the page table itself
  - cost: additional memory accesses (but caching helps)

Page Tables and Sharing

- Page tables permit different virtual addresses (frames of different processes) to map to the same physical address
  - convenient sharing of common code (dynamically-linked system libraries)
  - shared data segments for IPC

[Diagram of logical memory and page tables for P1 and P2]