Deadlock Avoidance

- Deadlock occurs because processes are waiting on each other to release resources
- Main idea of deadlock avoidance:
  - request additional information about how resources are to be requested
  - before allocating request, check if system will enter a deadlock state
    \[ F(\text{resources available, resources allocated, future requests/releases}) \]
    - if no: grant the request
    - if yes: block the process
- Algorithms differ in amount and type of information
  - simplest (also most useful) model: maximum number of resources
  - other choices
    - sequence of requests and releases
    - alternate request paths
- How can we find out if a system will enter a deadlock state?
Properties of Safe States

- 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

Example: A system with 12 units of a resource
- Three processes
  - P1: max need = 10, current need = 5
  - P2: max need = 4, current need = 2
  - P3: max need = 9, current need = 2
- This is a safe state, since a safe sequence <P2, P1, P3> exists
- P3 requests an additional unit. Should this request be granted?
  - No, because this would put the system in an unsafe state
- P1, P2, P3 will then hold 5, 2, and 3 resources (2 units are available)
- P2’s future needs can be satisfied, but no way to satisfy P1’s and P3’s needs

Avoidance algorithms **prevent** the system from entering an unsafe state

---

Deadlock Avoidance: Single Resource Instances

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

\[
\text{<P3, P2, P1> is a safe sequence}
\]

\[
\text{Say P1 requests R3: Should this be granted?}
\]

- No, because an assignment edge from R3 to P1 would create a cycle in the RAG.
  - No safe sequence exists
- Does this always imply a deadlock?
  - No, because P1 can release R3 before requesting R1

---

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

1. If \( \text{Request}_i \leq \text{Need}_i \), goto Step 2, else flag error
2. If \( \text{Request}_i \leq \text{Available} \), goto Step 3, else wait
3. Allocate the resources
   - \( \text{Available} := \text{Available} - \text{Request}_i \)
   - \( \text{Allocation} := \text{Allocation} + \text{Request}_i \)
   - \( \text{Need} := \text{Need} - \text{Request}_i \)
   - Check if this is a safe state.
   - If not: undo the allocation and wait
4. If \( \text{Finish}_i = \text{true} \) for all \( i \), then the system is in a safe state

1. \( \text{Work} := \text{Available} \)
2. Find an \( i \) such that
   - \( \text{Finish}_i = \text{false} \), and
   - \( \text{Need}_i \leq \text{Work} \)
   - if no such \( i \), goto Step 4
3. \( \text{Work} := \text{Work} + \text{Allocation} \)
4. If \( \text{Finish}_i = \text{true} \) for all \( i \), then the system is in a safe state

Banker’s Algorithm: Example

- Three resource types and three processes (P1, P2, P3)

<table>
<thead>
<tr>
<th>Capacity</th>
<th>([2, 4, 3])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max (=)</td>
<td>([1, 2, 2], [1, 2, 1], [1, 1, 1])</td>
</tr>
<tr>
<td>Allocation</td>
<td>([1, 0, 1])</td>
</tr>
<tr>
<td>Available</td>
<td>([0, 1, 0])</td>
</tr>
<tr>
<td>Need</td>
<td>([0, 0, 2], [1, 1, 0], [0, 1, 0])</td>
</tr>
</tbody>
</table>

- P1 requests \([0, 0, 1]\)
  - Should this be granted?
- Allocate and check if system is in a safe state
  - \( \text{Allocation} := \text{Allocation} + \text{Request}_i \)
  - \( \text{Available} := \text{Available} - \text{Request}_i \)
  - \( \text{Need} := \text{Need} - \text{Request}_i \)

Initially, \( \text{Work} = [0, 1, 0] \)
- Need \( \leq \text{Work} \), so P3 can finish
- \( \text{Work} = [1, 1, 1] \)
  - Now, both P1 and P2 can finish
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 last 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. \( \text{Work} := \text{Available}; \)
   \( \text{Finish}[i] := \text{false}, \text{for all } i; \)

2. Find an \( i \) such that
   - a. \( \text{Finish}[i] = \text{false}, \text{and} \)
   - b. \( \text{Request}_i \leq \text{Work} \)
   if no such \( i \), goto Step 4

3. \( \text{Work} := \text{Work} + \text{Allocation}_i; \)
   \( \text{Finish}[i] := \text{true}; \)
   goto Step 2;

4. If \( \text{Finish}[i] = \text{false} \) for some \( i \),
   then the system is in a deadlock state

1. \( \text{Work} := \text{Available}; \)
   \( \text{Finish}[i] := \text{false}, \text{for all } i; \)

2. Find an \( i \) such that
   - a. \( \text{Finish}[i] = \text{false}, \text{and} \)
   - b. \( \text{Need}_i \leq \text{Work} \)
   if no such \( i \), goto Step 4

3. \( \text{Work} := \text{Work} + \text{Allocation}_i; \)
   \( \text{Finish}[i] := \text{true}; \)
   goto Step 2;

4. If \( \text{Finish}[i] = \text{true} \) for all \( i \),
   then the system is in a safe 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><strong>✓</strong> P1 [2, 0, 0]</td>
<td>[2, 0, 2]</td>
<td></td>
</tr>
<tr>
<td>P2 [3, 0, 3]</td>
<td>[0, 0, 0]</td>
<td></td>
</tr>
<tr>
<td>P3 [2, 1, 1]</td>
<td>[1, 0, 0]</td>
<td></td>
</tr>
<tr>
<td>✓✓✓✓ P0 [0, 1, 0]</td>
<td>✓✓✓✓</td>
<td></td>
</tr>
<tr>
<td>P1 [2, 0, 0]</td>
<td>[2, 0, 2]</td>
<td></td>
</tr>
<tr>
<td><strong>✓</strong> P2 [3, 0, 3]</td>
<td>[0, 0, 0]</td>
<td></td>
</tr>
<tr>
<td>P3 [2, 1, 1]</td>
<td>[1, 0, 0]</td>
<td></td>
</tr>
<tr>
<td>✓✓✓✓ P4 [0, 0, 2]</td>
<td>✓✓✓✓</td>
<td></td>
</tr>
</tbody>
</table>

### Deadlock Recovery

- **Only general principles known** (read Section 8.7 for details)

Two options

- Break the cyclic waiting by **terminating** some of the processes
- Enable at least one of the processes to make progress (by preempting resources from another)

#### Two options

- choice 1: abort all deadlocked processes
- choice 2: abort one process at a time till deadlock resolved

- 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) Why?
  - swappable space: preallocate; no hold and wait (prevention)

### Outline

- **Announcements**
  - Lab 4 due Monday, April 7th
  - Demos on April 7th and 8th
  - Note: This lab is more involved than Labs 1–3, so start early!
    - New lecture material required for lab: Virtual addressing (today)

- **Deadlocks (cont’d)**
  - Avoidance techniques
  - Detection and recovery

- **Memory Management**
  - logical versus physical address space
  - swapping
  - allocation

[ Silberschatz/Galvin/Gagne: Sections 8.5 – 8.7, 9.1 – 9.3]
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 12 and 13: 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

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

Process Memory Requirements (cont’d)

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

Explicit management by the programmer

- symbol table
- common routines
- overlay
- driver
- Pass 1
- Pass 2

Implicit management by the OS

- 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, deccommit, 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 (Lecture 13)
  - Segmentation (Lecture 13)

Memory Mapping: 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 will use 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

- 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
  - fast and good in terms of storage utilization
- Best fit: allocate space from a void to leave minimum remaining space
  - very good storage utilization
- Worst fit: allocate a void such that the remaining space is a maximum
  - requires peculiar memory loads to perform well in terms of storage utilization

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
  - fast and good in terms of storage utilization
- Best fit: allocate space from a void to leave minimum remaining space
  - very good storage utilization
- Worst fit: allocate a void such that the remaining space is a maximum
  - requires peculiar memory loads to perform well in terms of storage utilization
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:

  ![Diagram showing memory compaction and void migration](image)

  - OS
  - P1
  - P2
  - P3

  Initial state:
  - OS
  - P1
  - P2
  - P3

  Steps:
  1. Move 600 K
  2. Move 400 K
  3. Move 200 K