Frequently asked questions from the previous class survey

- Difference between request and assignment edges is simply the direction? Yes.
- Is D involved in a deadlock? A → B → C → A and D → A
- Swap space?
Topics covered in this lecture

- Deadlock Prevention
- Deadlock Avoidance

DEADLOCK CHARACTERIZATION
Deadlocks:
Necessary Conditions (I)

- **Mutual Exclusion**
  - At least one resource held in nonsharable mode
  - When a resource is being used
    - Another requesting process must wait for its release

- **Hold-and-wait**
  - A process must hold one resource
  - Wait to acquire additional resources
    - Which are currently held by other processes

Deadlocks:
Necessary Conditions (II)

- **No preemption**
  - Resources cannot be preempted
  - Only voluntary release by process holding it

- **Circular wait**
  - A set of \( \{P_0, P_1, \ldots, P_n\} \) waiting processes must exist
    - \( P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n \rightarrow P_0 \)
  - Implies hold-and-wait
**Deadlock Prevention**

Ensure that **one** of the necessary conditions for deadlocks **cannot** occur:

1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular wait
Mutual exclusion must hold for non-sharable resources, but …

- Sharable resources do not require mutually exclusive access
  - *Cannot be involved* in a deadlock
- A process never needs to wait for sharable resource
  - Read-only files
- Some resources are *intrinsically nonsharable*
  - So denying mutual exclusion often not possible

Deadlock Prevention: Ensure hold-and-wait never occurs in the system [*Strategy 1*]

- Process must request and be allocated all its resources *before* execution
  - Resource requests must precede other system calls
- E.g. copy data from DVD drive, sort file & print
  - Printer needed only at the end
  - BUT process will hold printer for the *entire* execution
Deadlock Prevention: Ensure hold-and-wait never occurs in the system [Strategy 2]

- Allow a process to request resources only when it has none
  - Release all resources, before requesting additional ones

- E.g. copy data from DVD drive, store file, & print
  - First request DVD and disk file
    - Copy and release resources
  - Then request file and printer

Disadvantages of protocols doing hold-and-wait

- **Low resource utilization**
  - Resources are allocated but unused for long durations

- **Starvation**
  - If a process needs several popular resources
    - Popular resource might always be allocated to some other process
Deadlock Prevention: Eliminate the preemption constraint

- C1) If a process is holding some resources
- C2) Process requests another resource
   - Cannot be immediately allocated

- All resources currently held by process is preempted
  - Preempted resources added to list of resources process is waiting for

Deadlock Prevention: Eliminate the preemption constraint

- Process requests resources that are not currently available
  - If resources allocated to another waiting process
    - Preempt resources from the second process and assign it to the first one

- Often applied when resource state can be saved and restored
  - CPU registers and memory space
  - Unsuitable for tape drives
Deadlock Prevention: Eliminating Circular wait

- Impose **total ordering** of all resource types
  - Assign each resource type a unique number
  - One-to-one function \( F : R \rightarrow N \)
    
    \[
    \begin{align*}
    F(\text{tape drive}) &= 1; \\
    F(\text{printer}) &= 12
    \end{align*}
    \]

  1. Request resources in *increasing order*
  2. If several instances of a resource type needed?
    - Single request for all them must be issued

Requesting resources in an increasing order of enumeration

- Process initially requested \( R_i \)
- This process can now request \( R_j \) ONLY IF
  \[
  F(R_j) > F(R_i)
  \]
- Alternatively, process requesting \( R_j \) must have released resources \( R_i \) such that
  \[
  F(R_i) \geq F(R_j)
  \]
- Eliminates circular wait
Hierarchy of resources and deadlock prevention

- Hierarchy by itself does not prevent deadlocks
  - Developed programs must follow ordering

- F based on order of usage of resources
  - Tape drive needed before printing
    - F(tape drive) < F(printer)

Deadlock Prevention: Summary

- Prevent deadlocks by restraining how requests are made
  - Ensure at least 1 of the 4 conditions cannot occur

- Side effects:
  - Low device utilization
  - Reduced system throughput
Dining Philosophers:
Deadlock prevention (1)

- **Mutual exclusion**
  - Philosophers can *share* a chopstick

- **Hold-and-wait**
  - Philosopher should release the first chopstick if it cannot obtain the second one

Dining Philosophers:
Deadlock prevention (2)

- **Preemption**
  - Philosophers can *forcibly take* each other’s chopstick

- **Circular-wait**
  - Number the chopsticks
  - Pick up chopsticks in ascending order
    - Pick the lower numbered one before the higher numbered one
Deadlock avoidance

- Require *additional* information about *how* resources are to be requested.

- Knowledge about sequence of requests and releases for processes:
  - Allows us to decide if resource allocation *could cause a future deadlock*.
  - Process P: Tape drive, then printer.
  - Process Q: Printer, then tape drive.
Deadlock avoidance:
Handling resource requests

- For each resource request:
  - Decide whether or not process should wait
    - To avoid possible future deadlock

- Predicated on:
  1. Currently available resources
  2. Currently allocated resources
  3. Future requests and releases of each process

Avoidance algorithms differ in the amount and type of information needed

- **Resource allocation state**
  - Number of available and allocated resources
  - Maximum demands of processes

- Dynamically **examine** resource allocation state
  - Ensure circular-wait cannot exist

- Simplest model:
  - Declare maximum number of resources for each type
  - Use information to avoid deadlock
Safe sequence

- **Sequence** of processes \(<P_1, P_2, \ldots, P_n>\) for the current allocation state
- Resource requests made by \(P_i\) can be satisfied by:
  - Currently available resources
  - Resources held by \(P_j\) where \(j < i\)
    - If needed resources not available, \(P_i\) can wait
  - In general, when \(P_i\) terminates, \(P_{i+1}\) can obtain its needed resources
- If no such sequence exists: system state is **unsafe**

Deadlock avoidance: Safe states

- If the system can:
  1. Allocate resources to each process in **some order**
     - Up to the **maximum** for the process
  2. Still avoid deadlock
Safe states and deadlocks

- A system is safe ONLY IF there is a **safe sequence**
- A safe state is not a deadlocked state
  - Deadlocked state is an unsafe state
  - Not all unsafe states are deadlocks

State spaces
Unsafe states

- A unsafe state *may lead* to deadlock
- **Behavior** of processes controls unsafe states
- Cannot prevent processes from requesting resources such that deadlocks occur

Example: 12 Tape drives available in the system

<table>
<thead>
<tr>
<th>Maximum Needs</th>
<th>Current Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀</td>
<td>10</td>
</tr>
<tr>
<td>P₁</td>
<td>4</td>
</tr>
<tr>
<td>P₂</td>
<td>9</td>
</tr>
</tbody>
</table>

Before T₀: 3 drives available

Safe sequence: <P₁, P₀, P₂>

- At time T₀ the system is in a safe state
- P₁ can be given 2 tape drives
- When P₁ releases its resources; there are 5 drives
- P₀ uses 5 and subsequently releases them (# 10 now)
- P₂ can then proceed
Example: 12 Tape drives available in the system

<table>
<thead>
<tr>
<th>Maximum Needs</th>
<th>Current Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀</td>
<td>10</td>
</tr>
<tr>
<td>P₁</td>
<td>4</td>
</tr>
<tr>
<td>P₂</td>
<td>9</td>
</tr>
</tbody>
</table>

At time T₁, P₂ is allocated 1 tape drive

Before T₁:
3 drives available

After T₁:
2 drives available

- At time T₁, P₂ is allocated 1 tape drive
- Only P₁ can proceed.
- When P₁ releases its resources; there are 4 drives
  - P₀ needs 5 and P₂ needs 6
  - **Mistake** in granting P₂ additional tape drive
Crux of deadlock avoidance algorithms

- Ensure that the system will always remain in a safe state
- Resource allocation request granted only if it will leave the system in a safe state
Claim edges

- Indicates that a process $P_i$ may request a resource $R_j$ at some time in the future.

- Representation:
  - Same direction as request
  - Dotted line

Resource allocation graph with a claim edge

![Diagram showing a resource allocation graph with a claim edge](image)
Conversion of claim edges

- When process $P_i$ requests resource $R_j$
  - Claim edge converted to a request edge

- When resource $R_j$ released by $P_i$
  - The assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$

Allocating resources

- When process $P_i$ requests resource $R_j$

- Request granted only if
  - Converting claim edge to $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in a cycle
Using the allocation graph to allocate resources safely

Using the allocation graph to allocate resources safely

October 16, 2018
Professor: SHRI DEE P PALICKARA

October 16, 2018
Professor: SHRI DEE P PALICKARA
Resource allocation graph algorithm

- Not applicable in systems with multiple resource instances

The contents of this slide-set are based on the following references
