Deadlock in Resource Sharing Environment

- A deadlock occurs when 2 or more processes/threads permanently block each other by each having a lock on a resource which the other tasks are trying to lock.

- Deadlock can occur due to:
  - Locks: Waiting to acquire locks on resources, such as objects, pages etc.
  - Sharing resources such as I/O devices printer, disks etc.

Recap Example

- 2 Threads access 2 shared variables A and B
- Variable A is protected by lock x and variable B by lock y
- Here’s what Thread 1 and Thread 2 need to do:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = A + 10</td>
<td></td>
</tr>
<tr>
<td>B = B + 20</td>
<td></td>
</tr>
<tr>
<td>A = A + B</td>
<td></td>
</tr>
<tr>
<td>A = A + 30</td>
<td></td>
</tr>
<tr>
<td>B = B + 10</td>
<td></td>
</tr>
<tr>
<td>A = A + 20</td>
<td></td>
</tr>
<tr>
<td>A = A + B</td>
<td></td>
</tr>
<tr>
<td>B = B + 30</td>
<td></td>
</tr>
</tbody>
</table>

Representing Deadlock

- Tasks (processes/threads) represented with Circles
- Resources represented with squares
- Task T1 has a lock on resource R1 indicated by the arrow from R1 to T1
- Task T1 has requested a lock on resource R2 indicated by the arrow from T1 to R2.
- Task T2 has a lock on resource R2 and has requested a lock on resource R1
- Because neither task can continue until a resource is available and neither resource can be released until a task continues, a deadlock state exists
Conditions for Deadlock

- Mutual Exclusion
  - A resource that cannot be used by more than one task at a time

- Hold and Wait
  - A task holds one resource and may request a new resource

- No preemption
  - Resources are released voluntarily after competition

- Circular wait
  - Two or more tasks form a circular chain where each task waits for a resource that the next process in the chain holds

Deadlock Handling

- Deadlock Prevention
  - Ensure deadlock does not occur by ensuring at least one of 4 conditions does not occur
  - No generic solution

- Deadlock Detection and Recovery:
  - Periodically checks on waiting tasks to determine deadlock
  - If deadlock is determined, then
    - Abort all tasks within that are deadlock – expensive
    - Abort one task at a time till deadlock is cycle is eliminated
      - Undo effects of task -> must roll-back to safe state
      - Should not starve the task by repeatedly aborting it

- Avoidance
  - By not explicitly over allocating resources
  - However this requires prior knowledge of total resources needed by each task – not practical
  - Difficult but many algorithms are proposed
    - E.g. Banker’s Algorithm

Deadlock Prevention

- No Mutual Exclusion
  - Applies to limited scenario – E.g. shared files that are read only

- No Hold and Wait
  1. Only request resources when have none
  2. Atomically acquire all resources
     - New primitive to have single lock to protect all resources
     - Problem: Could lead to starvation for others if too many resources are acquired all at once by one task

- Allow preemption
  - E.g. B waiting for something held by A, then take resource away from B and give to A
  - Must rollback i.e. undo actions by A

- No Circular wait
  - Impose ordering on resources
    - E.g. Dining Philosophers Problem
      - Even # philosopher picks up lower # fork first, and then the higher # fork
      - After eating, put down the highest # fork first, followed by the lower # fork, freeing another philosopher to grab the latter and begin eating.

Hw 3: Dining Philosopher

- Actions
  - Acquire fork, eat, Put down fork

- N philosophers
  - A philosopher is a process
  - Create using fork()

- N forks
  - Shared resources for adjacent neighbors
  - Synchronize using semaphore

- Assignment: provide solution that is
  - deadlock free
  - fair(no starvation)
Hw3: Deadlock Prevention & Fairness

- Deadlock due to Circular wait
- Soln: Impose ordering on fork access

- Fairness
  - one of the philosophers could make progress but does not because others always go first.
  - Soln: Make sure that philosopher $i$ after eating lets their neighbors eat before eating again

Hw3: Fairness technique

- Every philosopher has two locks for each of its forks, therefore every philosopher sees 4 locks
- There are 2 exclusive locks, one each for philosopher's left and right, which he/she must be successful in acquiring
- Before eating, the philosopher will acquire its exclusive locks, once a philosopher is done eating with his exclusive locks, he will unlock the other two locks (which are exclusive semaphores of his/her neighboring philosopher)
- In the program, you have to decide some philosophers will already start out with their exclusive locks, eat and then unlock the other two locks (i.e. exclusive locks of neighbor)
- You will need to think little about initial condition i.e. what values the semaphore initial counts will be.

Hw3: Input/Output

- Command line input # of philosophers, and # steps to simulate
  - ./prog 2 10
  - Step 1: Philosopher 0 picks up fork 0
  - Step 2: Philosopher 0 picks up fork 1
  - Step 3: Philosopher 0 eats
  - Step 4: Philosopher 0 puts down fork 1
  - Step 5: Philosopher 0 puts down fork 0
  - Step 6: Philosopher 1 picks up fork 1
  - Step 7: Philosopher 1 picks up fork 0
  - Step 8: Philosopher 1 eats
  - Step 9: Philosopher 1 puts down fork 0
  - Step 10: Philosopher 1 puts down fork 1

Hw3: Inter-Process Communication (IPC)

- One IPC technique is message passing
  - Communication is made by the sending of messages to recipients via a message queue
  - For hw3
    - Message: Action performed by philosopher $i$ put into the queue
    - Observer (parent process): Retrieves the message
    - Parent process should create the message and have exclusive access and should only read from the queue.
    - Child processes should open the same queue (which the parent process had created) and open it in write only mode
      - Both parent and children will set the mode to be read, write and exclusive for the owner
Deadlock Detection

- Construct a resource graph
- A graph is collection of nodes and edges
  - Node: task (circles) or resource (square)
  - Edge: Connects to nodes together with arrow indicating direction
    - Resource held by task (Resource -> Task)
    - Task wanting resource (Task -> Resource)
- If the graph contains one or more cycles then a deadlock exists
  - Use Depth First Search (DFS)
    - Start at the node and explore as far as possible along each branch before backtracking
    - If you see a node revisited again then we have found a cycle

Example with only one instance of a resource type

- Task A – G & Resources R - W
  - A holds R and want S
  - B holds nothing but wants T
  - C holds nothing but wants S
  - D holds U and wants S and T
  - E holds T and wants V
  - F holds W and wants S
  - G holds V and wants U
- Search B leads – B, T, E, V, G, U, D, T
  - See T twice, found a cycle

Example with multiple instances of a resource type

- With more than one instances of resource type, a cycle can exist but deadlock may not

Deadlock Avoidance

- Grant a resource request only if request cannot lead to a deadlock either immediately or in the near future
  - No deadlock is known as safe state
- Conservative Approach:
  - Each process declares the maximum number of resource units of each resource class that it may require
  - Determine if resource should be given
    - Based on the current allocation and available resources leads to safe state
    - If safe state is possible then actually grant the request
- Safe State
  - Construct a sequence of task completion, resource release and resource allocation events through which each task can obtain its max resources for each resource class
**Bankers Algorithm**

- Modeled after banker that gives credit to customers
- Idea is that not all customers will need their maximum credit immediately
  - Reserve lesser units than the total need
- Before actually giving credit check to see if financially safe

**Example with Single Resource class**

- System contains 10 units of resource class
- Max Need for P1 = 8, P2 = 7, P3 = 5
- Current Allocation for (P1 = 3, P2 = 1, P3 = 3)
- If process P1 makes request for one more resource unit i.e. (1, 0, 0) is the system in safe state?

---

**Banker’s Algorithm**

- To complete a process $P_i$, check for all resource class $R_k$
  \[ \text{Total}_{R_k} - \text{Total}_{\text{allocated}}_k \geq \text{Max}_{\text{need}}_{i,k} - \text{Allocated}_{\text{resources}}_{i,k} \]
  - $\text{Total}_{\text{Allocated}}_k = \sum (\text{Allocated}_{\text{resources}}_{i,k})$
- If satisfied, it simulates completion of Process $P_i$ and release of all resources allocated to it by updating $\text{Total}_{\text{allocated}}_k$ for each $R_k$
- $P_1$’s request for one resource unit leads system into safe state
  - Sequence P3, P1, P2