Using Semaphore to Share Resource

Process P();

{ A.Down();
 B.Down();
 use both resource
 B.Up();
 A.Up(); }

Process Q();

{ A.Down();
 B.Down();
 use both resource
 B.Up();
 A.Up(); }

External Semaphore A(1), B(1);

2  External Semaphore A(0), B(1);

3  External Semaphore A(0), B(0);

4  External Semaphore A(0), B(1);

5  External Semaphore A(1), B(1);
But Deadlock can Happen!

Process P();
{
  A.Down();
  B.Down();
  use both
  B.Up();
  A.Up();
}  

Process Q();
{
  B.Down();
  A.Down();
  use both resources
  A.Up();
  B.Up();
}

1  External Semaphore A(1), B(1);
2  External Semaphore A(0), B(1);
3  External Semaphore A(0), B(0);
Deadlock

- Synchronization is a live gun – we can easily shoot ourselves in the foot
  - Incorrect use of synchronization can block all processes
  - You have likely been intuitively avoiding this situation already
- More generally, processes that allocate multiple resources, generate dependencies on those resources
  - Locks, semaphores, monitors, etc., just represent the resources that they protect
- If one process tries to allocate a resource that a second process holds, and vice-versa, they can never make progress
- We call this situation deadlock, and we’ll look at:
  - Definition and conditions necessary for deadlock
  - Representation of deadlock conditions
  - Approaches to dealing with deadlock
Traffic Deadlock
Let’s Start with Resource

- A **resource** is a commodity needed by a process.
- Resources can be either:
  - **serially reusable**: e.g., CPU, memory, disk space, I/O devices, files.
    acquire → use → release
  - **consumable**: produced by a process, needed by a process; e.g., messages, buffers of information, interrupts.
    create → acquire → use (consumed)
    Resource ceases to exist after it has been used, so it is not released.
Resource (2)

- Resources can also be either:
  - preemptible: e.g., CPU, or
  - non-preemptible: e.g., tape drives.
- And resources can be either:
  - shared among several processes or
  - dedicated exclusively to a single process.
Deadlock Definition

- Deadlock is a problem that can arise:
  - When processes compete for access to limited resources
  - When processes are incorrectly synchronized

- Definition:
  - A process is **deadlocked** if it is waiting for an event that will never occur.
  - Typically, but not necessarily, more than one process will be involved together in a deadlock (the *deadly embrace*).

```
lockA->Acquire();
...
lockB->Acquire();
Process 1

lockB->Acquire();
...
lockA->Acquire();
Process 2
```
Deadlock

Mechanism for Deadlock Control?
Deadlock vs. Starvation

- Is deadlock the same as starvation (or indefinitely postponed)?
  - A process is **indefinitely postponed** if it is delayed repeatedly over a *long* period of time while the attention of the system is given to other processes. I.e., logically the process may proceed but the system never gives it the CPU in reality.
Conditions for Deadlock

- What conditions should exist in order to lead to a deadlock?
Necessary and Sufficient Conditions for Deadlock

- **Mutual exclusion**
  - Processes claim exclusive control of the resources they require

- **Wait-for condition**
  - Processes hold resources already allocated to them while waiting for additional resources

- **No preemption condition**
  - Resources cannot be removed from the processes holding them until used to completion

- **Circular wait condition**
  - A circular chain of processes exists in which each process holds one or more resources that are requested by the next process in the chain
Deadlock can be described using a resource allocation graph (RAG).

The RAG consists of a set of vertices $P = \{P_1, P_2, \ldots, P_n\}$ of processes and $R = \{R_1, R_2, \ldots, R_m\}$ of resources:
- A directed edge from a process to a resource, $P_i \rightarrow R_j$, means that $P_i$ has requested $R_j$.
- A directed edge from a resource to a process, $R_i \rightarrow P_i$, means that $R_j$ has been allocated by $P_i$.
- Each resource has a fixed number of units.

If the graph has no cycles, deadlock cannot exist.
If the graph has a cycle, deadlock may exist.
Resource Allocation Graph

<table>
<thead>
<tr>
<th>Resource</th>
<th>Process</th>
<th>Resource Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Resource" /></td>
<td><img src="image2.png" alt="Process" /></td>
<td><img src="image3.png" alt="Resource Type" /></td>
</tr>
</tbody>
</table>

1 Process, 2 Resources of same Type

- Process requests resource
- Process is assigned resource
- Process releases resource
Deadlock Model

```
Resource  Process  Resource Type

2 Processes
2 resources

Processes request 2 resources each

Deadlock
Cycle in resource graph

Deadlock may not occur if there are enough resources
Cycle in resource graph
```
A Simpler Case

- If all resources are single unit and all processes make single requests, then we can represent the resource state with a simpler waits-for graph (WFG).
- The WFG consists of a set of vertices $P = \{P_1, P_2, \ldots, P_n\}$ of processes.
  - A directed edge $P_i \rightarrow P_j$ means that $P_i$ has requested a resource that $P_j$ currently holds.
- If the graph has no cycles, deadlock cannot exist.
- If the graph has a cycle, deadlock exists.
Deadlock Issues

- **Prevention**
  - design a system in such a way that **deadlocks cannot occur**, at least with respect to serially reusable resources.

- **Avoidance**
  - impose less stringent conditions than for prevention, allowing the possibility of deadlock, but sidestepping it as it approaches.

- **Detection**
  - in a system that allows the possibility of deadlock, **determine if deadlock has occurred**, and which processes and resources are involved.

- **Recovery**
  - after a deadlock has been detected, **clear the problem**, allowing the deadlocked processes to complete and the resources to be reused. Usually involves destroying the affected processes and starting them over.
The Ostrich Algorithm

- Don’t do anything, simply restart the system (stick your head into the sand, pretend there is no problem at all).
- Rational: make the common path faster and more reliable
  - Deadlock prevention, avoidance or detection/recovery algorithms are expensive
  - if deadlock occurs only rarely, it is not worth the overhead to implement any of these algorithms.
So why do we still lean about deadlocks?

- How about aircraft control systems?
- How about the software running in your car?
How do we prevent deadlocks?
  - You can use real life analogies
  - If you cannot give an answer, you are like an “ostrich”?
Deadlock Prevention: Havender's Algorithms

- Break one of the deadlock conditions.
  - **Mutual exclusion**
    - Solution: Avoid assigning a resource when it is not absolutely necessary. (print spool)
  - **Hold-and-Wait condition**
    - Solution: Force each process to request all required resources at once. It cannot proceed until all resources have been acquired.
  - **No preemption condition**
    - Solution: If a process holding some reusable resources makes a further request which is denied, and it wishes to wait for the new resources to become available, it must release all resources currently held and, if necessary, request them again along with the new resources. Thus, resources are removed from a process holding them.
    - Remember wait() in monitor?
  - **Circular wait condition**
    - Solution: All resource types are numbered. Processes must request resources in numerical order; if a resource of type \( R \) is held, the only resources which can be requested must be of types \( R \).
Two-Phase Locking

- **Phase One**
  - process tries to lock all records it needs, one at a time
  - if needed record found locked, start over
  - (no real work done in phase one)

- If phase one succeeds, it starts second phase,
  - performing updates
  - releasing locks

- Note similarity to requesting all resources at once
Break Circular Wait Condition

- **Method 1**: Request one resource at a time. Release the current resource when request the next one.
- **Method 2**: Global ordering of resources
  - Requests have to made in increasing order
  - `Req(resource1), req(resource2)...
  - Why no circular wait?
### Summary: Deadlock Prevention

<table>
<thead>
<tr>
<th>condition</th>
<th>How to break it</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutual Exclusion</td>
<td>Spool everything</td>
</tr>
<tr>
<td>Hold and wait</td>
<td>Request all resources initially</td>
</tr>
<tr>
<td>No preemption</td>
<td>Take resources away</td>
</tr>
<tr>
<td>Circular wait</td>
<td>Order resources numerically</td>
</tr>
</tbody>
</table>
Deadlock Avoidance

- The system needs to know the resource requirement ahead of time
- Banker Algorithm (Dijkstra, 1965)
Banker’s Algorithm

- The Banker’s Algorithm is the classic approach to deadlock avoidance for resources with multiple units

1. Assign a credit limit to each customer (process)
   - Maximum credit claim must be stated in advance

2. Reject any request that leads to a dangerous state
   - A dangerous state is one where a sudden request by any customer for the full credit limit could lead to deadlock
   - A recursive reduction procedure recognizes dangerous states
Safe State and Unsafe State

- **Safe State**
  - there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately
  - From safe state, the system can guarantee that all processes will finish

- **Unsafe state: no such guarantee**
  - Not a deadlocked state
  - Some process may be able to complete
How to Compute Safety

Given:
n kinds of resources
p processes
Set P of processes
struct {resource needs[n], owns[n]} ToDo[p]
resource_available[n]

while there exists a p in P such that
for all i (ToDo[p].needs[i] < available[i])  // pick one that can be satisfied
{ do for all i  //return the allocated resources back to system
    available[i] += ToDo[p].owns[i];
    P = P - p;
}
If P is empty then system is safe
Need = Max - Alloc

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>2 0 0</td>
<td>3 2 2</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>3 0 0</td>
<td>9 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td></td>
</tr>
</tbody>
</table>

If P1 requests max resources, can complete
Is Allocation (1 0 2) to P1 Safe?

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>7 4 3</td>
<td>10 5 5</td>
</tr>
<tr>
<td>P1</td>
<td>2 0 0</td>
<td>3 2 2</td>
<td>1 2 2</td>
<td>3 3 2</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 0</td>
<td>9 0 2</td>
<td>6 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

If P1 requests max resources, can complete
Is Allocation \((1 \ 0 \ 2)\) to \(P1\) Safe?

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>7 4 3</td>
<td>10 5 5</td>
</tr>
<tr>
<td>P1</td>
<td>3 0 0</td>
<td>3 2 2</td>
<td>0 2 2</td>
<td>2 3 2</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 0</td>
<td>9 0 2</td>
<td>6 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

If \(P1\) requests max resources, can complete
And Run Safety Test: Group Discussion

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>P0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>P1</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
Allocate to P1, Then

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>7 4 3</td>
<td>10 5 5</td>
</tr>
<tr>
<td>P1</td>
<td>3 2 2</td>
<td>3 2 2</td>
<td>0 0 0</td>
<td>2 1 0</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 0</td>
<td>9 0 2</td>
<td>6 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>
### Release - P1 Finishes

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>7 4 3</td>
<td>10 5 5</td>
</tr>
<tr>
<td>P1</td>
<td>0 0 0</td>
<td>3 2 2</td>
<td>3 2 2</td>
<td>5 3 2</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 0</td>
<td>9 0 2</td>
<td>6 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2 1 1</td>
<td>2 2 2</td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 2</td>
<td>4 3 3</td>
<td>4 3 1</td>
<td></td>
</tr>
</tbody>
</table>

Now P3 can acquire max resources and release.
Release - P3 Finishes

Now P4 can acquire max resources and release
Release - P4 Finishes

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>7 4 3</td>
<td>10 5 5</td>
</tr>
<tr>
<td>P1</td>
<td>0 0 0</td>
<td>3 2 2</td>
<td>3 2 2</td>
<td>7 4 5</td>
</tr>
<tr>
<td>P2</td>
<td>3 0 0</td>
<td>9 0 2</td>
<td>6 0 2</td>
<td>7 4 5</td>
</tr>
<tr>
<td>P3</td>
<td>0 0 0</td>
<td>2 2 2</td>
<td>2 2 2</td>
<td>7 4 5</td>
</tr>
<tr>
<td>P4</td>
<td>0 0 0</td>
<td>4 3 3</td>
<td>4 3 3</td>
<td>7 4 5</td>
</tr>
</tbody>
</table>

Now P2 can acquire max resources and release
### Release - P2 Finishes

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0</td>
<td>0 1 0</td>
<td>7 5 3</td>
<td>7 4 3</td>
<td>10 5 5</td>
</tr>
<tr>
<td>P1</td>
<td>0 0 0</td>
<td>3 2 2</td>
<td>3 2 2</td>
<td>10 4 5</td>
</tr>
<tr>
<td>P2</td>
<td>0 0 0</td>
<td>9 0 2</td>
<td>9 0 2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>0 0 0</td>
<td>2 2 2</td>
<td>2 2 2</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>0 0 0</td>
<td>4 3 3</td>
<td>4 3 3</td>
<td></td>
</tr>
</tbody>
</table>

Now P0 can acquire max resources and release
So P1 Allocation (1 0 2) Is Safe

<table>
<thead>
<tr>
<th>Process</th>
<th>Alloc</th>
<th>Max</th>
<th>Need</th>
<th>Available</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>P0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>P1</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

CSE 120 – Scheduling and Deadlock
Video

- https://www.youtube.com/watch?v=olXj9FfSyxY
- https://www.youtube.com/watch?v=47_shKlKv5w
- https://www.youtube.com/watch?v=P-e0teUUMqc
Detection and Recovery

- Detection and recovery
  - If we don’t have deadlock prevention or avoidance, then deadlock may occur
  - In this case, we need to detect deadlock and recover from it

- To do this, we need two algorithms
  - One to determine whether a deadlock has occurred
  - Another to recover from the deadlock

- Possible, but expensive (time consuming)
  - Implemented in VMS
  - Run detection algorithm when resource request times out
Deadlock Detection

- Detection
  - Traverse the resource graph looking for cycles
  - If a cycle is found, preempt resource (force a process to release)

- Expensive
  - Many processes and resources to traverse

- Only invoke detection algorithm depending on
  - How often or likely deadlock is
  - How many processes are likely to be affected when it occurs
Deadlock Recovery

Once a deadlock is detected, we have two options…

1. Abort processes
   - Abort all deadlocked processes
     - Processes need start over again
   - Abort one process at a time until cycle is eliminated
     - System needs to rerun detection after each abort

2. Preempt resources (force their release)
   - Need to select process and resource to preempt
   - Need to rollback process to previous state
   - Need to prevent starvation
Deadlock Summary

- Deadlock occurs when processes are waiting on each other and cannot make progress
  - Cycles in Resource Allocation Graph (RAG)

- Deadlock requires four conditions
  - Mutual exclusion, hold and wait, no resource preemption, circular wait

- Four approaches to dealing with deadlock:
  - Ignore it – Living life on the edge
  - Prevention – Make one of the four conditions impossible
  - Avoidance – Banker’s Algorithm (control allocation)
  - Detection and Recovery – Look for a cycle, preempt or abort