Deadlock

✓ Permanent blocking of a set of processes that either compete for system resources or communicate with each other
✓ Involve conflicting needs for resources by two or more processes
Example of Deadlock

A Required

B Required

Progress of Q

Release A
Release B
Get A
Get B

Progress of P

Get A Get B Release A Release B

1 2

3

deadlock inevitable

P and Q want A

P and Q want B

5 6

A Required

B Required
Example of No Deadlock

A Required

B Required

P and Q want A

P and Q want B

Progress of Q

Progress of P

Get A

Get B

Release A

Release B

Required
Reusable, Nonsharable Resources

- Used by one process at a time and not depleted by that use
- Processes obtain resources that they later release for reuse by other processes
- Processor time, I/O channels, main and secondary memory, files, databases, and semaphores
- Deadlock occurs if each process holds one resource and requests the other
Example of Deadlock

✓ Space is available for allocation of 200K bytes, and the following sequence of events occur

P1

... Request 80K bytes;
... Request 60K bytes;

P2

... Request 70K bytes;
... Request 80K bytes;

✓ Deadlock occurs if both processes progress to their second request
Consumable Resources

- Created (produced) and destroyed (consumed) by a process
- Interrupts, signals, messages, and information in I/O buffers
- Deadlock may occur if a Receive message is blocking
- May take a rare combination of events to cause deadlock
Example of Deadlock

✓ Deadlock occurs if receive is blocking

```
P1

...  
Receive(P2);  
...  
Send(P2);

```

```
P2

...  
Receive(P1);  
...  
Send(P1);
```
Conditions for Deadlock

✔ **Mutual exclusion**
  - only one process may use a resource at a time

✔ **Hold-and-wait**
  - a process is allowed to hold allocated resources while awaiting assignment of others

✔ **No preemption**
  - no resource can be forcibly removed from a process holding it
Conditions for Deadlock

- **Circular wait**
  - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain
  - Necessary for deadlock to happen
  - Sufficient under certain circumstances

- Other conditions are necessary but not sufficient for deadlock
Circular Wait

- Process P1 requests Resource A
- Process P2 requests Resource B
- Resource A is held by Process P2
- Resource B is held by Process P1

Diagram shows a circular wait scenario where two processes are waiting for each other's resources, leading to a deadlock.
Deadlock Prevention

✓ Disallowing “Mutual Exclusion”
   ▪ cannot be achieved, due to the nature of non-sharable resources

✓ Disallowing “Hold-and-Wait”

✓ Require that a process request all its required resources *at once*
   Block the process until all requests can be granted simultaneously

✓ Problem:
   o process can be held up for a long time waiting for all its requests
   o starvation is possible
Deadlock Prevention

✓ Disallowing “No Preemption”
  - If a process is denied a further request, the process must release the original resources.
  - If a process cannot obtain a resource, the process may have to release the resources it already holds. Process must have capability to restore these resources to current state.
  - Practical only only when the state can be easily saved and restored later, such as the CPU or main memory.
Deadlock Prevention

✓ Disallowing “Circular Wait”
  - define a linear ordering for resources
  - once a resource is obtained, only those resources that have id number higher can be obtained (if a process requests a resource with lower id number than what it has obtained so far, that request is denied)
  - Problem: may deny resources for no reason other than the “wrong” id number. Low resource utilization.
Deadlock Avoidance

✅ Do not start a process if its demands might lead to deadlock

✅ Do not grant an incremental resource request to a process if this allocation might lead to deadlock

✅ Not necessary to preempt and rollback processes
Deadlock Avoidance

- Maximum resource requirement must be stated in advance
- Processes under consideration must be independent; no synchronization requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources
Deadlock Avoidance: The Banker’s Algorithm

✓ Assume: M resources $R_1, \ldots, R_M$
  o system has several identical copies of each resource: $Q_1, \ldots, Q_M$ (maximums/resource)
  o a process can ask for several copies of each resource type; doesn’t care which specific copies are given as long as they are of the requested type

✓ State: a particular allocation of resources to processes:

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Q1=9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Q2=7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Q3=4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Deadlock Avoidance: Banker’s Algorithm

.safe state:
- state where we can order the remaining processes so they can run to completion without a deadlock.

A safe state test:
// Let L be the list of unfinished processes
while L is non-empty do
    if can find process P in L whose outstanding requests can be satisfied using available resources then {
        delete P from L
        add all resources held by P to the pool of available resources
    } else return (unsafe)
return (safe)

This is only a sufficient - not a necessary, condition for safety!
Deadlock Avoidance: Banker’s Algorithm

✓ Request granting algorithm

```plaintext
req(P, Qty, Rsrc):

if Qty + allocated(P, Rsrc) > limit(Rsrc) then
    abort P;
    tentatively allocate request;
if safe(resulting state) then
    grant request;
else
    suspend P on Rsrc;
```
Deadlock Detection

✓ Operating system checks for deadlock

✓ When:
  ▪ Check at resource request time
    o early detection of deadlock
    o frequent checks consume processor time
  ▪ Check periodically
    o deadlocks stay undetected between checks
Strategies once Deadlock Detected

✓ Abort all deadlocked processes
✓ Back up each deadlocked process to some previously defined checkpoint, and restart all processes
  - original deadlock may occur once again
✓ Successively abort deadlocked processes until deadlock no longer exists
✓ Successively preempt resources until deadlock no longer exists (if resource preemption is feasible and cost-effective)
Selection Criteria
Deadlocked Processes

- Least amount of processor time consumed so far (so the least amount of work will be wasted)
- Least number of lines of output produced so far (so cheaper to preempt file resources)
- Most estimated time remaining (presumably, this process is most deadlock-prone)
- Least total resources allocated so far
- Lowest priority
Deadlock Detection Algorithms

✓ Single instance of each resource
  ▪ draw a resource request graph as follows:
    o arc Proc --> Res exists if Proc requested Res
    o arc Proc <-- Res exists if Proc holds Res
  ▪ The system has a deadlock iff the resource request graph has a cycle
    o Why?
    o Multiple resource instances:
      • is the cycle condition necessary for the existence of a deadlock?
      • Is it sufficient?
Deadlock Detection Algorithms

✓ Multiple resource instances similar to the safety test:

// Let L be the list of all unfinished processes
while nonempty(L) do{
    if found a process P in L all of whose requests can be satisfied with available resources do {
        delete P from L;
        add all resources held by P to the pool of available resources;
    } else return(deadlock);
} else return(nodedlock);

✓ The algorithm ensures that if none of the processes issues additional resource requests, then there is no deadlock.
Dining Philosophers Problem

Illustrates:
- deadlocks
- starvation
With starvation & deadlock:

```
repeat
  <think>
  wait(fork[I]);
  wait(fork[(I+1) mod 5]);
  <eat>
  signal(fork[I]);
  signal(fork[(I+1) mod 5]);
forever
```

No starvation or deadlock:

```
repeat
  <think>
  wait(room);
  wait(fork[I]);
  wait(fork[(I+1) mod 5]);
  <eat>
  signal(fork[(I+1) mod 5]);
  signal(fork[I]);
  signal(room);
forever
```
**Interprocess Concurrency Mechanisms**

- **Pipes**
  - buffer allowing two processes to communicate
  - queue written by one process and read by another
  - operating system enforces mutual exclusion for writing and reading the pipe
  - write requests are immediately executed if there is room in the pipe, otherwise the process is blocked
  - read request is blocked if attempts to read more bytes than currently available in the pipe
**Inter-process Concurrency Mechanisms**

**Messages**
- block of text with accompanying type
- receiver can either retrieve messages in FIFO order or by type
- process suspends when trying to send a message to a queue that is full
- process suspends when reading from an empty queue
- process trying to read a certain type of messages *fails, not suspended*
Inter-process Concurrency Mechanisms

✓ Shared memory

- common block of virtual memory shared by multiple processes
- fast form of interprocess communication
- mutual exclusion must be provided by the processes, not the operating system
Inter-process Concurrency Mechanisms

✓ **Semaphores**
  - wait and signal and **much** more
  - operating system handles all these requests

✓ **Signals (UNIX-derived OS only)**
  - software mechanism that informs a process of the occurrence of asynchronous events
  - e.g. *interrupt process, quit process, kill process, floating point exception, ...*
Solaris *Thread* Synchronization Primitives

- **Mutual exclusion lock**
  - prevents more than one thread from proceeding when the lock is acquired

- **Semaphores**
  - used for incrementing and decrementing

- *Unlike the general synchronization primitives (which operate on resources shared by all threads of a process), these synchronization resources aren’t shared by threads in the same process!*
Multiple readers, single writer locks
- multiple threads have simultaneous read-only access
- only one thread has access for writing

Condition variables
- used to wait until a particular condition is true
Windows NT Concurrency Mechanisms

✓ Synchronization Objects
  - process
  - thread
  - file
  - console input
  - file change notification
  - mutex
  - semaphore (counting)
  - event
  - waitable timer

◆ Each synchronization object can be in either \textit{signaled} or \textit{un-signaled} state

◆ \textbf{Un-signaled state:}
  - thread can be suspended on synchronization objects that are un-signaled

◆ \textbf{Signaled state:}
  - when object is in signaled state, all threads waiting on this object wake up