Concurrency: Deadlock and Starvation

Chapter 6

What is 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

• No efficient solution
Deadlock Illustration

Resources: Quadrants a, b, c, d

Joint Progress Diagram
Reusable Resources

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

Another Example of Deadlock

• Space is available for allocation of 200Kbytes, and the following sequence of events occur

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

- Created (produced) and destroyed (consumed)
- Examples:
  - Interrupts, signals, messages, and info in I/O buffers
- Deadlock may occur if a Receive message is blocking

```
P1
  ...  
  Receive(P2);
  ...  
  Send(P2, M1);

P2
  ...  
  Receive(P1);
  ...  
  Send(P1, M2);
```

Resource Allocation Graphs

- Directed graph that depicts a state of the system of resources and processes

(a) Resource is requested
(b) Resource is held
Resource Allocation Graphs

![Diagram of Resource Allocation Graphs]

3 units of Ra

2 units of Rb

Figure 6.5  Examples of Resource Allocation Graphs

Conditions for Deadlock

1. Mutual exclusion
   - Only one process may use a resource at a time
2. Hold-and-wait
   - A process may hold allocated resources while awaiting assignment of others
3. No preemption
   - No resource can be forcibly removed from a process holding it
Conditions for Deadlock

4. 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.

Possibility of Deadlock

- Mutual Exclusion
- No preemption
- Hold and wait
Existence of Deadlock

- Mutual Exclusion
- No preemption
- Hold and wait
- Circular wait

Three Solutions to Deadlock

#1: Mr./Ms. Conservative (Prevention)

“We had better not allocate if it could ever cause deadlock”

Process waits until all needed resource free
Resources underutilized
Three Solutions to Deadlock …

#2: Mr./Ms. Prudent *(Avoidance)*

"If resource is free and with its allocation we can still guarantee that everyone will finish, use it."

Better resource utilization
Process still waits

Three Solutions to Deadlock…

#3: Mr./Ms. Liberal *(Detection/Recovery)*

"If it’s free, use it -- why wait?"

Good resource utilization, minimal process wait time
Until deadlock occurs….
Names for The Three Methods

1) **Deadlock Prevention**
   - Design system so possibility of deadlock avoided *a priori*

2) **Deadlock Avoidance**
   - Design system so that if a resource request is made that *could* lead to deadlock, then block requesting process.
   - Requires knowledge of future resource requests by processes

3) **Deadlock Detection and Recovery**
   - Algorithm to detect deadlock
   - Recovery scheme

---

**Deadlock Prevention**

**Deny one of the 4 necessary conditions**

- Mutual Exclusion
- No preemption
- Hold and wait
- Circular wait
Deadlock Prevention

• **Do not allow “Mutual Exclusion”**
  – Use only sharable resources

  => Impossible for practical systems

Deadlock Prevention …

• **Prevent “Hold and Wait”**
  (a) Preallocation - process must request and be allocated all of its required resources before it can start execution

  (b) Process must release all of its currently held resources and re-request them along with request for new resources*

  => Very inefficient

  => Can cause "indefinite postponement": jobs needing lots of resources may never run
Deadlock Prevention …

- **Allow “Resource Preemption”**
  - Allowing one process to acquire exclusive rights to a resource currently being used by a second process

  => Some resources can not be preempted without detrimental implications (e.g., printers, tape drives)

  => May require jobs to restart

Deadlock Prevention …

- **Prevent Circular Wait**
  - Order resources and
  - Allow requests to be made only in an increasing order
Preventing Circular Wait

Impose an ordering on Resources:

<table>
<thead>
<tr>
<th>Process: A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>Request: W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C D</td>
<td>W X Y Z</td>
<td>X Y Z W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A / W

After first 4 requests:

<table>
<thead>
<tr>
<th>D / Z</th>
<th>B / X</th>
</tr>
</thead>
<tbody>
<tr>
<td>C / Y</td>
<td></td>
</tr>
</tbody>
</table>

Process D cannot request resource W without voluntarily releasing Z first

Problems with Linear Ordering Approach

1. Adding a new resource that upsets ordering requires all code ever written for system to be modified!

2. Resource numbering affects efficiency

   => A process may have to request a resource well before it needs it, just because of the requirement that it must request resources in ascending sequence
Deadlock Avoidance

• OS never allocates resources in a way that could lead to deadlock
  => Processes must tell OS in advance how many resources they will request

• Process Initiation Denial
  – Process is started only if maximum claim of all current processes plus those of the new process can be met.

• Resource Allocation Denial
  – Do not grant request if request might lead to deadlock

Resource Allocation Denial: Banker’s Algorithm

• Banker's Algorithm runs each time:
  – a process requests resource - Is it Safe?
  – a process terminates - Can I allocate released resources to a suspended process waiting for them?

• A new state is safe if and only if every process can complete after allocation is made
  => Make allocation, then check system state and de-allocate if safe/unsafe
Definition: Safe State

• State of a system
  – An enumeration of which processes hold, are waiting for, or might request which resources

• Safe state
  – No process is deadlocked, and there exists no possible sequence of future requests in which deadlock could occur.
  or alternatively,
  – No process is deadlocked, and the current state will not lead to a deadlocked state

Deadlock Avoidance

Safe State:

<table>
<thead>
<tr>
<th></th>
<th>Current Loan</th>
<th>Max Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Process 2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Process 3</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

Available = 2
## Deadlock Avoidance

### Unsafe State:

<table>
<thead>
<tr>
<th>Current Loan</th>
<th>Max Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 1</td>
<td>8</td>
</tr>
<tr>
<td>Process 2</td>
<td>2</td>
</tr>
<tr>
<td>Process 3</td>
<td>1</td>
</tr>
</tbody>
</table>

Available = 1

### Safe to Unsafe Transition

**Current Safe State:**

<table>
<thead>
<tr>
<th>Current Loan</th>
<th>Maximum Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 1</td>
<td>1</td>
</tr>
<tr>
<td>Process 2</td>
<td>4</td>
</tr>
<tr>
<td>Process 3</td>
<td>5</td>
</tr>
</tbody>
</table>

Available = 2

Suppose Process 3 requests and gets one more resource

<table>
<thead>
<tr>
<th>Current Loan</th>
<th>Maximum Need</th>
</tr>
</thead>
<tbody>
<tr>
<td>User1</td>
<td>1</td>
</tr>
<tr>
<td>User2</td>
<td>4</td>
</tr>
<tr>
<td>User3</td>
<td>6</td>
</tr>
</tbody>
</table>

Available = 1
Essence of Banker's Algorithm

- Find an allocation schedule satisfying maximum claims that allows to complete jobs
  => Schedule exists iff safe

- Method: "Pretend" you are the CPU.
  1. Scan table (PCB?) row by row and find a job that can finish
  2. Add finished job's resources to number available.

Repeat 1 and 2 until
- all jobs finish (safe), or
- no more jobs can finish, but some are still “waiting” for their maximum claim (resource) request to satisfied (unsafe)

Banker's Algorithm

Constants

```
int N {number of processes}
int Total_Units
int MaximumNeed[i]
```

Variables

```
int i {denotes a process}
int Available
int CurrentLoan[i]
boolean Cannot_Finish[i]
```

Function

```
Claim[i] = MaximumNeed[i] - CurrentLoan[i];
```
Banker's Algorithm

Begin
Available = Total_Units;

For i = 1 to N Do
Begin
Available = Available - CurrentLoan[i];
Cannot_Finish[i] = TRUE;
End;
i = 1;

while (i <= N) Do
begin
If (Cannot_Finish[i] AND Claim[i] <= Available)
Then Begin
Cannot_Finish[i] = False;
Available = Available + CurrentLoan[i];
i = 1;
End;
Else i = i+1;
End;

If (Available == Total_Units)
Then Return (SAFE)
Else Return (UNSAFE);
End;

Banker's Example #1

Total_Units = 10 units
N = 3 processes

Process: 1 2 3 1
Request: 2 3 4 1

Can the fourth request be satisfied?

<table>
<thead>
<tr>
<th>Process</th>
<th>Current Loan</th>
<th>Maximum Need</th>
<th>Claim</th>
<th>Cannot Finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Available =
i =
### Banker's Example #2

Total Units = 10 units  
N = 3 processes  

<table>
<thead>
<tr>
<th>Process</th>
<th>Current Loan</th>
<th>Maximum Need</th>
<th>Claim</th>
<th>Cannot Finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Can the fourth request by satisfied? 

<table>
<thead>
<tr>
<th>Process</th>
<th>Current Loan</th>
<th>Maximum Need</th>
<th>Claim</th>
<th>Cannot Finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Available =  

| i = |  

### Determination of a Safe State  
Multi-Resource Scenario

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Claim matrix C  

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Allocation matrix A  

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Resource vector R  

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Available vector V  

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Initial state  

Is the resulting state (above) safe?  
Is \( C[*] - A[*] \) \( \leq \) \( V[*] \)?  

P2 -> P1 -> P3 -> P4
Determination of an Unsafe State
Multi-resource Scenario

<table>
<thead>
<tr>
<th>Claim matrix C</th>
<th>Allocation matrix A</th>
<th>C − A</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 3 2 2</td>
<td>P1 1 0 0</td>
<td>P1 2 2 2</td>
</tr>
<tr>
<td>P2 6 1 3</td>
<td>P2 5 1 1</td>
<td>P2 1 0 2</td>
</tr>
<tr>
<td>P3 3 1 4</td>
<td>P3 2 1 1</td>
<td>P3 1 0 3</td>
</tr>
<tr>
<td>P4 4 2 2</td>
<td>P4 0 0 2</td>
<td>P4 4 2 0</td>
</tr>
</tbody>
</table>

(a) Initial state

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Resource vector R

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Available vector V

Determination of an Unsafe State

<table>
<thead>
<tr>
<th>Claim matrix C</th>
<th>Allocation matrix A</th>
<th>C − A</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 3 2 2</td>
<td>P1 2 0 1</td>
<td>P1 1 2 1</td>
</tr>
<tr>
<td>P2 6 1 3</td>
<td>P2 5 1 1</td>
<td>P2 1 0 2</td>
</tr>
<tr>
<td>P3 3 1 4</td>
<td>P3 2 1 1</td>
<td>P3 1 0 3</td>
</tr>
<tr>
<td>P4 4 2 2</td>
<td>P4 0 0 2</td>
<td>P4 4 2 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Resource vector R

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Available vector V

(b) P1 requests one unit each of R1 and R3
Deadlock Avoidance Logic

(a) global data structures

```c
struct state {
    int resource[n];
    int available[n];
    int claim[n][n];
    int alloco[n][n];
}
```

If (alloc[i,*] + request[*] > claim[i,*]) /* total request > claim*/
    < error >;
else if (request[*] > available[*])
    < suspend process >;
else /* simulate alloc */
    < define newstate by>
    alloc[i,*] = alloc[i,*] + request[*];
    available[*] = available[*] - request[*] >;
if (safe (newstate))
    < carry out allocation >;
else
    < restore original state >;
    < suspend process >;

(b) resource alloc algorithm

Deadlock Avoidance Logic

```c
boolean safe (state S) {
    int currentavail[n];
    process rest(<number of processes>);
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible) {
        <find a process P_k in rest such that>
        claim [k,*] - alloc [k,*] <= currentavail;>
        if (found) /* simulate execution of P_k */
            currentavail = currentavail + alloc [k,*];
            rest = rest - {P_k};
        else
            possible = false;
    }
    return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)
Banker's Algorithm: Summary

(+) PRO's:

😊 Deadlock never occurs.

😊 More flexible & more efficient than deadlock prevention. (Why?)

(-) CON's:

😊 Must know max use of each resource when job starts.

=> No truly dynamic allocation

😊 Process might block even though deadlock would never occur

Deadlock Detection

Allow deadlock to occur, then recognize that it exists

• Run deadlock detection algorithm whenever locked resource is requested

• Could also run detector in background
Deadlock Detection

Set $\text{Avail}^* = \text{Avail}[*]

Remove process $i$ from consideration if:

(a) $\text{Alloc} [i, *] = 0$, or

(b) $\text{Request} [i, *] \leq \text{Avail}^*[*]

Add $\text{Alloc} [I, *]$ to $\text{Avail}^*[*]

Processes not removed from consideration are blocked

Strategies Once Deadlock Detected

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint, and restart all process
  - Hoping alternate request sequence (non-determinism)
  - However, original deadlock may still occur
- Successively abort deadlocked processes until deadlock no longer exists
  - Free up needed resources
Selection Criteria Aborting Deadlocked Processes

- Least amount of processor time consumed so far
- Least number of lines of output produced so far
- Most estimated time remaining
- Least total resources allocated so far
- Lowest priority

Strengths and Weaknesses of the Strategies

Table 6.1 Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems (ESLO80)

<table>
<thead>
<tr>
<th>Approach</th>
<th>Resource Allocation Policy</th>
<th>Different Schemes</th>
<th>Major Advantages</th>
<th>Major Disadvantages</th>
</tr>
</thead>
</table>
| Prevention | Conservative; undercommits resources | Requesting all resources at once | -更能有效将进程映射到单一进程组 | -过程间通信延迟 
- 未来资源需求必须标定为动态资源 |
| Prevention | Preemption | -资源分配按需替换 | -资源分配在系统执行中可动态调整 | -允许资源动态重分配 |
| Prevention | Resource ordering | -灵活执行 | -资源分配延迟 | -资源分配延迟 |
| Avoidance | Midway between detection and prevention | Manipulate to find at least one safe path | -没有预设 | -未来资源需求必须标定为动态资源 |
| Detection | Very liberal; requested resources are granted where possible | Invoke periodically to test for deadlock | -降低延迟过程初始化 | -资源分配可被锁定长期 |

48
Dining Philosophers Problem

```c
/* program diningphilosophers */
semaphore fork[5] = (1);
int i;
void philosopher(int i)
{
    while (true)
    {
        think();
        wait(fork[i]);
        wait(fork[(i+1) mod 5]);
        eat();
        signal(fork[(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin(philosopher(0), philosopher(1), philosopher(2),
             philosopher(3), philosopher(4));
}
```

Each Philosopher request/gets fork(i)...
Deadlock
Dining Philosophers Problem

```c
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true)
    {
        think();
        wait (room);
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal (fork [(i+1) mod 5]);
        signal (fork[i]);
        signal (room);
    }
}

void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
               philosopher (3), philosopher (4));
}
```

Limit number of Philosophers in dining room... No Deadlock

Dining Philosophers: Monitor Solution

First philosopher entering monitor is guaranteed to get both forks....

Appropriate waiting philosopher “woken” up