Chapter 6
Concurrency: Deadlock and Starvation
Deadlock

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- No efficient solution
- Involve conflicting needs for resources by two or more processes
Deadlock in Traffic

(a) Deadlock possible

(b) Deadlock

Figure 6.1 Illustration of Deadlock
Non-deadlock - Joint Progress Diagram

Figure 6.3 Example of No Deadlock [BACO03]
Deadlock in a Computer – Fatal Region

Figure 6.2 Example of Deadlock
Deadlock Definition

• Formal definition:
  A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

• Usually the event is release of a currently held resource

• None of the processes can …
  – run
  – release resources
  – be awakened
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
Reusable Resources

• 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 dedicated resource and requests another held by another process
# Reusable Resources

## Process P

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_0$</td>
<td>Request (D)</td>
</tr>
<tr>
<td>$p_1$</td>
<td>Lock (D)</td>
</tr>
<tr>
<td>$p_2$</td>
<td>Request (T)</td>
</tr>
<tr>
<td>$p_3$</td>
<td>Lock (T)</td>
</tr>
<tr>
<td>$p_4$</td>
<td>Perform function</td>
</tr>
<tr>
<td>$p_5$</td>
<td>Unlock (D)</td>
</tr>
<tr>
<td>$p_6$</td>
<td>Unlock (T)</td>
</tr>
</tbody>
</table>

## Process Q

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>Request (T)</td>
</tr>
<tr>
<td>$q_1$</td>
<td>Lock (T)</td>
</tr>
<tr>
<td>$q_2$</td>
<td>Request (D)</td>
</tr>
<tr>
<td>$q_3$</td>
<td>Lock (D)</td>
</tr>
<tr>
<td>$q_4$</td>
<td>Perform function</td>
</tr>
<tr>
<td>$q_5$</td>
<td>Unlock (T)</td>
</tr>
<tr>
<td>$q_6$</td>
<td>Unlock (D)</td>
</tr>
</tbody>
</table>

Figure 6.4 Example of Two Processes Competing for Reusable Resources
Reusable Resources

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

  P1
  ...
  Request 80 Kbytes;
  ...
  Request 60 Kbytes;

  P2
  ...
  Request 70 Kbytes;
  ...
  Request 80 Kbytes;

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

- Created (produced) and destroyed (consumed)
- 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
Conditions for Deadlock

• Mutual exclusion
  – Only one process may use a resource at a time

• Hold-and-wait
  – A process may hold allocated resources while awaiting assignment of others
Conditions for Deadlock

• No preemption
  – No resource can be forcibly removed from a process holding it

• 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
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

(c) Circular wait
(d) No deadlock
Resource Allocation Graphs

Figure 6.6 Resource Allocation Graph for Figure 6.1b
Deadlock Prevention

• Mutual Exclusion
  – Spooling

• Hold and Wait
  – Require that a process request all of its required resources at one time
  – Requests would be granted/denied simultaneously
Deadlock Prevention (cont.)

• No Preemption
  – Process must release resource and request again
  – OS may preempt a process and require it to release its resources

• Circular Wait
  – Define a linear ordering of resources
  – Require that processes request resources according to the ordering
Deadlock Avoidance

• A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock

• Requires knowledge of future process requests
Two Approaches to 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
Resource Allocation Denial

• Referred to as the *Banker’s Algorithm*

• State of the system is the current allocation of resources to process

• *Safe state* is where there is at least one sequence of execution of processes that does not result in deadlock

• *Unsafe state* is a state that is not safe
Determination of a Safe State

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>6</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td><strong>Claim matrix C</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td><strong>Allocation matrix A</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td><strong>C - A</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td><strong>Resource vector R</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td><strong>Available vector V</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) Initial state
## Determination of a Safe State

### Claim matrix C

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

### Allocation matrix A

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</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>6</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

### C - A

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(b) P2 runs to completion
Determination of an Unsafe State

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>6</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Claim matrix C

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Allocation matrix A

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

C - A

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

Resource vector R

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

Available vector V

(a) Initial state

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

(a) global data structures

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

(b) resource alloc algorithm

```c
if (alloc[i,*] + request[*] > claim[i,*])
    < error >; /* total request > claim*/
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 >;
}
```
boolean safe (state S) {
    int currentavail[m];
    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)

Figure 6.9 Deadlock Avoidance Logic
Deadlock Avoidance

• Maximum resource requirement must be stated in advance
• Processes under consideration must be independent; no synchronization (order of execution) requirements
• No process may exit/block while holding resources
Deadlock Detection

**Request matrix Q**

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Allocation matrix A**

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Resource vector**

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**Allocation vector**

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

*Figure 6.10 Example for Deadlock Detection*
Strategies Once Deadlock Detected

• Abort all deadlocked processes
• Back up each deadlocked process to some previously defined checkpoint, and restart all process - original deadlock may re-occur
• Successively abort deadlocked processes until deadlock no longer exists
• Successively preempt resources until deadlock no longer exists
Dining Philosophers Problem

Figure 6.11  Dining Arrangement for Philosophers
Dining Philosophers Problem

/* 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));
}
Dining Philosophers Problem with Semaphores

/* 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));
}
Dining Philosophers Problem with Monitor

```c
void philosopher[k=0 to 4] /* the five philosopher clients */
{
    while (true) {
        <think>;
        get forks(k); /* client requests two forks via monitor */
        <eat spaghetti>;
        release forks(k); /* client releases forks via the monitor */
    }
}
```

Figure 6.14 A Solution to the Dining Philosophers Problem Using a Monitor
Dining Philosophers Problem with Monitor

```java
monitor dining_controller;
cond ForkReady[5]; /* condition variable for synchronization */
boolean fork[5] = {true}; /* availability status of each fork */

void get_forks(int pid) /* pid is the philosopher id number */
{
    int left = pid;
    int right = (++pid) % 5;
    /* grant the left fork */
    if (!fork(left)
        cwait(ForkReady[left]); /* queue on condition variable */
    fork(left) = false;
    /* grant the right fork */
    if (!fork(right)
        cwait(ForkReady[right]); /* queue on condition variable */
    fork(right) = false;
}

void release_forks(int pid)
{
    int left = pid;
    int right = (++pid) % 5;
    /* release the left fork */
    if (empty(ForkReady[left]) /* no one is waiting for this fork */
        fork(left) = true;
    else /* awaken a process waiting on this fork */
        csignal(ForkReady[left]);
    /* release the right fork */
    if (empty(ForkReady[right]) /* no one is waiting for this fork */
        fork(right) = true;
    else /* awaken a process waiting on this fork */
        csignal(ForkReady[right]);
}
```
UNIX Concurrency Mechanisms

- Pipes: circular buffer for two processes like in producer-consumer
- Messages: blocking receive
- Shared memory: shared pages – mutual exclusion is not guaranteed
- Semaphores
- Signals
Linux Kernel Concurrency Mechanism

• Includes all the mechanisms found in UNIX

• Atomic operations execute without interruption and without interference (by blocking the memory bus)
Linux Atomic Operations

- arithmetic operation plus setting condition code
- spinlocks for mutual exclusion (loop until lock acquired)
- traditional and readers-writer semaphores
- memory barrier operations: limit compiler or CPU in re-ordering instructions