Previous Class...

• What is Deadlock?
• How to detect it?
• Dealing with deadlock:
  – Prevention
  – Avoidance
  – Detection
Outline

• Barrier Problem
• Readers-Writers Problem
• Review of Synchronization
• Issues of lock-based programming
Deadlock

- A *set* of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set.

Some Slides Courtesy of Dr. William Stallings
Potential Deadlock

I need quad C and D

I need quad B and C

I need quad D and A

I need quad A and B
Actual Deadlock

HALT until D is free

HALT until C is free

HALT until A is free

HALT until B is free
Resource Categories

Reusable
- can be safely used by only one process at a time and is not depleted by that use
- processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores

Consumable
- one that can be created (produced) and destroyed (consumed)
- interrupts, signals, messages, and information in I/O buffers
Example of Deadlock: Memory Request

• 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
Example of Deadlock: waiting for messages

• Consider a pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process:

S1 = 1; s2 = 1;

P1:  
P(s1)  
P(s2)  

P2:  
P(s2)  
P(s1)
There is a circle in the graph, which indicates deadlock.
Resource Allocation Graph describing the traffic jam

Resource Allocation Graph

P1 → Ra
P2 → Rb
P3 → Rc
P4 → Rd
## Conditions for Deadlock

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mutual Exclusion</strong></td>
<td>A process cannot access a resource that has been allocated to another process</td>
</tr>
<tr>
<td><strong>Hold-and-Wait</strong></td>
<td>A process may hold allocated resources while awaiting assignment of others</td>
</tr>
<tr>
<td><strong>No Pre-emption</strong></td>
<td>No resource can be forcibly removed from a process holding it</td>
</tr>
<tr>
<td><strong>Circular Wait</strong></td>
<td>A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain</td>
</tr>
</tbody>
</table>
Dealing with Deadlock

• Three general approaches exist for dealing with deadlock:

  Prevent Deadlock
  • adopt a policy that eliminates one of the conditions

  Avoid Deadlock
  • make the appropriate dynamic choices based on the current state of resource allocation

  Detect Deadlock
  • attempt to detect the presence of deadlock and take action to recover
Deadlock Condition Prevention

Mutual Exclusion

Avoiding mutual exclusion is not realistic

Hold and Wait

Countermeasure:
require that a process request all required resources at once; blocking the process until all requests can be granted simultaneously
Deadlock Condition Prevention

• No Preemption
  – *Countermeasure*: if a process holding certain resources is denied a further request, that process must release its original resources and request them again

• Circular Wait
  – *Countermeasure*: define a linear ordering of resource numbers; if a process has been allocated a resource of number $R$, then it may subsequently request only those resources of numbers following $R$ in the ordering.
  – *Why does this work?*
    • *Think about the Resource Allocation Graph*
Deadlock Avoidance

- **Deadlock prevention** breaks one of the deadlock conditions through rules, which are defined before execution, while **deadlock avoidance** is enforced during execution.
- A decision is made dynamically whether the current resource allocation request will lead to an unsafe state.
- Requires knowledge of future process requests.
- We will examine some examples.
Example

State of a system consisting of 4 processes and 3 resources

- Allocations have been made as follows

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

Resource vector R

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3</td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>P4</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Available vector V

(a) Initial state
Determination of a Safe State

- P2 requests one of R1 and one unit of R3
- Should this request be granted?
- **Banker’s algorithm**: assume this request is granted, then check whether the resulted state is safe
- A state is **safe** if there is at least one sequence of resource allocations that satisfies all the processes’ needs

<table>
<thead>
<tr>
<th>P1</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td></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>P1</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td></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>
</tbody>
</table>

Allocation matrix A

<table>
<thead>
<tr>
<th>P1</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td></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>
</tbody>
</table>

C – A

<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

Is this a safe state?
P2 Runs to Completion

Old Available vector (0, 1, 1) + Resources released by P2 (6, 1, 2) =
Updated available vector(6, 2, 3)
P1 Runs to Completion

Old Available vector (6, 2, 3) + Resources Released by P1 (1, 0, 0) = Updated available vector (7, 2, 3)
P3 Runs to Completion

Thus, the state defined originally is safe
Determination of an Unsafe State

P1 requests for one more R1 and one more R3

The request should not be granted, because it leads to an unsafe state
Deadlock detection
Recovery strategies

- Kill one deadlocked process at a time and release its resources
- Kill all deadlocked processes
- Steal one resource at a time
- Roll back all or one of the processes to a checkpoint that occurred before they requested any resources, then continue
  - Difficult to prevent indefinite postponement
Recovery by killing processes

Figure 6.6 Resource Allocation Graph for Figure 6.1b
<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 | • Works well for processes that perform a single burst of activity  
• No preemption necessary | • Inefficient  
• Delays process initiation  
• Future resource requirements must be known by processes |
|           |                             | Preemption         | • Convenient when applied to resources whose state can be saved and restored easily | • Preempts more often than necessary                     |
|           |                             | Resource ordering  | • Feasible to enforce via compile-time checks  
• Needs no run-time computation since problem is solved in system design | • Disallows incremental resource requests                |
| Avoidance | Midway between that of detection and prevention | Manipulate to find at least one safe path | • No preemption necessary                                                      | • Future resource requirements must be known by OS  
• Processes can be blocked for long periods |
| Detection | Very liberal; requested resources are granted where possible | Invoke periodically to test for deadlock | • Never delays process initiation  
• Facilitates online rescheduling | • Inherent preemption losses |
Dining Philosophers: failed solution with deadlock

```c
#define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1)%N);
        eat(); /* yummy */
        put_fork(i);
        put_fork((i+1)%N);
    }
}
```
Dining Philosophers: failed solution with deadlock

# define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1)%N);
        eat(); /* yummy */
        put_fork(i);
        put_fork((i+1)%N);
    }
}
Dining Philosophers: failed solution with deadlock

```c
#define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1)%N);
        eat(); /* yummy */
        put_fork(i);
        put_fork((i+1)%N);
    }
}
```
Dining Philosophers solution with numbered resources

Instead, number resources

First request lower numbered fork

```c
#define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(LOWER(i));
        take_fork(HIGHER(i));
        eat(); /* yummy */
        put_fork(LOWER(i));
        put_fork(HIGHER(i));
    }
}
```
Dining Philosophers solution with numbered resources

Instead, number resources...

Then request higher numbered fork

```c
#include <stdio.h>

#define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(LOWER(i));
        take_fork(HIGHER(i));
        eat(); /* yummy */
        put_fork(LOWER(i));
        put_fork(HIGHER(i));
    }
}
```
Dining Philosophers solution with numbered resources

Instead, number resources...

Then request higher numbered fork

```c
#define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(LOWER(i));
        take_fork(HIGHER(i));
        eat(); /* yummy */
        put_fork(LOWER(i));
        put_fork(HIGHER(i));
    }
}
```
Dining Philosophers solution with numbered resources

Instead, number resources...

One philosopher can eat!

```c
#define N 5

void philosopher (int i) {
    while (TRUE) {
        think();
        take_fork(LOWER(i));
        take_fork(HIGHER(i));
        eat(); /* yummy */
        put_fork(LOWER(i));
        put_fork(HIGHER(i));
    }
}
```
Summary

• What is Deadlock?
• How to detect it?
• Dealing with deadlock:
  – Prevention
  – Avoidance
  – Detection