OPERATING SYSTEMS

Week 6

Much of the material on these slides comes from the recommended textbook by William Stallings

Announcement

Weekly program

Midterm

- ❑ Week   8 : 17/09/2019
- ❑ Syllabus: Up to Real-time Scheduling (Lecture and Workshop)
- ❑ Venue: LSTH100
- ❑ Time: 12:00-13:00
- ❑ We will have lecture from 11:00 to 12:00 after that we will have the midterm.

Detailed content

Weekly program

- ✓ Week 1 – Operating System Overview
- ✓ Week 2 – Processes and Threads
- ✓ Week 3 – Scheduling
- ✓ Week 4 – Real-time System Scheduling and Multiprocessor Scheduling
- ✓ Week 5 – Concurrency: Mutual Exclusion and Synchronization

➡ ❑ **Week 6 – Concurrency: Deadlock and Starvation**

- ❑ Week 7 – Memory Management I
- ❑ Week 8 – Memory Management II
- ❑ Week 9 – Disk and I/O Scheduling
- ❑ Week 10 – File Management
- ❑ Week 11 – Security and Protection
- ❑ Week 12 – Revision of the course
- ❑ Week 13 – Extra revision (if needed)

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Key Concepts From Last Lecture

- Semaphore is a special purpose signaling variable that can be operated using two primitives: `semSignal` and `semWait`

- Two types: general semaphore and binary semaphore – essentially have the same expressive power

- Semaphores can be implemented in software or with support from hardware

- Semaphores are powerful and felxible tools for enforcing mutual exclusion but difficult to manage and handle
  - Programmers are responsible to ensure mutual exclusion and synchronization

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Key Concepts From Last Lecture

- Monitors provide equivalent functionality of semaphores but easier to use
    - Monitor itself ensures mutual exclusion and programmers are responsible to ensure synchronization
- Solution to two classic problems
    - Producer/Consumer problem
    - Readers/Writers problem

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Week 06 Lecture Outline

Concurrency: Mutual Exclusion, Deadlock and Starvation

- ❑ Software Approaches to mutual exclusion
- ❑ Dekker's Algorithm
- ❑ Peterson's Algorithm
- ❑ Deadlock
- ❑ Principles of Deadlock
- ❑ Consumable and Reusable Resources
- ❑ Conditions for Deadlock
- ❑ Deadlock Prevention
- ❑ Deadlock Avoidance
- ❑ Deadlock Detection
- ❑ Dining Philosopher Problem

 Videos to watch before lecture

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Mutual Exclusion: Software Approach

- Software approaches can be implemented for concurrent processes
    - Executing on single processor or multiple processor
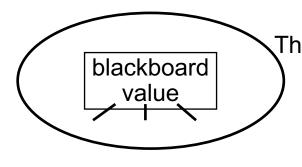   - Assumes elementary mutual exclusion at memory access level
    - No other support in hardware, OS or programming language is assumed
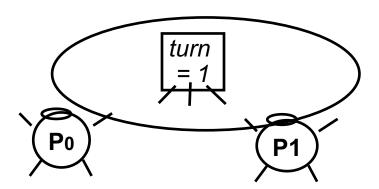
Mutual Exclusion: Software Approach

- In any case, we can assume that the hardware provides **elementary mutual exclusion** for memory accesses.
  - Simultaneous access to the same location in main memory is prevented by the hardware.
  - Requests are serialised and granted in an unspecified order.

blackboard value

The elementary hardware protection is modelled by the **protocol igloo**: the entrance and interior of the igloo are small enough that at most **one person can enter** or **be in** the igloo at a time.

The igloo contains a **blackboard** for writing a message (usually a single value).

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Software approach: Attempt 1

- A process Pi that wants to execute its critical section first enters the igloo and checks the blackboard.

- If process Pi finds the value i, then it leaves the igloo and goes critical.

- If not, then Pi repeats the entry and check until the blackboard shows i.

- When it finishes the critical section, it returns to the igloo and writes the number of the other process.

- **Mutual exclusion** is guaranteed.

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

```
global var turn:0..1;

process  Pi
begin
 while turn !=i do  <nothing>;
 <critical section>
 turn := 1-i
end.
```

Turn = 0

```
process  P0
begin
 while turn !=0
       do  <nothing>;
 <critical section>
turn := 1-0
end.
```

```
process  P1
begin
 while turn !=1
       do  <nothing>;
  <critical section>
 turn := 1-1
end.
```

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

```
global var turn:0..1;

process  Pi
begin
 while turn !=i do  <nothing>;
 <critical section>
 turn := 1-i
end.
```

Turn = 0

```
process  P0
begin
  while turn !=0
        do  <nothing>;
 <critical section>
turn := 1-0
end.
```

```
process  P1
begin
  while turn !=1
        do  <nothing>;
 <critical section>
 turn := 1-1
end.
```

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

global var turn:0..1;

process P*i*
begin
  while turn !=i *do* <nothing>;
  <critical section>
  turn := 1-i
end.

Turn = 0

process P*0*
begin
  while turn !=0
        do <nothing>;
  <critical section>
turn := 1-0
end.

process P*1*
begin
  while turn !=1
        do <nothing>;
  <critical section>
  turn := 1-1
end.

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

global var turn:0..1;

process **P***i*
begin
  while turn !=i **do** <nothing>;
  <critical section>
  turn := 1-i
end.

Turn = 1

process **P***0*
begin
  while turn !=0
       do <nothing>;
  <critical section>
 turn := 1-0
end.

process **P***1*
begin
  while turn !=1
       do <nothing>;
 <critical section>
 turn := 1-1
end.

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

global var *turn:0..1;*

process **P**_*i*_
begin
 while *turn !=i* **do** <nothing>;
 <critical section>
 turn := 1-i
end.

Turn = 1

process **P**_*1*_
begin
 while *turn !=1*
  do <nothing>;
 <critical section>
 turn := 1-1
end.

process **P**_*0*_
begin
 while *turn !=0*
  do <nothing>;
 <critical section>
turn := 1-0
end.

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

```
global var turn:0..1;

process Pi
begin
 while turn !=i do  <nothing>;
 <critical section>
 turn := 1-i
end.
```

Turn = 1

```
process P0
begin
 while turn !=0
      do  <nothing>;
 <critical section>
turn := 1-0
end.
```

```
process P1
begin
 while turn !=1
      do  <nothing>;
<critical section>
 turn := 1-1
end.
```

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

```
global var turn:0..1;

process Pi
begin
  while turn !=i do <nothing>;
  <critical section>
  turn := 1-i
end.
```

Turn = 0

```
process P1
begin
  while turn !=1
       do <nothing>;
  <critical section>
  turn := 1-1
end.
```

```
process P0
begin
  while turn !=0
       do <nothing>;
  <critical section>
turn := 1-0
end.
```

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

```
global var turn:0..1;

process  Pi
begin
 while turn !=i do  <nothing>;
 <critical section>
 turn := 1-i
end.
```

Turn = 0

```
process  P0
begin
 while turn !=0
       do  <nothing>;
 <critical section>
turn := 1-0
end.
```

```
process  P1
begin
 while turn !=1
       do  <nothing>;
 <critical section>
 turn := 1-1
end.
```

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

```
global var turn:0..1;

process  Pi
begin
 while turn !=i do  <nothing>;
 <critical section>
 turn := 1-i
end.
```

Turn = 1

```
process  P0
begin
 while turn !=0
      do  <nothing>;
 <critical section>
turn := 1-0
end.
```

```
process  P1
begin
 while turn !=1
      do  <nothing>;
 <critical section>
 turn := 1-1
end.
```

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

```
global var turn:0..1;

process  Pi
begin
  while turn !=i do  <nothing>;
  <critical section>
  turn := 1-i
end.
```

Turn = 1

```
process  P0
begin
  while turn !=0
        do  <nothing>;
  <critical section>
turn := 1-0
end.
```
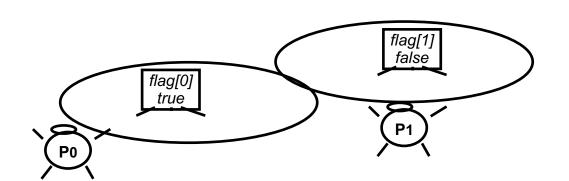
```
process  P1
begin
  while turn !=1
        do  <nothing>;
  <critical section>
  turn := 1-1
end.
```

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

global var *turn:0..1;*

process **P***i*
begin
 while *turn !=i* *do* <nothing>;
 <critical section>
 turn := 1-i
end.

Turn = 1

process **P**0
begin
 while *turn !=0*
      do <nothing>;
 <critical section>
turn := 1-0
end.

process **P**1
begin
 while *turn !=1*
      do <nothing>;
<critical section>
 turn := 1-1
end.

03/09/2019
COMP2240 - Semester 2 - 2019 | www.newcastle.edu.au

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

global var *turn:0..1;*

process **P***i*
begin
 while *turn !=i* ***do*** <nothing>;
 <critical section>
 turn := 1-i
end.

Turn = 1

process **P***1*
begin
 while *turn !=1*
      do <nothing>;
 <critical section>
 turn := 1-1
end.

process **P***0*
begin
 while *turn !=0*
      do <nothing>;
 <critical section>
turn := 1-0
end.

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Software approach: Attempt 1

Problems:

- The procedure is "**busy-waiting**", that is, the thwarted process can do nothing productive while it waits, yet it **keeps the CPU active** by continually checking.

- Pace of execution is **limited to the slower process**, because the executions must strictly alternate.

- **It does NOT prevent starvation**, because one process may fail, permanently blocking the other.

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Software approach: Attempt 2

```
var flag  : array[0..1] of Boolean;

process  Pi
  while flag[1-i] do  <nothing>;
  flag[i]:=true;
  <critical section>
  flag[i]:=false;
```

- Each process should have its own key to its critical section, rather than rely on others as in last attempt.

- Process Pi is in its critical section if and only if we have flag[i] = true.

- Each process can **check the other** blackboard, but it can **update only its own** blackboard.

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

```
var flag  : array[0..1] of Boolean;

process  Pi
  while flag[1-i] do  <nothing>;
 flag[i]:=true;
  <critical section>
  <critical section>
 flag[i]:=false;
```

```
process  P0
  while flag[1] do
         <nothing>;
 flag[0]:=true;
  <critical section>
  <critical section>
flag[0]:=false;
```

```
process  P1
  while flag[0] do
         <nothing>;
 flag[1]:=true;
  <critical section>
  <critical section>
flag[1]:=false;
```

flag[0] = false   *flag[1] = false*

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Software approach: Attempt 2

Problem:

1. This method **does not guarantee mutual exclusion**. It is not independent of relative speeds of process execution.
    1. P0 executes **while** and finds flag[1]=false.
    2. P1 executes **while** and finds flag[0]=false.
    3. P0 sets flag[0]:=true and goes critical.
    4. P1 sets flag[1]:=true and goes critical.

2. If one process fails outside the critical section then the other process is not blocked but if a process fails inside its critical section then the other process is permanently blocked.

Software approach: Attempt 3

- Attempt 2 failed because one process could change its state (flag)
  - after the other process has checked it,
  - before it enters its critical section.
- We can fix this by changing state **before** checking the other's flag.

```
var flag  : array[0..1] of Boolean;

process  Pi
  flag[i]:=true;
  while flag[1-i] do  <nothing>
  <critical section>
  flag[i]:=false;
```

- This enforces mutual exclusion.

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

```
var flag  : array[0..1] of Boolean;

process  Pi
  flag[i]:=true;
  while flag[1-i] do  <nothing>
  <critical section>
  <critical section>
  flag[i]:=false;
```

process **P0**
  flag[0]:=true;
  while *flag[1]* ***do***
          <nothing>
  <critical section>
  <critical section>
  flag[0]:=false;

process **P1**
  flag[1]:=true;
  while *flag[0]* ***do***
          <nothing>
  <critical section>
  <critical section>
  flag[1]:=false;

flag[0] = true   *flag[1] = true*

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Software approach: Attempt 3

Problem

- It does **not prevent deadlock**: both processes may set their flags to *true* and be caught on the *while* statement before either enters the critical section. Both processes would then think that the other was in the critical region, when neither are!

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Software approach: Attempt 4

- Attempt 3 fails because **a process sets its state without knowing the state of the other process**.

- Thus we change the procedure so that **each process gives the other some time to jump ahead of it**, i.e. it "backs off", or defers to the other process.

```
flag[i]:=true
while flag[1-i] do
  begin
    flag[i]:=false;
    <delay for a short time>
    flag[i]:=true
  end
<critical section>
flag[i]:=false
```

- **Mutual exclusion** is enforced.

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Software approach: Attempt 4

```
flag[i]:=true
while flag[1-i] do
  begin
    flag[i]:=false;
    <delay for a short time>
    flag[i]:=true
  end
<critical section>
flag[i]:=false
```

- **Problem**
  – P0 sets flag[0]:=true
  – P1 sets flag[1]:=true
  – P0 checks flag[1]
  – P1 checks flag[0]
  – P0 sets flag[0]:=false
  – P1 sets flag[1]:=false
  – P0 sets flag[0]:=true
  – P1 sets flag[1]:=true

  – . . . .

This is **livelock**.
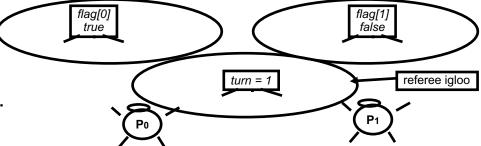
Although it seems **unlikely**, it is a **possible** scenario, and can easily happen in practice.

Alteration in relative speed of the 2 processes will break the cycle.

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Correct S/W approach: Dekker's Algorithm

We to give the processes **the right to insist** that they can enter their critical sections. The processes can take **turns** to insist.

flag[0]
true

flag[1]
false

turn = 1

referee igloo

P0

P1

- Suppose that P0 wants to go critical.
    - it sets its own flag to true.
    - it checks the flag of P1:
        - if P1's flag is false, then P0 goes critical.
        - if P1's flag is true, then it checks the referee
            - if turn = 0, then P0 has the right to insist on taking a turn, and it checks P1's flag again.
            - if turn = 1, then P0 sets its flag to false, waits for its turn, resets its flag and goes critical.
    - after finishing its critical section, P0 sets turn = 1 and its flag to false transferring the right to insist to P1.

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Dekker's Algorithm

var flag: *array*[0..1] of Boolean;
    turn:0..1;

```
main program
begin
  flag[0]:=false;
  flag[1]:=false;
  turn:=1;
  parbegin
    P0;
    P1
  parend
end
```

```
process  P0
begin
 repeat
   flag[0]:=true;
   while flag[1] do
           if turn=1 then
             begin
             flag[0]:=false;
             while turn=1 do <nothing>
             flag[0]:=true
             end;
   <critical section>
   turn:=1;
   flag[0]:=false;
   <remainder of the process>
 forever
end;
```

```
process  P1
begin
 repeat
   flag[1]:=true;
   while flag[0] do
           if turn=0 then
             begin
             flag[1]:=false;
             while turn=0 do <nothing>
             flag[1]:=true
             end;
   <critical section>
   turn:=0;
   flag[1]:=false;
   <remainder of the process>
 forever
end;
```

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Dekker's Algorithm

- Dekker's Algorithm solves the mutual exclusion but with a rather complicated program that is difficult to follow and whose correctness is difficult to prove

- Peterson (1981) provided a simple elegant solution.

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Peterson's Algorithm

- **Mutual exclusion is enforced**:
- **Deadlock is prevented**:

```
var flag: array[0..1] of Boolean;
     turn:0..1;
```

```
procedure  P0;
begin
  repeat
    flag[0]:=true;
    turn:=1;
    while flag[1] and turn=1 do <nothing>
    <critical section>
    flag[0]:=false;
    <remainder>
  forever
end
```

```
procedure  P1;
begin
  repeat
    flag[1]:=true;
    turn:=0;
    while flag[0] and turn=0 do  <nothing>
    <critical section>
    flag[1]:=false;
    <remainder>
  forever
end
```

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Peterson's Algorithm

- **Mutual exclusion is enforced**: Suppose that P0 wants to enter its critical section. First, it sets flag[0]   to true; then it is impossible for P1 to enter its critical section. If P1 is already in its critical section, then flag[1] must be true, and so P0 cannot get to its critical section.

- **Deadlock is prevented**: Suppose that both P0 and P1 are blocked; this can only occur indefinitely if both are in the **while** loops. But the conditions on these **while** loops are inconsistent: it is not possible to have turn=1 and turn=0 at the same time.

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Summary

- Software approaches to mutual exclusion does not assume any support from hardware, OS or programming language
    - Except that no simultaneous access to the same location in main memory is possible.
- Dekker's algorithm/Peterson algorithms
    - Can ensure mutual exclusion and freedom from deadlock and freedom from starvation
    - Employs busy waiting

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Deadlock
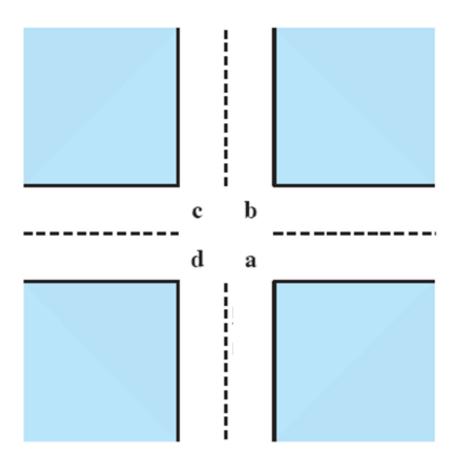
- The permanent blocking of a set of processes that either compete for system resources or communicate with each other

- 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

- Permanent – none of the events is ever triggered

- No efficient solution
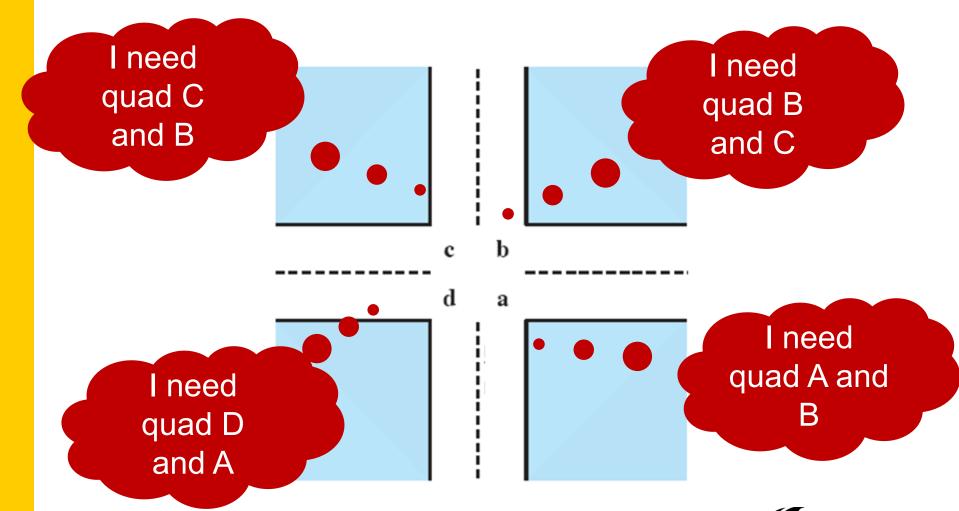
Potential Deadlock
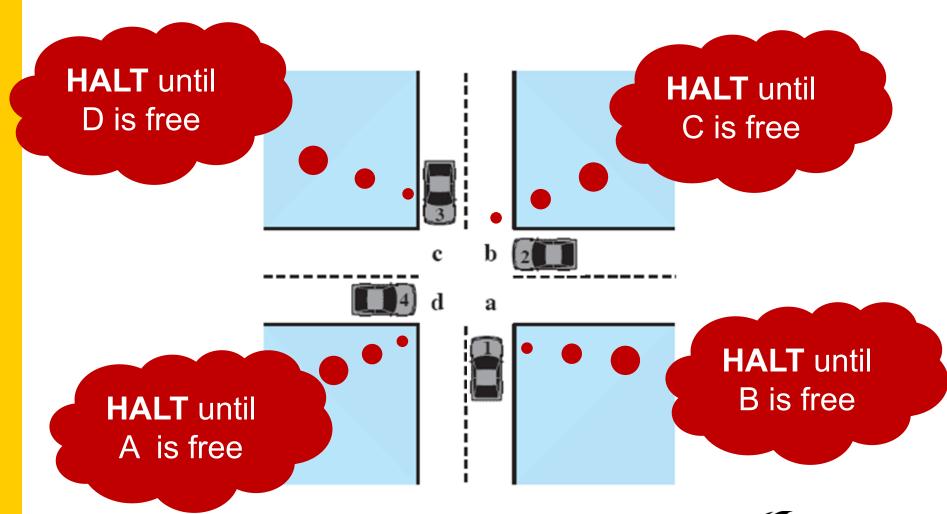
c   b

d   a

Potential Deadlock

Actual Deadlock

Process deadlock

- Consider two processes

Process P	Process Q
….	….
Get A	Get B
….	….
Get B	Get A
….	….
Release A	Release B
….	….
Release B	Release A
….	….
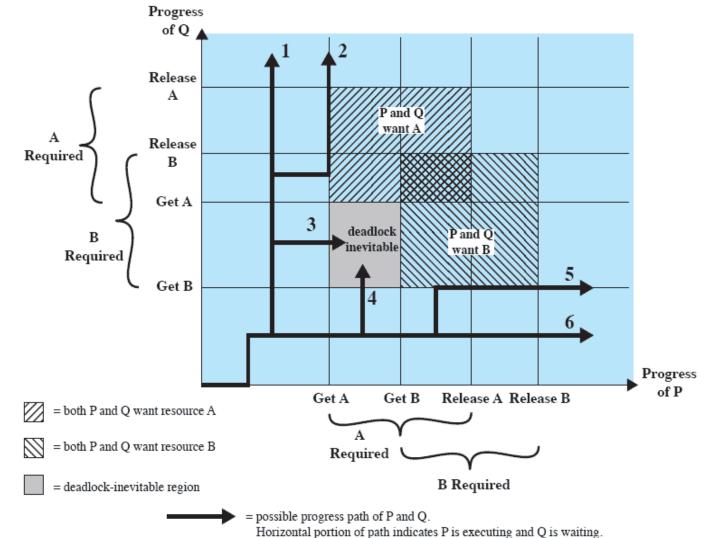
THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Joint Process Diagram

Figure 6.2 Example of Deadlock

Process deadlock

- How about?

Process P	Process Q
….	….
Get A	Get B
….	….
Release A	Get A
….	….
Get B	Release B
….	….
Release B	Release A
….	….

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

No Deadlock example

Figure 6.3 Example of No Deadlock [BACO03]
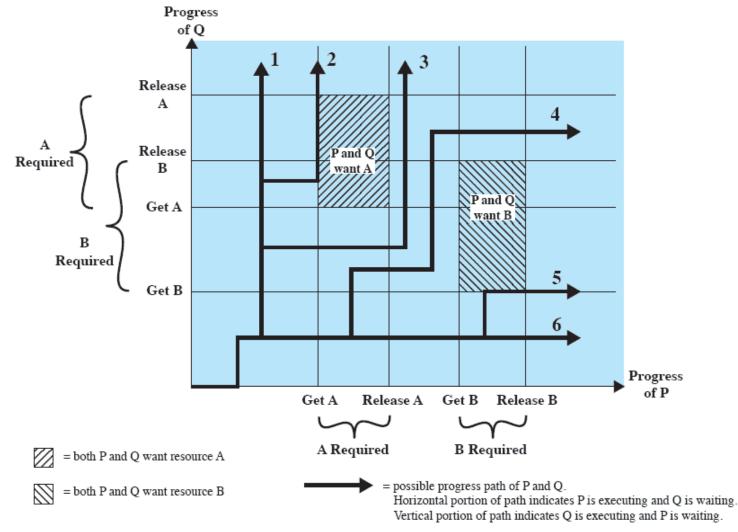
Resource Categories

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

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Reusable resource example 1

	Process P		Process Q
Step	**Action**	**Step**	**Action**
p_0	Request (D)	q_0	Request (T)
p_1	Lock (D)	q_1	Lock (T)
p_2	Request (T)	q_2	Request (D)
p_3	Lock (T)	q_3	Lock (D)
p_4	Perform function	q_4	Perform function
p_5	Unlock (D)	q_5	Unlock (T)
p_6	Unlock (T)	q_6	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

- Process P and Q competing for resource D and T
- Deadlock if execution order is : p0, p1, q0, q1, p2, q2

- Such deadlocks do occur, and the cause is often embedded in complex program logic, making detection difficult.

- One strategy is to impose system design constraints concerning the order in which resources can be requested.

Reusable example 2: Memory request

- Space is available for allocation of 200 Kbytes, 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

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Consumable resource deadlock

- 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:

P1	P2
…	…
Receive (P2);	Receive (P1);
…	…
Send (P2, M1);	Send (P1, M2);

- Deadlock occurs if the Receive is blocking

Resource Allocation Graphs

- **The resource allocation** graph is a directed graph that depicts a state of the system of resources and processes, with each process and each resource represented by a node.

- A graph edge directed from a process to a resource indicates a resource that has been requested by the process but not yet granted.
    - Within a resource node, a dot is shown for each instance of that resource.
    - Examples of resource types that may have multiple instances are I/O devices that are allocated by a resource management module in the OS.

- A graph edge directed from a reusable resource node dot to a process indicates a request that has been granted
    - The process has been assigned one unit of that resource. A graph edge directed from a consumable resource node dot to a process indicates that the process is the producer of that resource.

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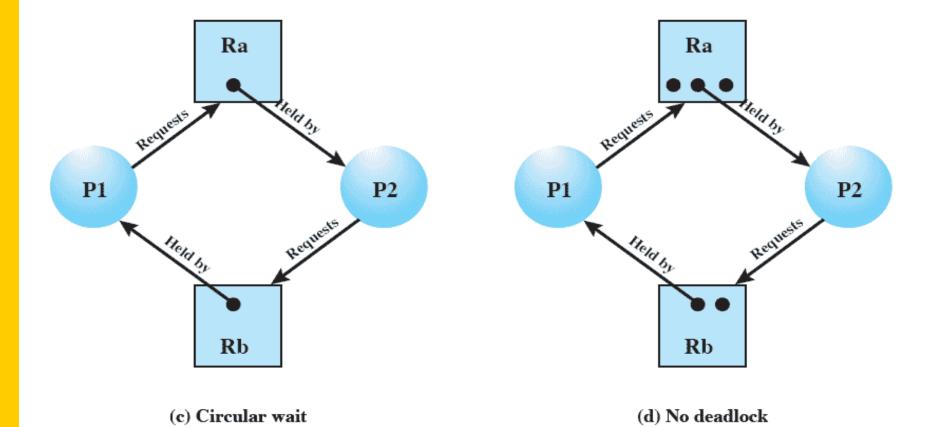
Resource Allocation Graphs

(c) Circular wait

(d) No deadlock

Circular chain deadlock

Figure 6.1   Illustration of Deadlock
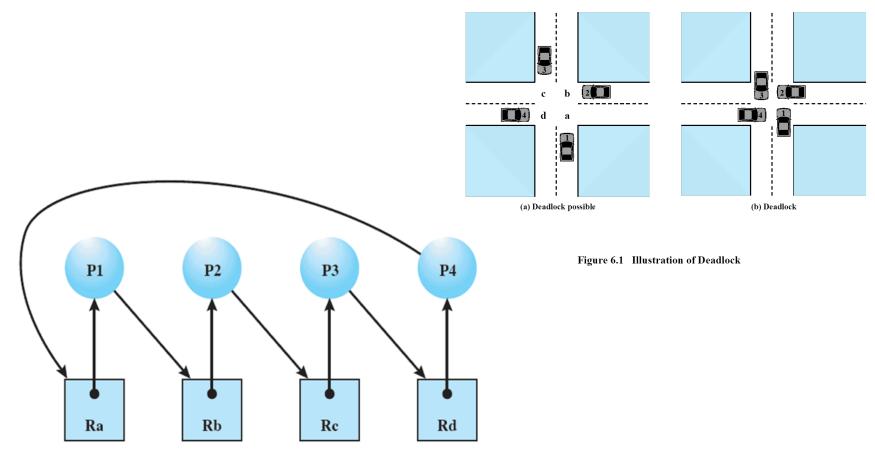
Figure 6.6   Resource Allocation Graph for Figure 6.1b

Conditions for Deadlock

Mutual Exclusion	Hold-and-Wait	No Pre-emption	Circular Wait
• only one process may use a resource at a time	• a process may hold allocated resources while awaiting assignment of others	• no resource can be forcibly removed from a process holding it	• a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

Conditions for Deadlock

Mutual Exclusion	Hold-and-Wait	No Pre-emption
• only one process may use a resource at a time	• a process may hold allocated resources while awaiting assignment of others	• no resource can be forcibly removed from a process holding it

- These conditions are quite desirable?
- The first three conditions are necessary but not sufficient for a deadlock to exist

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Conditions for Deadlock
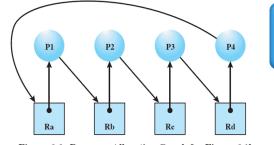
Figure 6.6   Resource Allocation Graph for Figure 6.1b

Mutual Exclusion	Hold-and-Wait	No Pre-emption	Circular Wait
• only one process may use a resource at a time	• a process may hold allocated resources while awaiting assignment of others	• no resource can be forcibly removed from a process holding it	• a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

- "Circular Wait" is actually a potential consequence of the first three conditions
- The unresolvable circular wait is in fact the definition of deadlock

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Dealing with deadlock

- **Prevent deadlock**
  - Adopt a policy that eliminates one of the conditions (conditions 1 through 4).

- **Avoid deadlock**
  - Make the appropriate dynamic choices based on the current state of resource allocation.

- **Detect** the presence of deadlock
  - When conditions 1 through 4 hold and take action to recover.

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Deadlock prevention strategy

- Design a system in such a way that the possibility of deadlock is excluded.

- Two main methods:
  - Indirect method
    - Prevent the occurrence of one of the three necessary conditions listed previously (items 1 through 3).

  - Direct method
    - Prevent the occurrence of a circular wait (item 4).

THE UNIVERSITY OF NEWCASTLE
AUSTRALIA

Deadlock condition prevention

- **Mutual Exclusion:**
  - In general this can not be disallowed
  - If access to a resource requires mutual exclusion then it must be supported by the OS

- **Hold and Wait:**
  - Require that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously
  - Inefficient
    - Process may be held up for a long time
    - Resources underutilized

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Deadlock condition prevention

- **No Preemption**
  - If a process holding certain resources is denied a further request, that process must release its original resources and request them again
  - OR, OS may preempt the second process and require it to release its resources
  - Practical when resources state can be easily saved and restored later

- **Circular Wait**
  - Define a linear ordering of resource types
  - If a process has been allocated resources of type R_i then it may subsequently request resource R_j if $i < j$.

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Deadlock avoidance

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

- Deadlock avoidance thus requires knowledge of future process resource requests.

Two approaches to deadlock avoidance

Deadlock Avoidance

Resource Allocation Denial

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

Process Initiation Denial

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

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Resource Allocation Denial

- Referred to as the **banker's algorithm**

- Consider a system with a fixed number of processes and a fixed number of resources. At any time a process may have zero or more resources allocated to it.

- **State** of the system reflects the current allocation of resources to processes
  - Two vectors, Resource and Available, and
  - Two matrices, Claim and Allocation

- **Safe state** is one in which there is at least one sequence of resource allocations to processes that does not result in a deadlock (i.e., all of the processes can be run to completion).

- **Unsafe state** is a state that is not safe.

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Example state 1

- State of a system consisting of four processes and three resources

- Allocations have been made to the four processes

- But is it safe?

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

(a) Initial state

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

P2 runs to completion

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

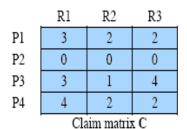| R1	R2	R3
6	2	3

Available vector V

(b) P2 runs to completion

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

P1 runs to completion

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
6	2	3

Available vector V

(b) P2 runs to completion

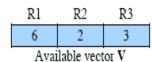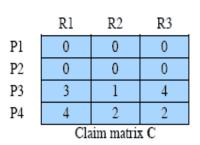|	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
7	2	3

Available vector V

(c) P1 runs to completion

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

P3 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix A
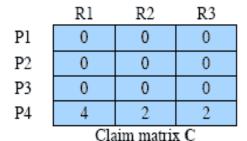
	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
9	3	4

Available vector V
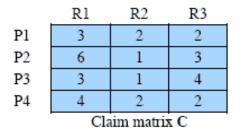
(d) P3 runs to completion

Thus, the state defined originally is a safe state

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Example state 2

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V
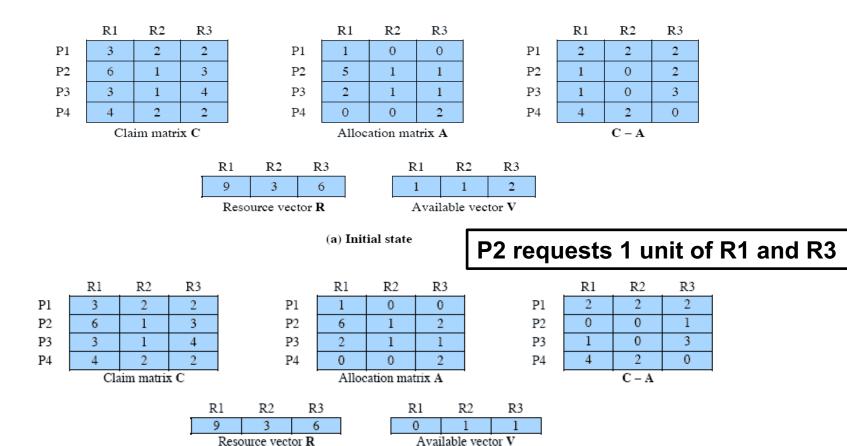
(a) Initial state

- Is currsent state safe? YES

- **P2 requests 1 R3. Is it safe to allocate the requested resources?**

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Example 3

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

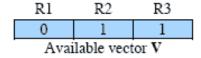| R1	R2	R3
1	1	2

Available vector V

(a) Initial state

P2 requests 1 unit of R1 and R3

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

| R1	R2	R3
0	1	1

Available vector V

(b) P2 requests one unit or R1 and R3

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Example 4

(a) Initial state

P1 requests 1 unit of R1 and R3

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

Deadlocked?

Deadlock avoidance logic[1]

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

(a) global data structures

```
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 >;
}
```

(b) resource alloc algorithm

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Deadlock avoidance logic₂

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

(c) test for safety algorithm (banker's algorithm)

Figure 6.9 Deadlock Avoidance Logic

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Deadlock avoidance

Advantages

- It is not necessary to preempt and rollback processes, as in deadlock detection

- Is less restrictive than deadlock prevention

Restrictions

- The maximum resource requirement for each process must be stated in advance

- The processes under consideration must be independent
  - the order in which they execute must be unconstrained by any synchronization requirements

- There must be a fixed number of resources to allocate.

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Deadlock Detection

- Deadlock prevention strategies are very conservative

  – Limit access to resources by imposing restrictions on processes

- Deadlock detection strategies do the opposite

  – Resource requests are granted whenever possible

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Deadline detection algorithms

- A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur.

- Checking at each resource request
  - Advantages:
    1. It leads to early detection
    2. The algorithm is relatively simple because it is based on incremental changes to the state of the system.
  - Disadvantage
    - Frequent checks consume considerable processor time.

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Deadline detection algorithm

- Use an Allocation matrix (**A**) and Available vector (**V**)
- Also have a Request matrix (**Q**)

1. Mark any process with all 0s in Allocation matrix **A**
2. Make temporary vector **W** = Availability vector **V**
3. Find an index **i** *where*
   - **P$_i$** is unmarked
   - *ith* row of **Q** (requests) <= *W* (available) [i.e. $Q_{ik} \leq W_k$ for $1 \leq k \leq m$]
   - If no such row is found then terminate
4. Mark **P$_i$**, add *A$_i$* to *W* [i.e. $W_k = W_k + A_{ik}$ for $1 \leq k \leq m$]
5. Goto Step 3

- On termination, if there are unmarked processes, then there is deadlock.
- Unmarked processes are involved in deadlock

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Deadline detection algorithm example

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available vector

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Deadline detection algorithm example

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available vector

1. Mark any process with all 0s in Allocation matrix **A**

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Deadline detection algorithm example

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

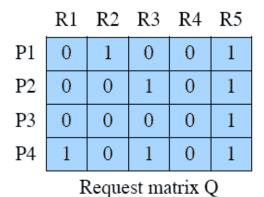| R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1
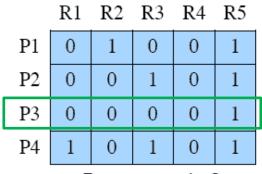
Available vector
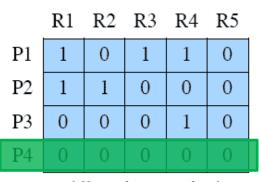
2. Make temporary vector **W** = Availability vector **V**

Initialise W = Available Vector
W = 0 0 0 0 1

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Deadline detection algorithm example

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1
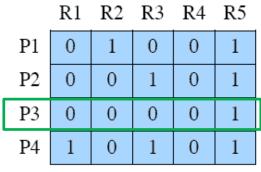
Available vector

W = 0 0 0 0 1

3. Find an index i where
  - P_i is unmarked
  - ith row of Q (requests) <= W (available) [i.e. $Q_{ik} \le W_k$ for $1 \le k \le m$]
  - If no such row is found then terminate

Deadline detection algorithm example

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available vector
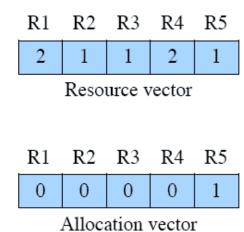
$$W = 0\ 0\ 0\ 1\ 1$$

4. Mark P_i, add A_i to W [i.e. $W_k = W_k + A_{ik}$ for $1 \leq k \leq m$]
5. Goto Step 3

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Deadline detection algorithm example

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

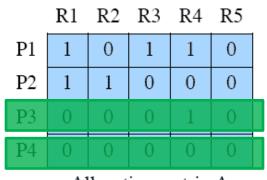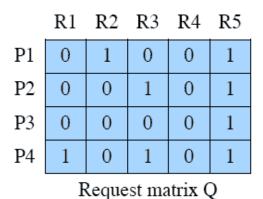| R1	R2	R3	R4	R5
0	0	0	0	1

Allocation vector

$$W = 0\ 0\ 0\ 1\ 1$$

3. Find an index i where
   - P_i is unmarked
   - ith row of Q (requests) <= W (available) [i.e. $Q_{ik} \leq W_k$ for $1 \leq k \leq m$]
   - If no such row is found then terminate

UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Deadline detection algorithm example

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

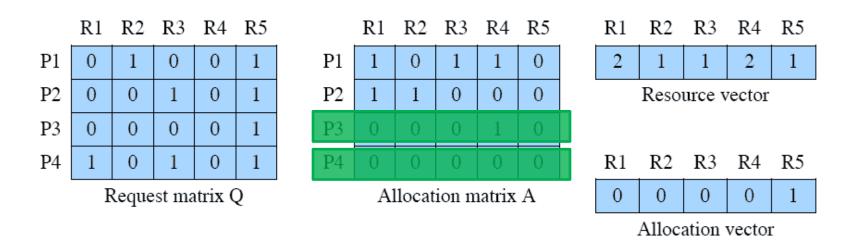| R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Allocation vector

Deadlock: Involved process are P1 and P2

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Deadlock recovery

- **Abort all deadlocked processes.**
  - One of the most common, if not the most common, solution adopted in operating systems.

- **Back up each deadlocked process to some previously defined checkpoint, and restart all processes**.
  - This requires that rollback and restart mechanisms be built in to the system.
  - The risk in this approach is that the original deadlock may recur but the nondeterminancy of concurrent processing may ensure that this does not happen.

- **Successively abort deadlocked processes until deadlock no longer exists.**
  - The order in which processes are selected for abortion should be on the basis of some <u>criterion of minimum cost</u>. After each abortion, the detection algorithm must be reinvoked to see whether deadlock still exists.

- **Successively preempt resources until deadlock no longer exists.**
  - As (3) above, a cost-based selection should be used, and reinvocation of the detection algorithm is required after each preemption.
  - A process that has a resource preempted from it must be rolled back to a point prior to its acquisition of that resource.

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Deadlock recovery

- **Possible Selection Criteria for aborting processes for deadlock recovery**
  - Choose the process
    - Least amount of time consumed so far
    - Lease amount of output produced so far
    - Most estimated time remaining
    - Least total resources allocated so far
    - Lowest priority

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

In integrated deadlock strategy

- There is no single effective strategy that can deal with all types of deadlock

- Use of different strategies in different situations

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
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 handling	•Inherent preemption losses

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Dining Philosophers Problem

Think

…
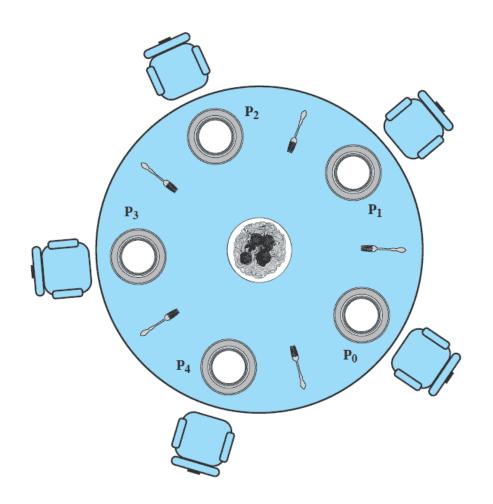
Eat

…

Think

…

Eat

…

(and spaghetti is the best)

Figure 6.11   Dining Arrangement for Philosophers

Dining Philosophers Problem

- No two philosophers can use the same fork at the same time (mutual exclusion)

- No philosopher must starve to death (avoid deadlock and starvation)
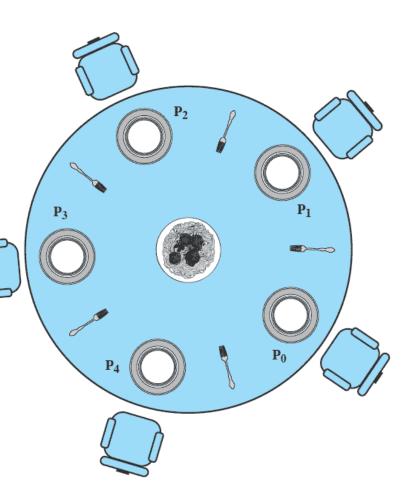
Figure 6.11   Dining Arrangement for Philosophers

Using semaphores (1)

```
/* 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));
    }
```

Figure 6.12   A First Solution to the Dining Philosophers Problem

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Using semaphores (1)

```
/* 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));
    }
```

Figure 6.12   A First Solution to the Dining Philosophers Problem

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Using semaphores (2)

```
/* 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));
}
```

Figure 6.13 A Second Solution to the Dining Philosophers Problem

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Using semaphores (2)

```
/* 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));
}
```

Figure 6.13 | A Second Solution to the Dining Philosophers Problem

UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Using a monitor

```
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]);
}

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

Using a monitor

```
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]);
}

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 */
   }
}
```

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

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

Summary

- Deadlock is blocking of a set of processes that either compete for resources or communicate with each other

- The blocking is permanent unless external intervention (e.g. OS action - such as killing of one or more processes or forcing process to release resource) is enforced

- Deadlock may involve reusable or consumable resources
  - Reusable – not depleted or destroyed by use
    - Memory, I/O channel
  - Consumable – destroyed when acquired by a process
    - Signal, message

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

Summary

- Three general approaches to deadlock: prevention, avoidance, detection
- Deadlock Prevention guarantees that deadlock will not occur
  - By assuring that one of the necessary conditions for deadlock will not occur
- Deadlock Avoidance involves analysis of each new resource request to determine if it could lead to deadlock and granting it only if deadlock is not possible
- Deadlock detection OS always grant resource requests but periodically OS check for deadlock and take action to break the deadlock

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

References

- **Operating Systems – Internal and Design Principles**
  - By William Stallings
- Chapter 6
- Appendix A

THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA