

2026

CS 332

Subject 12

NJIT

New Jersey's Science &
Technology University

THE EDGE IN KNOWLEDGE

Alex. Gerbessiotis

CS Department



The yellowish/brownish notes are copyright the textbook author.

The bluish notes are copyright the instructor and are add-ons and provide more explanation or additional details



Module 1



Necessary and sufficient

If **S** then **N** evaluates to $\text{Not}(S) \text{ OR } N$

N is necessary for **S**

S is sufficient for **N**

"S: person named Steve" "N: person has name"

'person named Steve' is sufficient for 'person has name'
'person has name' is necessary for 'person named Steve'
(ie if someone has a name it does not necessarily mean his name is Steve)

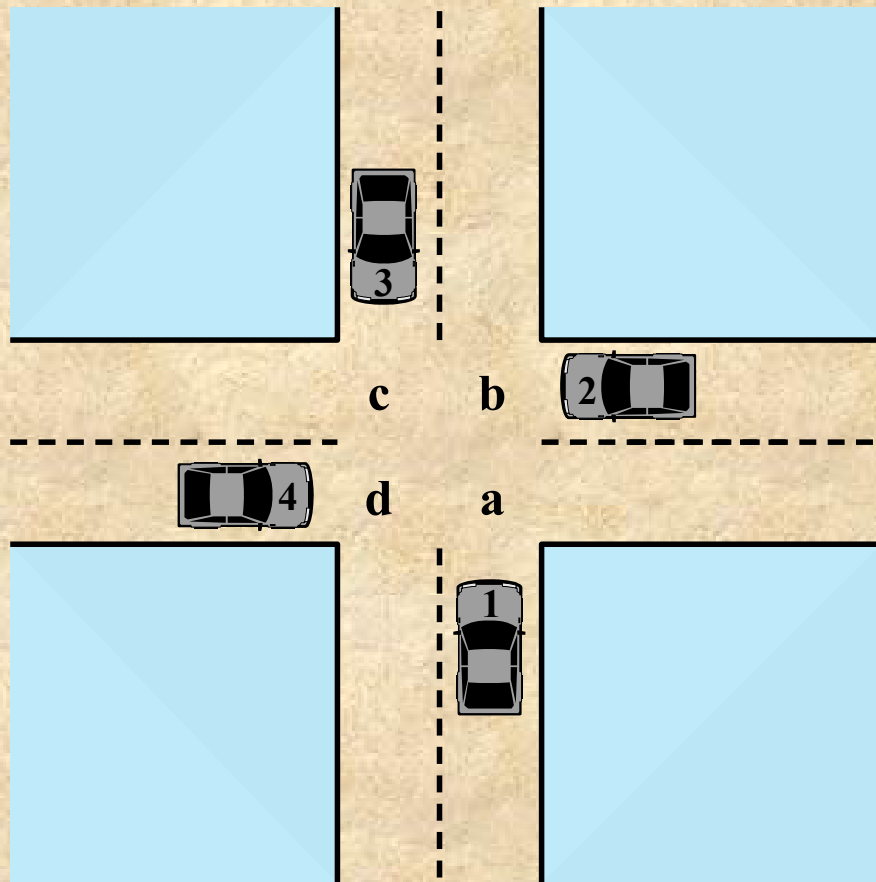
*Operating
Systems:
Internals
and Design
Principles*

Chapter 6 Concurrency: Deadlock and Starvation

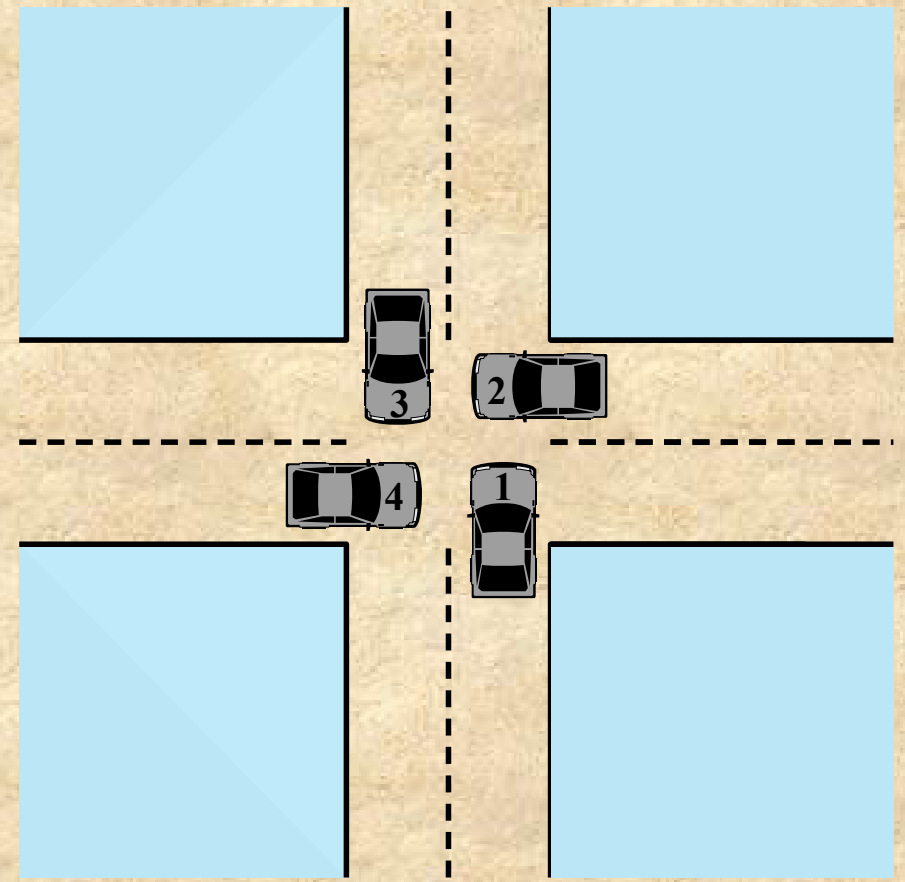
Ninth Edition
By William Stallings

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 because none of the events is ever triggered
- No efficient solution in the general case



(a) Deadlock possible



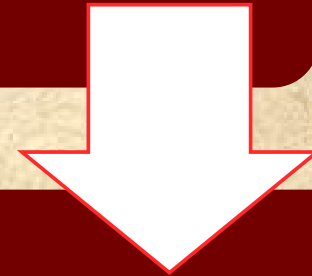
(b) Deadlock

Figure 6.1 Illustration of Deadlock

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

Process P	
Step	Action
p ₀	Request (D)
p ₁	Lock (D)
p ₂	Request (T)
p ₃	Lock (T)
p ₄	Perform function
p ₅	Unlock (D)
p ₆	Unlock (T)

Process Q	
Step	Action
q ₀	Request (T)
q ₁	Lock (T)
q ₂	Request (D)
q ₃	Lock (D)
q ₄	Perform function
q ₅	Unlock (T)
q ₆	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

Example 2: 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;

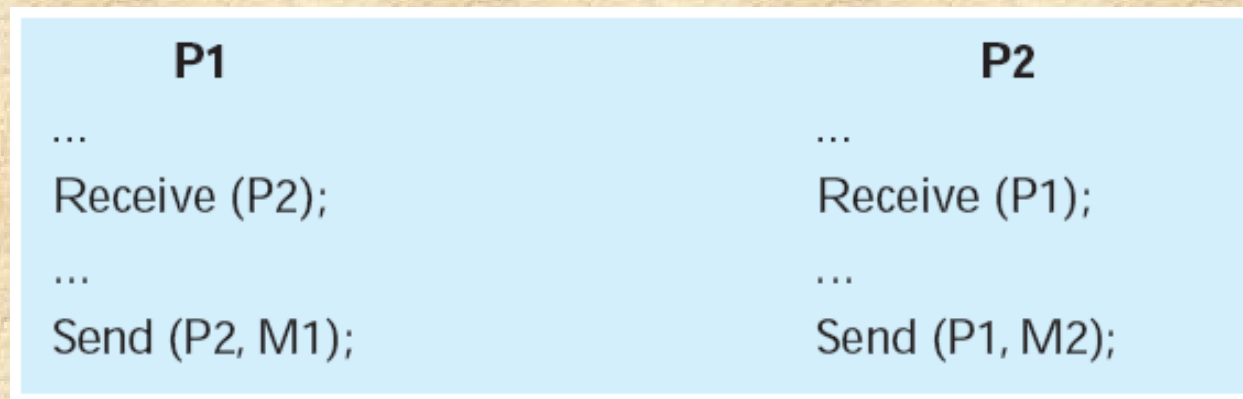
...

Request 80 Kbytes;

- Deadlock occurs if both processes progress to their second request

Consumable Resources 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:



- Deadlock occurs if the Receive is blocking

Deadlock Approaches

- There is no single effective strategy that can deal with all types of deadlock
- Three approaches are common:

- **Deadlock prevention** **P**
 - Disallow one of the three necessary conditions for deadlock occurrence, or prevent circular wait condition from happening

- **Deadlock avoidance** **A**
 - Do not grant a resource request if this allocation might lead to deadlock (eg do not allocate memory:-)

- **Deadlock detection** **D**
 - Grant resource requests when possible, but periodically check for the presence of deadlock and take action to recover (eg suspend a process requesting more memory)



Possibility for Deadlock

1. Mutual Exclusion
2. No preemption
3. Hold+Wait

Existence of Deadlock

1. Mutual Exclusion
2. No preemption.
3. Hold+Wait
4. Circular wait

Prevention:

Avoidance:

Detection:

Policy that eliminates ≥ 1 of 1-4

Dynamic choices based on current situation

Determine conditions 1-4 hold

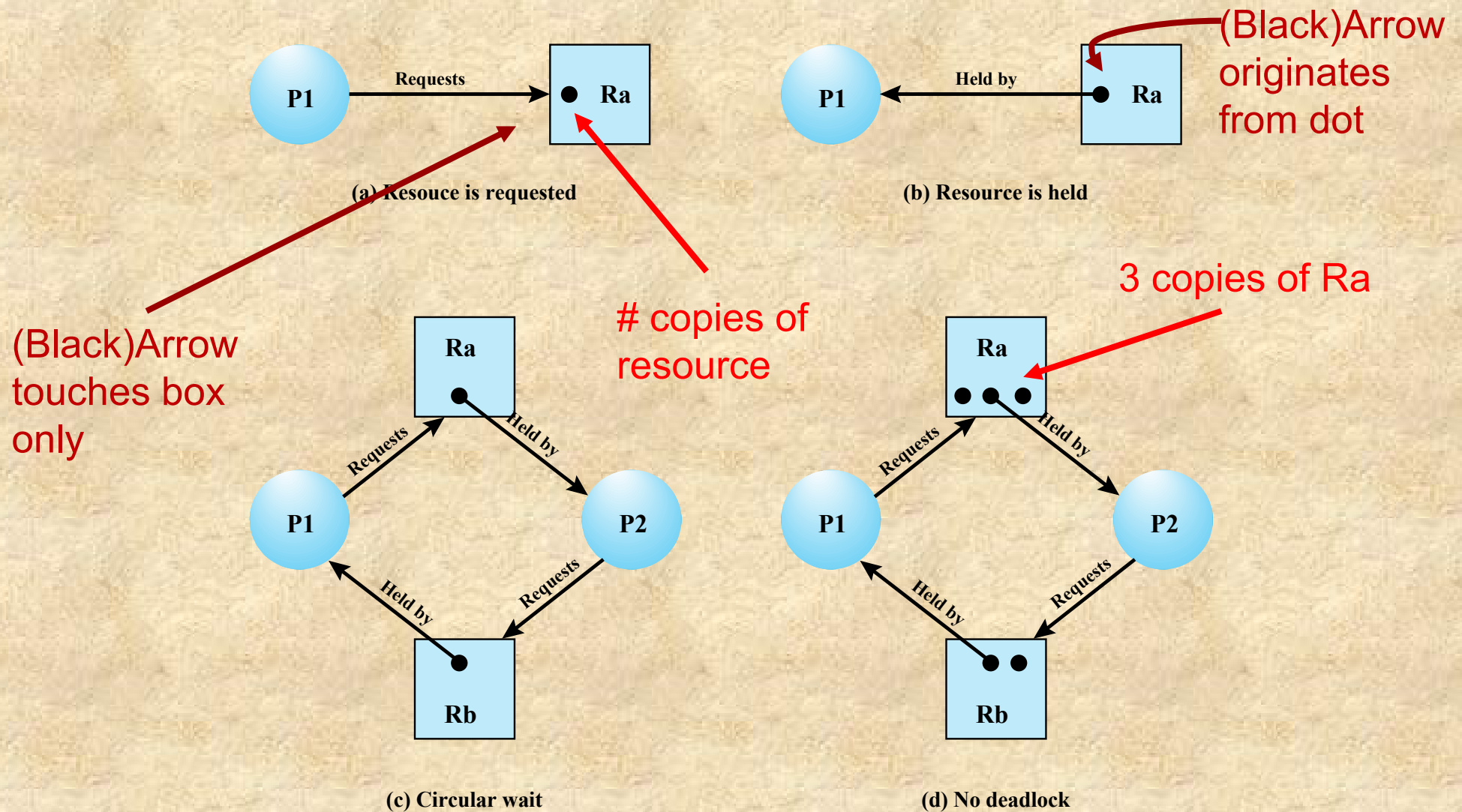


Figure 6.5 Examples of Resource Allocation Graphs

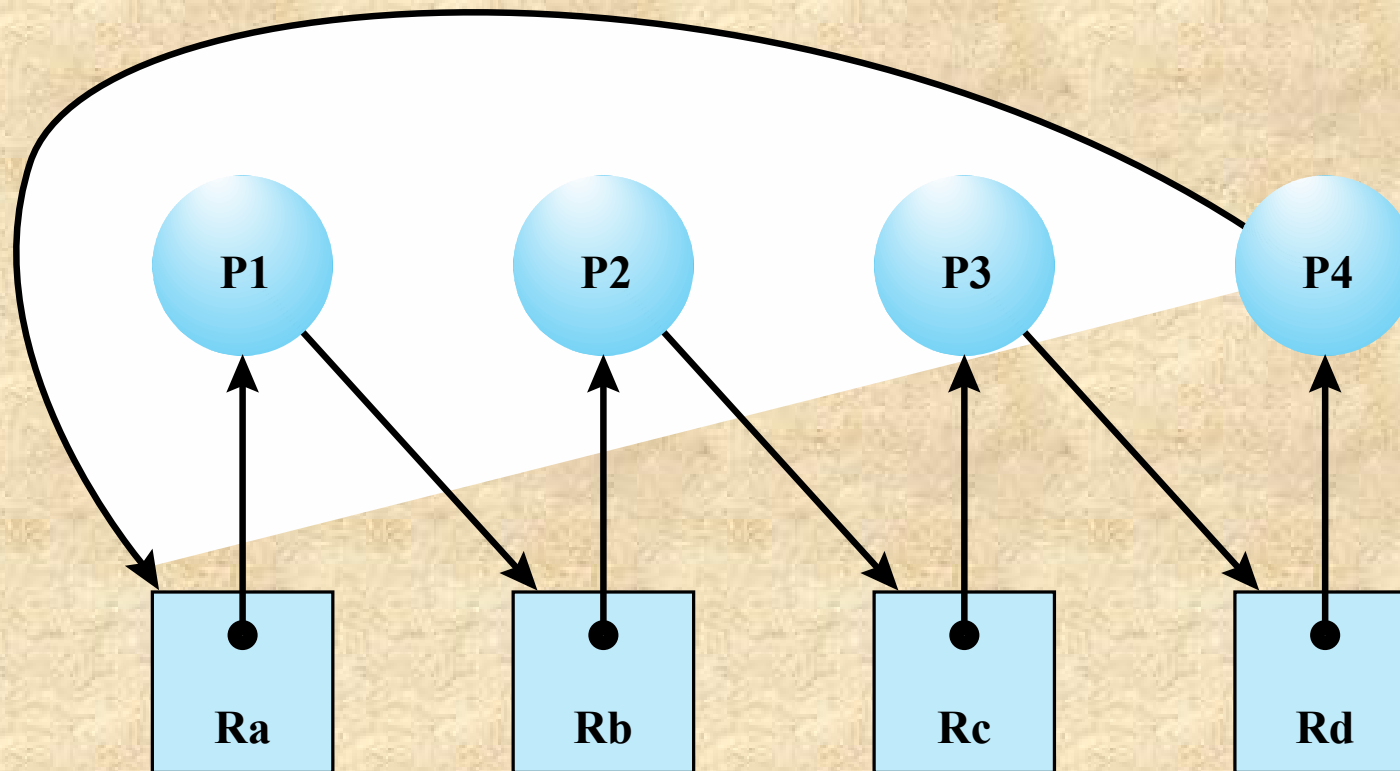


Figure 6.6 Resource Allocation Graph for Figure 6.1b

Conditions for Deadlock

Mutual Exclusion

- Only one process may use a resource at a time
- No process may access a resource that has been allocated to another process

1

Hold-and-Wait

- A process may hold allocated resources while awaiting assignment of other resources

2

No Pre-emption

- No resource can be forcibly removed from a process holding it

3

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

4

Deadlock Prevention Strategy



- Design a system in such a way that the possibility of deadlock is excluded
- Two main methods:
 - Indirect **i.e. 1,2,3**
 - Prevent the occurrence of one of the three necessary conditions
 - Direct
 - Prevent the occurrence of a circular wait **i.e. 4**

Deadlock Condition Prevention



- Mutual exclusion
 - If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the OS
 - Some resources, such as files, may allow multiple accesses for reads but only exclusive access for writes
 - Even in this case, deadlock can occur if more than one process requires write permission
- Hold and wait
 - Can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously

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
- OS may preempt the second process and require it to release its resources

- Circular Wait

- The circular wait condition can be prevented by defining a linear ordering of resource types

Deadlock Avoidance^A

- Allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached
- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached
- **Requires knowledge of future process requests**

Two Approaches to Deadlock Avoidance

A

A₁

Resource Allocation Denial

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

Deadlock Avoidance

A₂

Process Initiation Denial

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

Resource Allocation Denial

A₁A

- Referred to as the *banker's algorithm*
- **State** of the system reflects the current allocation of resources to processes
- **Safe state** is one in which there is at least one sequence of resource allocations to processes that does not result in a deadlock
- **Unsafe state** is a state that is not 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

Figure 6.7 Determination of a Safe State

A₁A

	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

Figure 6.7 Determination of a Safe State

	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

Figure 6.7 Determination of a Safe State

	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

Figure 6.7 Determination of a Safe State

	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

	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	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	1	2	1
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
0	1	1

Available vector V

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

Figure 6.8 Determination of an Unsafe State

```

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 allocation algorithm

```

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

A

Deadlock Avoidance Advantages

- It is not necessary to preempt and rollback processes, as in deadlock detection
- It is less restrictive than deadlock prevention



Deadlock Avoidance Restrictions

- 
- Four red chevrons pointing downwards are arranged vertically on the left side of the slide, each preceding a bullet point.
- Maximum resource requirement for each process must be stated in advance
 - Processes under consideration must be independent and with no synchronization requirements
 - There must be a fixed number of resources to allocate
 - No process may exit while holding resources

Deadlock Strategies

Deadlock prevention strategies are very conservative

'expect the worst' be proactive

- Limit access to resources by imposing restrictions on processes

Deadlock detection strategies do the opposite

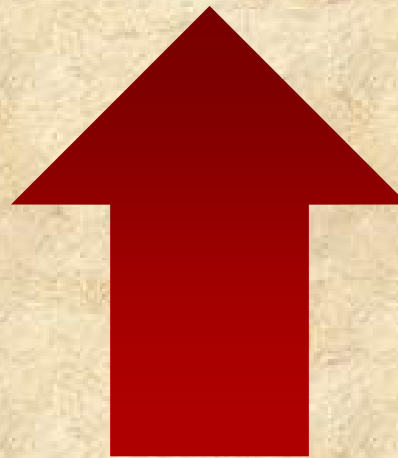
'worst-case analysis'

- Resource requests are granted whenever possible

Deadline Detection Algorithm



- 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



Advantages:

- It leads to early detection
- The algorithm is relatively simple



Disadvantage

- Frequent checks consume considerable processor time

Available vector (in textbook)

	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 R

R1	R2	R3	R4	R5
0	0	0	0	1

Allocation vector W

Figure 6.10 Example for Deadlock Detection

Step 1: Mark A rows with * if all 0 zeroes (eg P4)

Step 2: Add columns of A $Ac = [2 \ 1 \ 1 \ 2 \ 0]$. We must have $Ac < R$ $[2 \ 1 \ 1 \ 2 \ 0] < [2 \ 1 \ 1 \ 2 \ 1]$

Step 3: Compute $W = R - Ac = [0 \ 0 \ 0 \ 0 \ 1]$ i.e. available vector.

Step 4: Find in Q a row $< W$ that is unmarked : P3. Mark P3 add P3 of A to W

Step 5: Now $W = [0 \ 0 \ 0 \ 1 \ 1]$ P1 and P2 are unmarked::: P1,P2 deadlocked

Objective: Mark Processes that are not part of a deadlocked set.

Recovery Strategies



- **Abort all deadlocked processes** (if it ain't broke ...)
- Back up each deadlocked process to some previously defined checkpoint and restart all processes (**restoration point**)
- Successively abort deadlocked processes until deadlock no longer exists (aka ... **Cross your Fingers ... or ...Trial and Error**)
- Successively preempt resources until deadlock no longer exists

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)

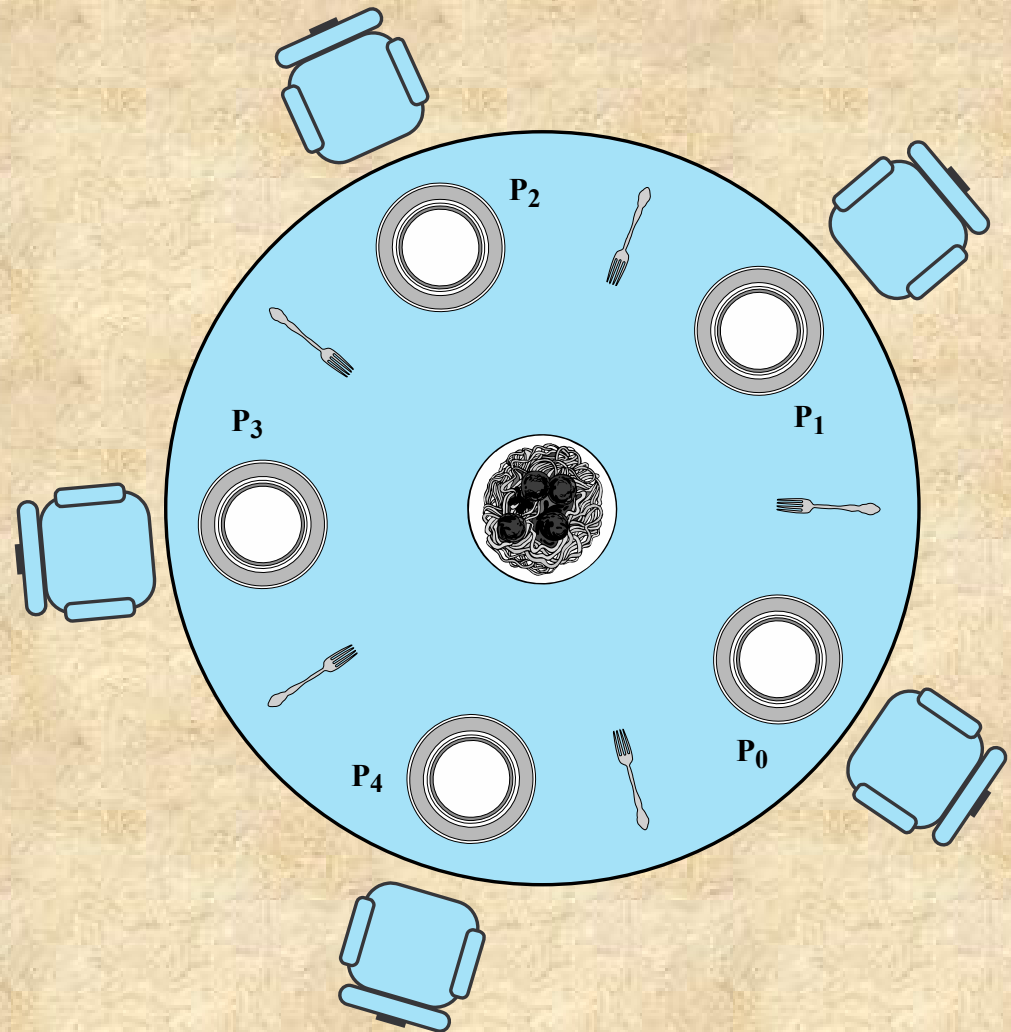


Figure 6.11 Dining Arrangement for Philosophers

```

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

UNIX Concurrency Mechanisms

- UNIX provides a variety of mechanisms for interprocessor communication and synchronization including:

Pipes

Messages

Shared
memory

Semaphores

Signals



Skipped (Excluded) Sections 6.8 through 6.12

NJIT

A white curved line that starts under the 'N', arches over the 'J' and 'I', and ends under the 'T'.

New Jersey's Science &
Technology University

THE EDGE IN KNOWLEDGE

Alex. Gerbessiotis

CS Department