MONASH University

Information Technology

FIT2100 Operating Systems

ADVANCED CONCURRENCY

Week 8
Semester 2 2024
(Reading: Tanenbaum: Chapter 2, Stallings: Chapter 5, 6)

Dr Charith Jayasekara
Faculty of Information Technology

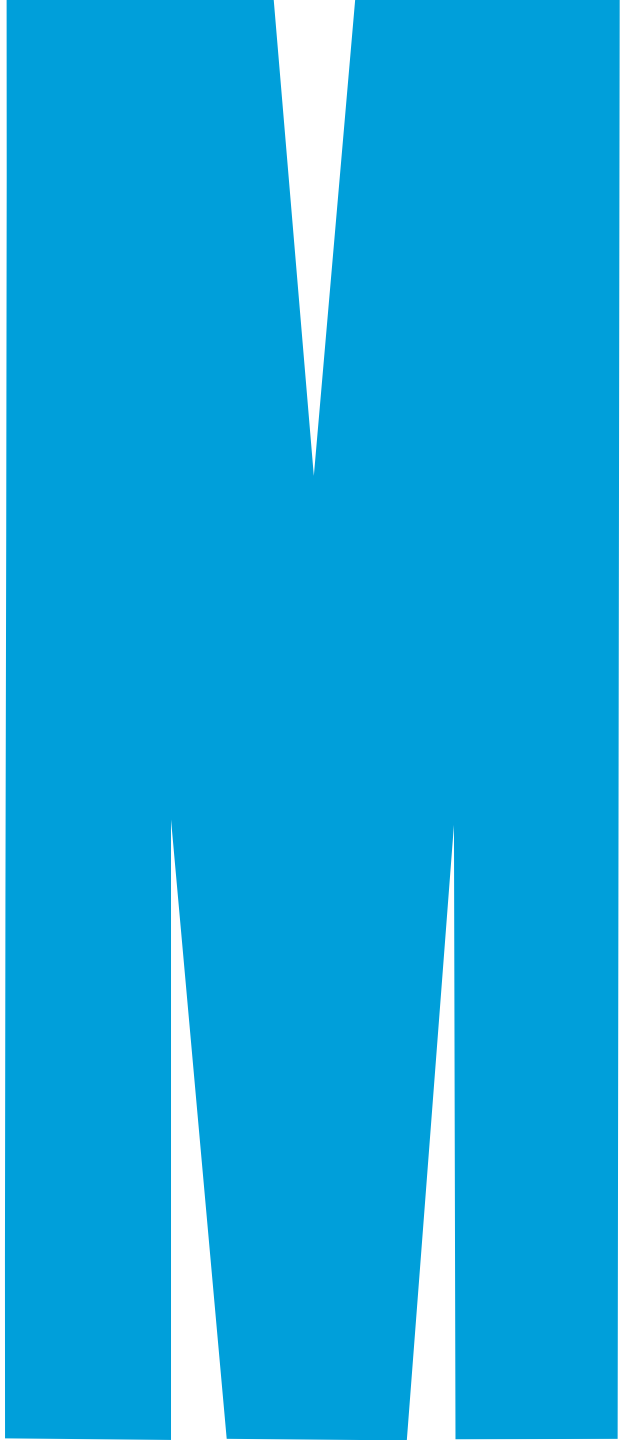
OUTLINE

- Concurrency Problems
- Semaphores
- MUTEX
- Deadlock Basics
- Deadlock Conditions
- Dealing with Deadlocks

LEARNING OUTCOMES

- Upon the completion of Module 8, you should be able to:
  - Discuss different concurrency mechanisms
  - Describe how **semaphores** and **mutexes** support **mutual exclusion**
  - Understand the conditions of **deadlock**
  - Explain three common approaches to **deal** with deadlock

CONCURRENCY PROBLEMS



CONCURRENCY PROBLEMS

EDSGER W. DIJKSTRA

- The problem of concurrent processing was first identified and solved by Dijkstra.
- "Solution of a Problem in Concurrent Programming Control" (1965)

CONCURRENCY PROBLEMS

BASICS (REVISIT)

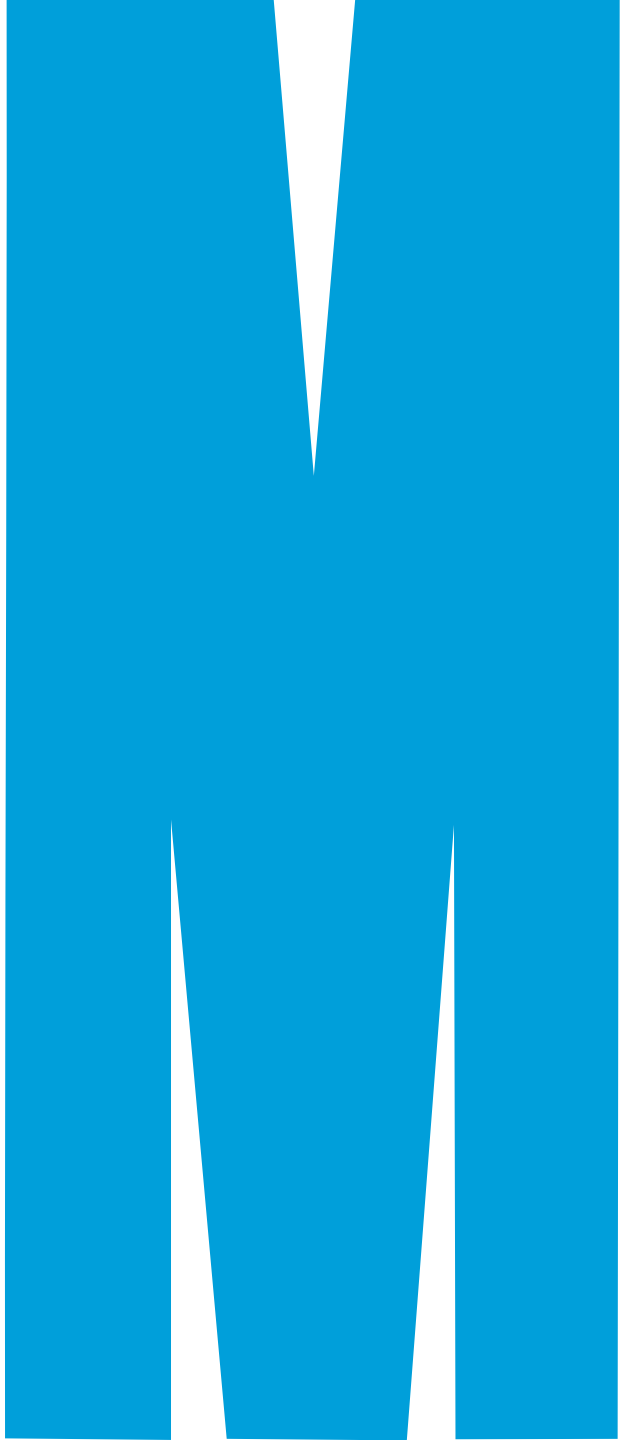
- **Concurrency** is the fundamental concern in supporting multiprogramming, multiprocessing, and distributed processing.
- **Race condition** occurs when multiple processes or threads read and write data items concurrently.
- **Mutual exclusion** is the condition where there is a set of concurrent processes — only one of which is able to access a given resource or perform a given function at at any time.

CONCURRENCY PROBLEMS

CONCURRENCY CONTROL MECHANISMS

- To avoid race conditions when two operations run in parallel, **critical regions (or sections)** of code have to be **serialised**
  - Make one thread/process wait for another
  - Create an imaginary ‘resource’ that only one process may hold at a time
- Busy Waiting
- Semaphores
- Mutexes

BUSY WAITING



BUSY WAITING

MUTUAL EXCLUSION WITH BUSY WAITING: STRICT ALTERNATION

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

A proposed solution to the critical region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the *while* statements.

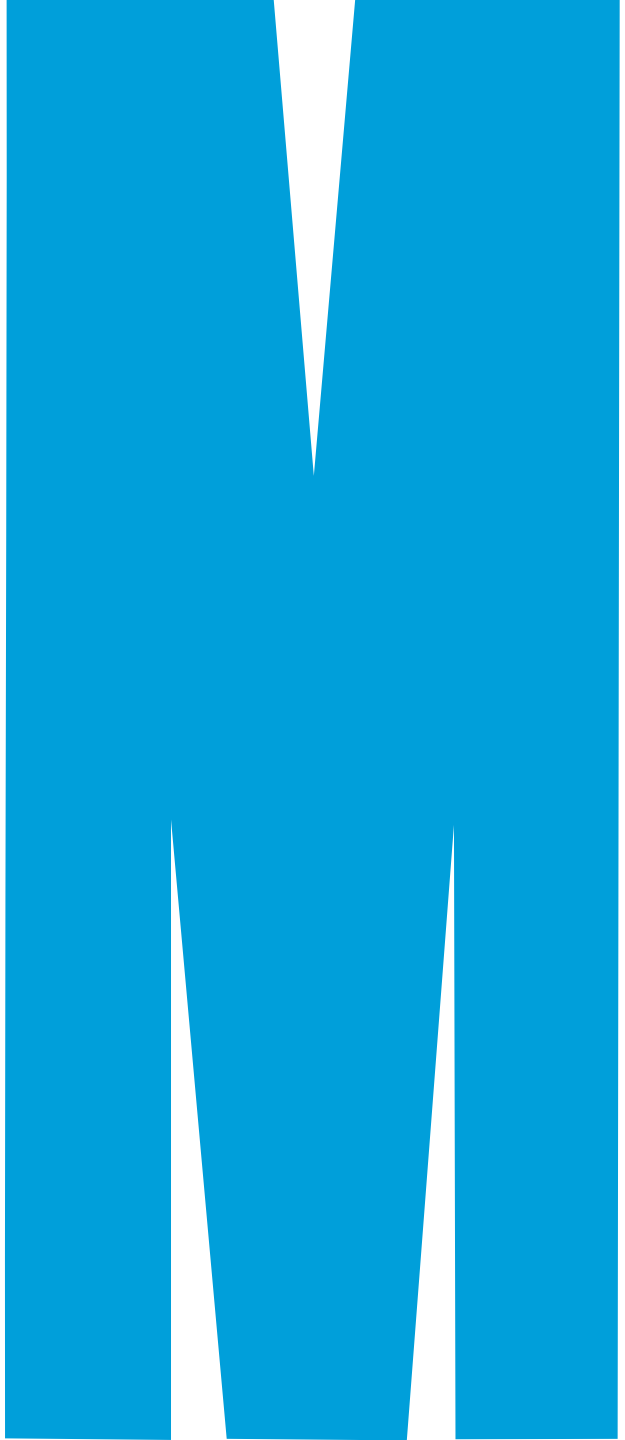
BUSY WAITING

MUTUAL EXCLUSION WITH BUSY WAITING: PETERSON'S SOLUTION

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Peterson's solution for achieving mutual exclusion.

SEMAPHORES



SEMAPHORES

WHAT ARE SEMAPHORES

- A **semaphore** is a variable and a mechanism to support mutual exclusion.
- There are 2 types of semaphores:
  - **Binary semaphores:** Binary semaphores can take only 2 values (0 or 1). They are used to acquire locks. When a resource is available, the process in charge set the semaphore to 1 else 0.
  - **Counting or general semaphores:** may have value to be greater than one, typically used to allocate resources from a pool of identical resources.

SEMAPHORES

SEMAPHORE OPERATIONS

- A Semaphore can be initialised as an integer value upon which only **two operations** are permitted.

- There is **no way** to manipulate semaphores other than these two **atomic** operations:

  - The **semWait** operation *decrements* the value (alternate name: down)
    - If the value becomes negative, then the process executing the semWait is blocked. Otherwise, the process continues execution.
  - The **semSignal** operation *increments* the value (alternate name: up/post)
    - If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.

SEMAPHORES

ATOMIC OPERATION

- A function or action implemented as a sequence of one or more instructions that appears to be indivisible.
- The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state.
- Atomicity guarantees isolation from concurrent processes.

SEMAPHORES

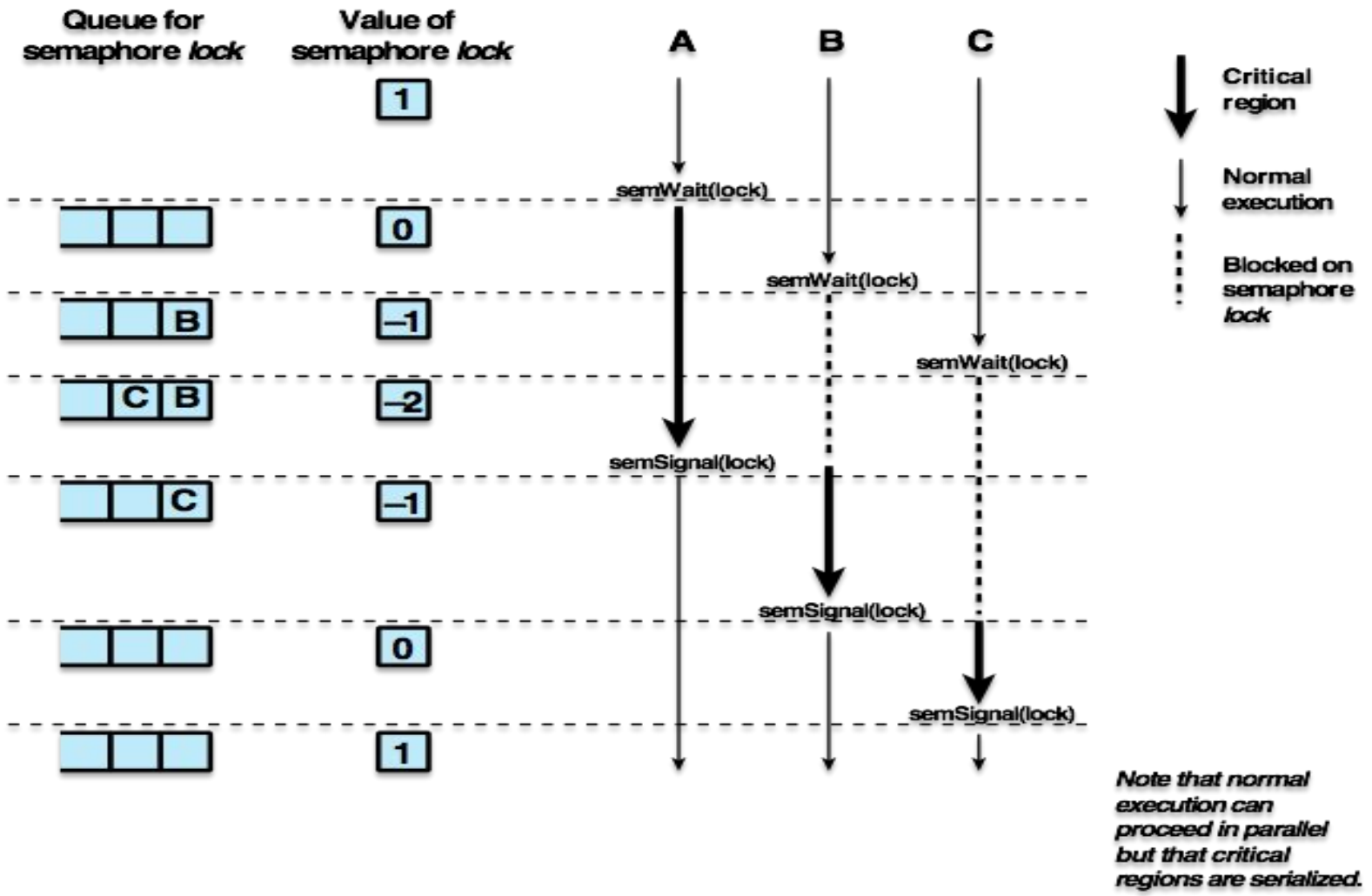
MUTUAL EXCLUSION USING SEMAPHORES

```
/* program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
     while (true) {
          semWait(s);
          /* critical section */;
          semSignal(s);
          /* remainder */;
     }
}
void main()
{
     parbegin (P(1), P(2),…, P(n));
}
```

The semaphore s is initialised to 1. The first process that executes a semWait() will be able to enter the critical section immediately, setting the value of s to 0.

SEMAPHORES

MUTUAL EXCLUSION ENFORCED BY USING A COUNTING SEMAPHORE

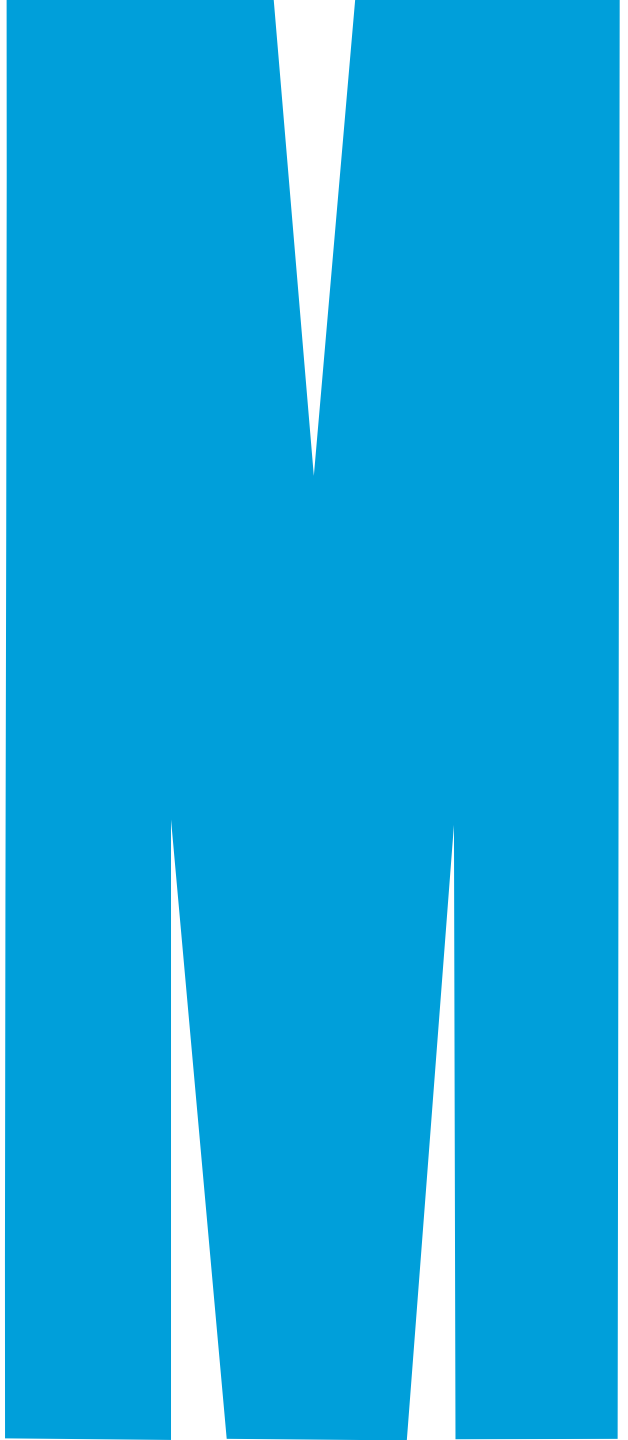


SEMAPHORES

SEMAPHORE CONSEQUENCES

- There is no way to know before a process decrements a semaphore whether it will block or not.
- There is no way to know which process will continue immediately on a uniprocessor system when two processes are running concurrently.
- There is no way to know whether another process is waiting so the number of unblocked processes may be zero or one.

MUTEX



MUTEX

WHAT IS A MUTEX

- A related concept to binary semaphores
- **Mutex (Mutual Exclusion Lock)** is a programming flag:
  - set to 0 when it is locked
  - set to 1 when it is unlocked
  - Only one process (or thread) can hold the lock at a time (others will block on attempting to get the lock)
- **Difference from binary semaphores:**
  - A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).
  - In contrast, it is possible for one process to lock a binary semaphore and for another to unlock it.

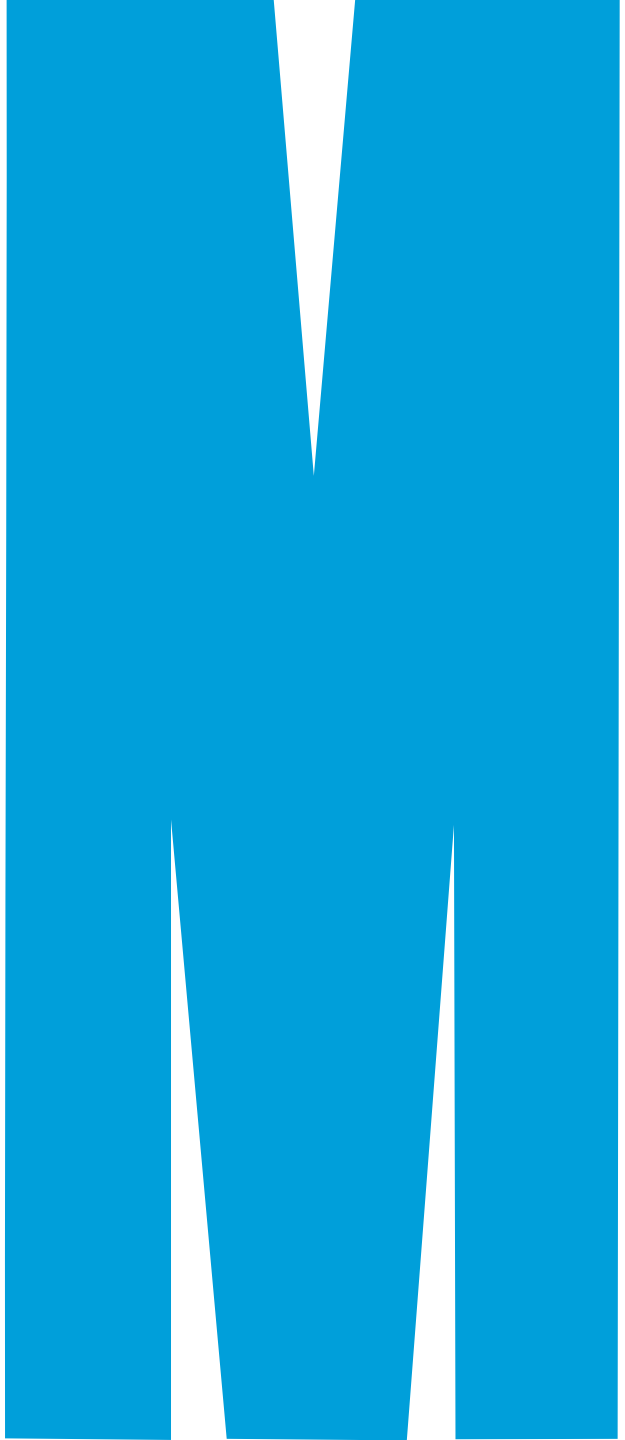
MUTEX

MUTEXES IN PTHREADS

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Some of the Pthreads calls relating to mutexes.

DEADLOCK BASICS



DEADLOCK BASICS

WHAT IS A DEADLOCK

- **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.

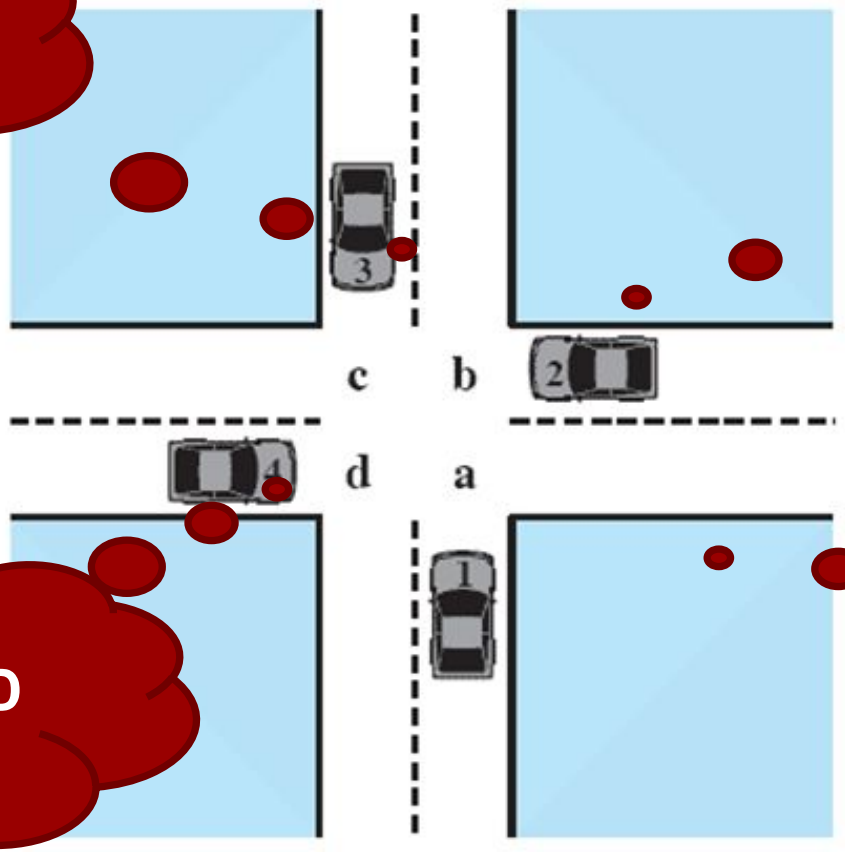
No efficient solution in general.

DEADLOCK BASICS

EXAMPLE: POTENTIAL DEADLOCK

I need quad C and D

I need quad B and C

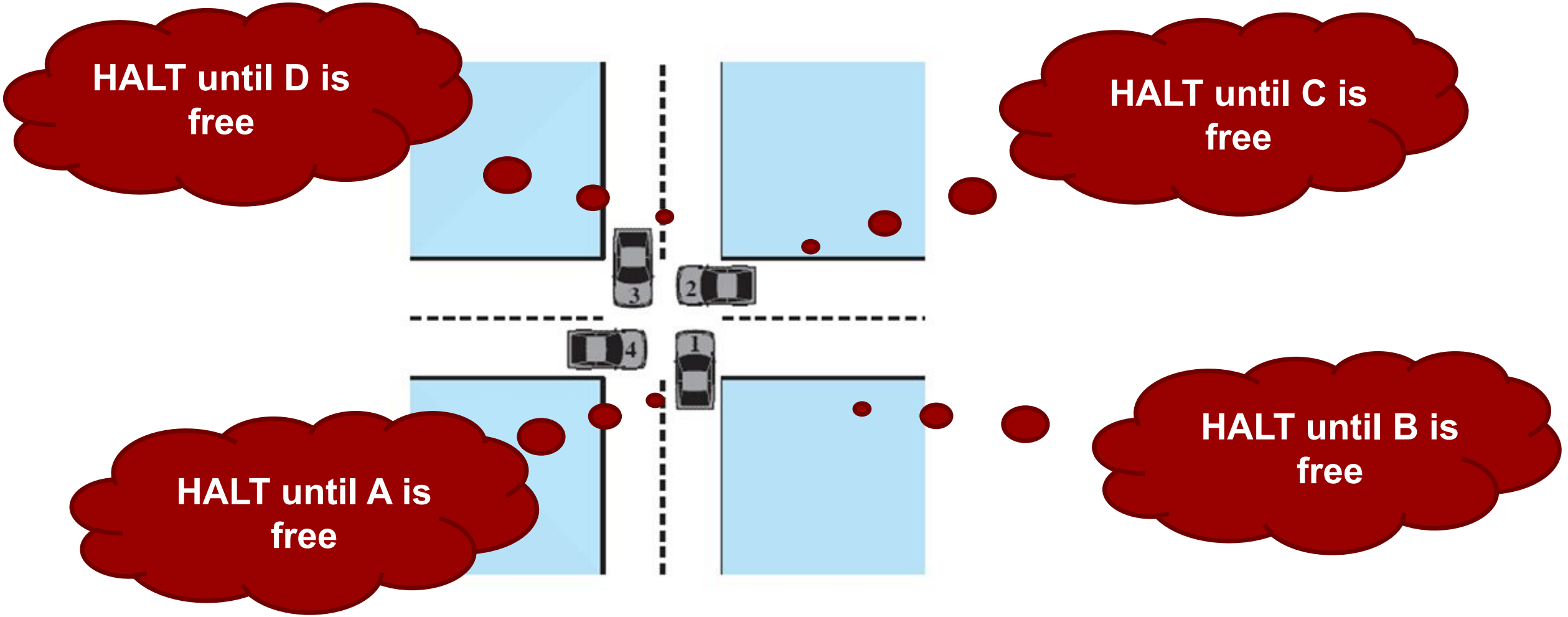


I need quad D and A

I need quad A and B

DEADLOCK BASICS

EXAMPLE: ACTUAL DEADLOCK



DEADLOCK BASICS

RESOURCE ACQUISITION WITH SEMAPHORES

```
typedef int semaphore;
semaphore resource_1;

void process_A(void) {
        down(&resource_1);
        use_resource_1( );
        up(&resource_1);
}
```

(a)

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
}
```

(b)

Figure: Using a semaphore to protect resources.
(a) One resource. (b) Two resources.

DEADLOCK BASICS

DEADLOCK-FREE VS DEADLOCK-POTENTIAL IMPLEMENTATIONS

```
typedef int semaphore;
        semaphore resource_1;
        semaphore resource_2;

        void process_A(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }

        void process_B(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }
```

(a)

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
}

void process_B(void) {
        down(&resource_2);
        down(&resource_1);
        use_both_resources( );
        up(&resource_1);
        up(&resource_2);
}
```

(b)

Figure: (a) Deadlock-free code.
(b) Code with a potential deadlock.

DEADLOCK BASICS

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

DEADLOCK BASICS

REUSABLE RESOURCES: TWO PROCESSES COMPETING

D – Disk resource
T –Tape resource

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)

p0 => p1 => q0 => q1 => p2 => q2 =>…

DEADLOCK BASICS

CONSUMABLE RESOURCES: TWO PROCESSES COMMUNICATING

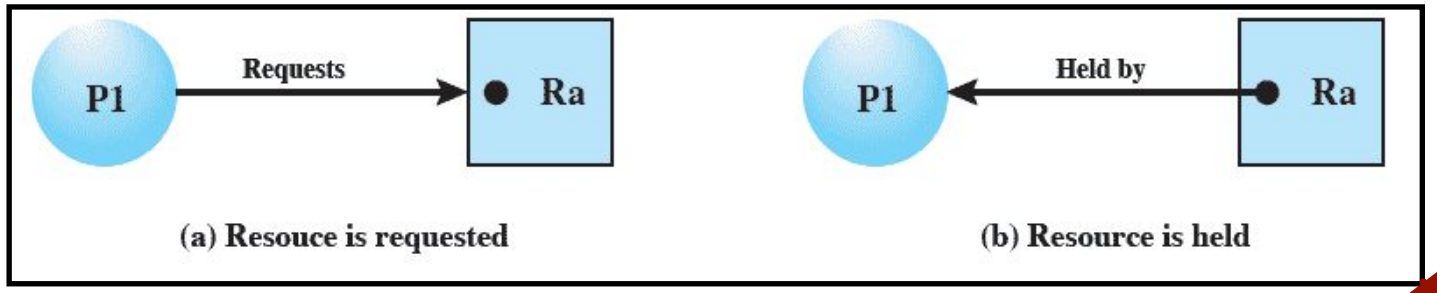
- Consider a pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process:

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

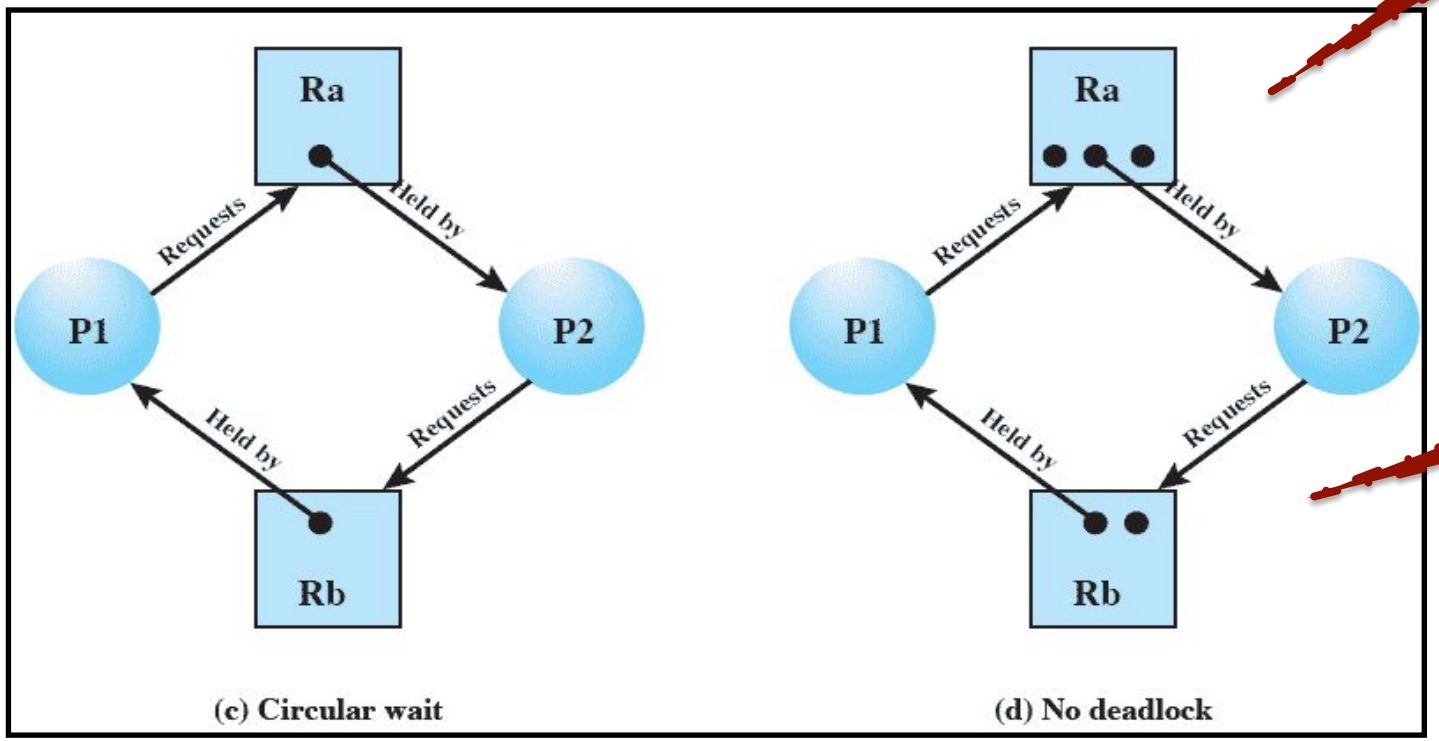
Deadlock occurs if the Receive is blocking (i.e., the receiving process is blocked until the message is received).

DEADLOCK BASICS

RESOURCE ALLOCATION GRAPH



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



(c) Circular wait (d) No deadlock

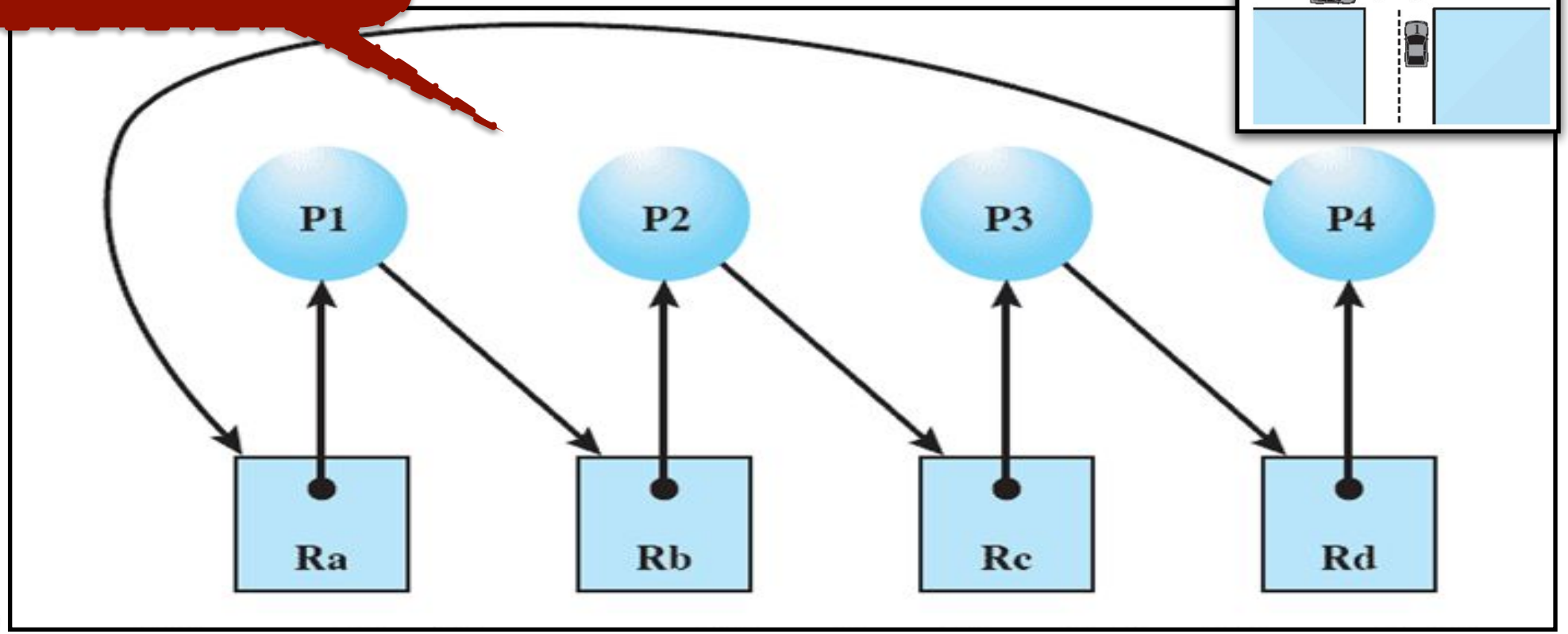
Within a resource node, a dot is shown for each instance of that resource

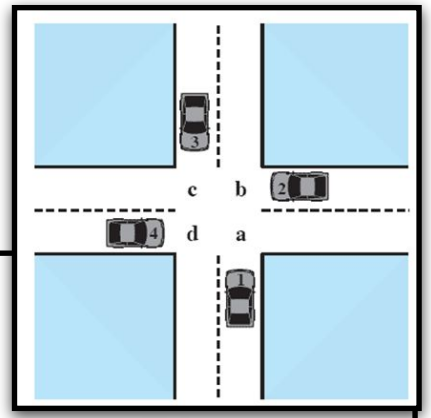
Multiple copies of the same resource

DEADLOCK BASICS

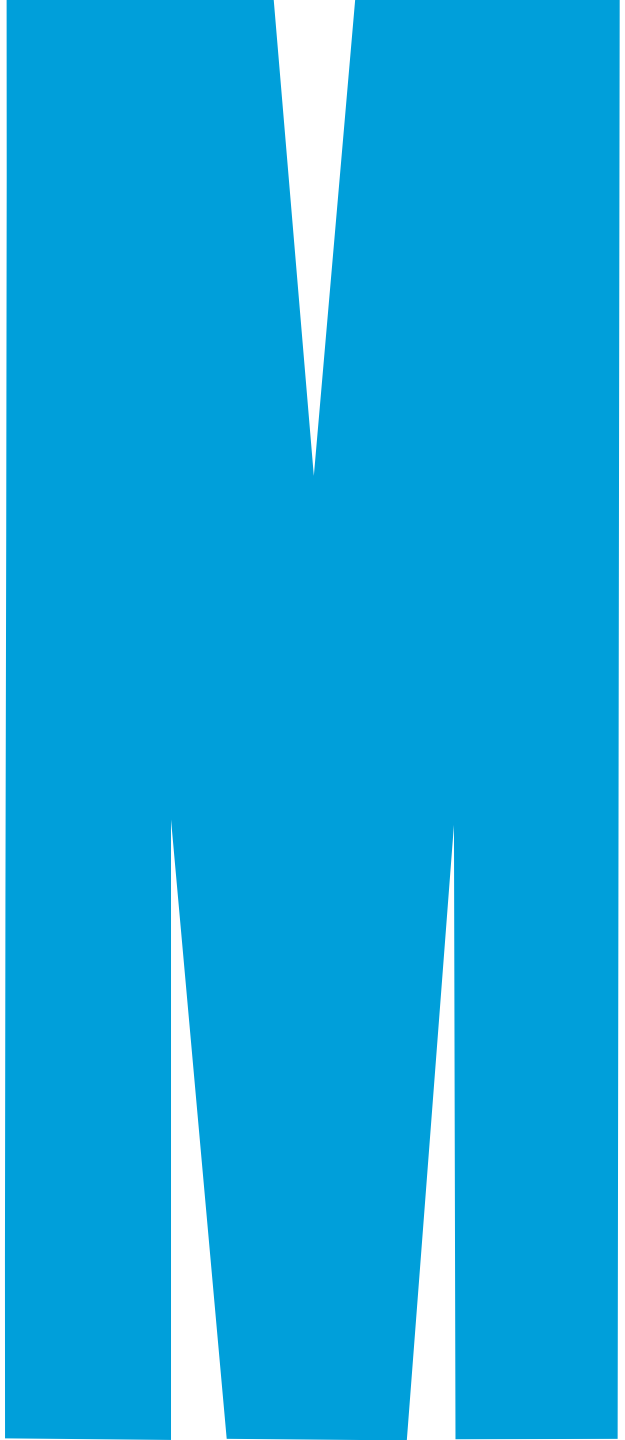
RESOURCE ALLOCATION GRAPH: DEADLOCK EXAMPLE

Circular chain of processes and resources that results in deadlock





DEADLOCK CONDITIONS



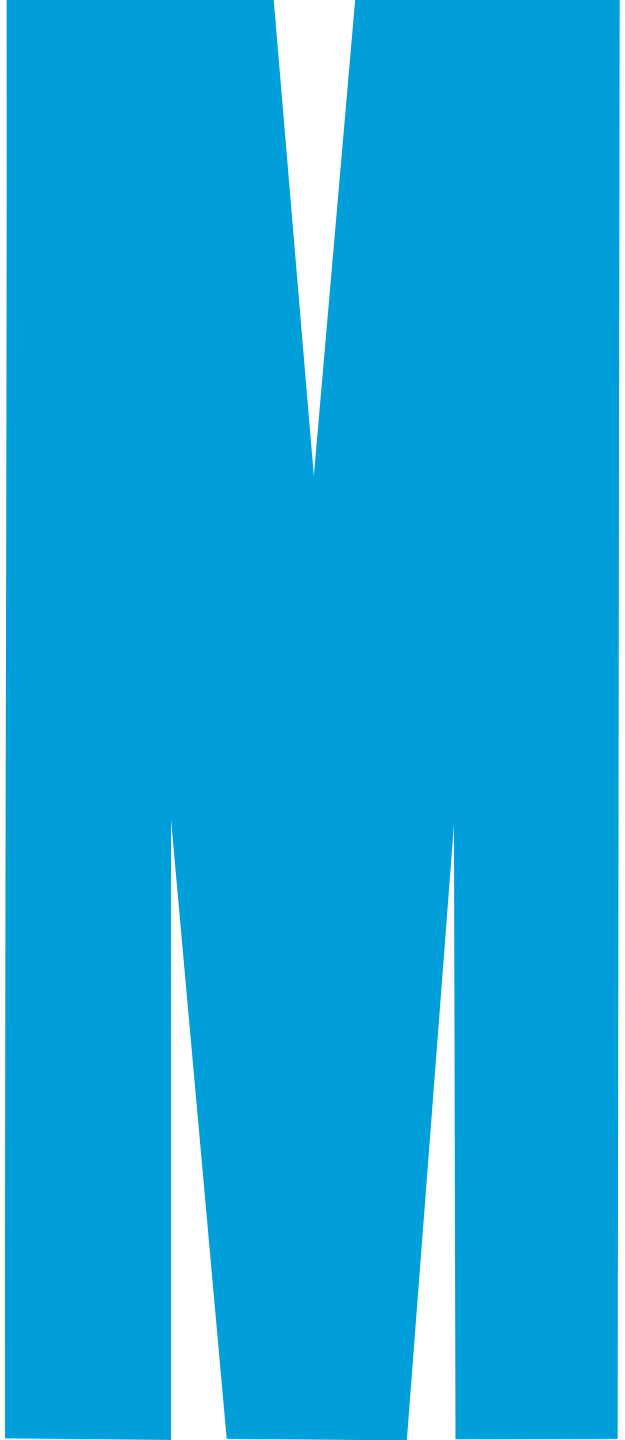
DEADLOCK CONDITIONS

THE FOUR CONDITIONS FOR DEADLOCK

1. **Mutual exclusion:** Only one process may use a resource at a time. No process may access a resource unit that has been allocated to another process.
2. **Hold and wait:** A process may hold allocated resources while awaiting assignment of other resources.
3. **No preemption:** No resource can be forcibly removed from a process holding it.
4. **Circular wait:** A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.

➢ ***The first three conditions are necessary but not sufficient for a deadlock to exist. For deadlock to actually take place, a fourth condition is required.***

DEALING WITH DEADLOCKS

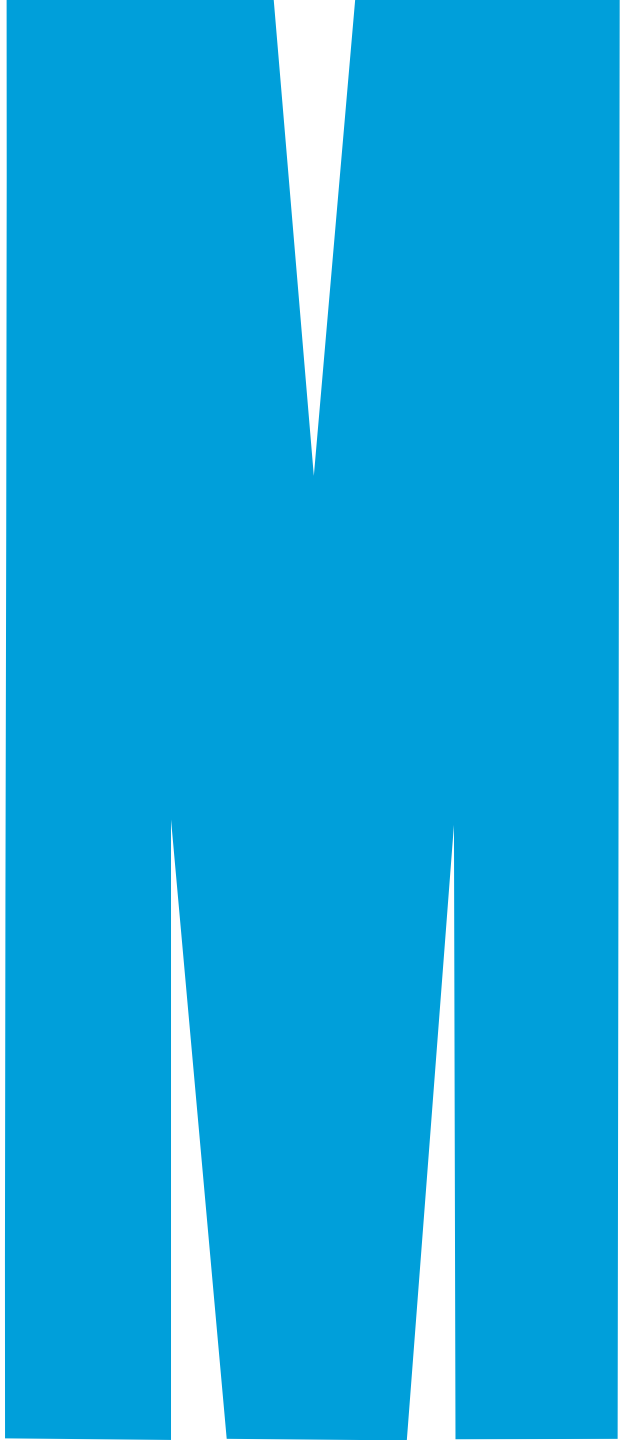


DEALING WITH DEADLOCK

WHAT ARE THE STRATEGIES TO DEAL WITH DEADLOCK

- **Prevention:** adopt a policy that eliminates one of the conditions.
- **Avoidance:** make the appropriate dynamic choices based on the current state of resource allocation.
- **Detection:** attempt to detect the presence of deadlock and take actions to recover when needed.
- **"Ostrich Strategy":** Another (unsafe) strategy is to **ignore** the problem, maybe it will go away!

DEADLOCK PREVENTION



DEADLOCK PREVENTION

HOW TO ELIMINATE DEADLOCK CONDITIONS

- **Mutual Exclusion**
  - No protected access to shared resources.

Should not be disallowed

- **Hold and Wait**
  - A process must request all of its required resources at one time.
  - A process will be blocked until all requests can be granted simultaneously.

Slow down processes and denying resource access

DEADLOCK PREVENTION

HOW TO ELIMINATE DEADLOCK CONDITIONS

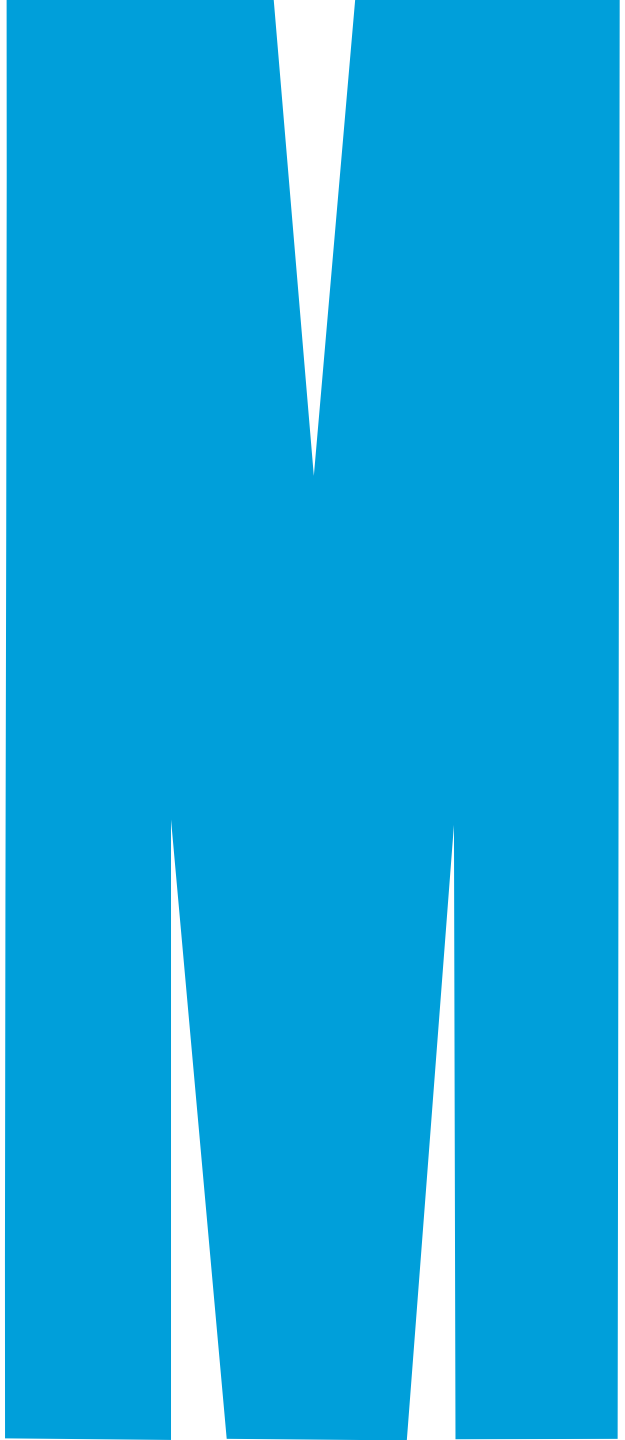
- **No Preemption**
  - If a process holding certain resources is denied a further request, that process must release its original resources and request them again.
  - OS may preempt the second process (which holds the requested resources) and require it to release its resources to the first process.

Only practical when states of resources can be easily saved and restored later (hard with I/O devices)

- **Circular Wait**
  - Define a linear ordering of resource types.

Slow down processes and denying resource access

DEADLOCK AVOIDANCE



DEADLOCK AVOIDANCE

WHAT IS A DEADLOCK AVOIDANCE STRATEGY

- A decision is made **dynamically** whether the current resource allocation request will, if granted, potentially lead to a deadlock.
- Avoidance allows more concurrency than prevention.
- Requires **knowledge** of future process resource requests.
- **Two Approaches:**
  - Process Initiation Denial: do not start a process if its demands might lead to deadlock.
  - Resource Allocation Denial: do not grant an incremental resource request to a process if this allocation might lead to deadlock.

DEADLOCK AVOIDANCE

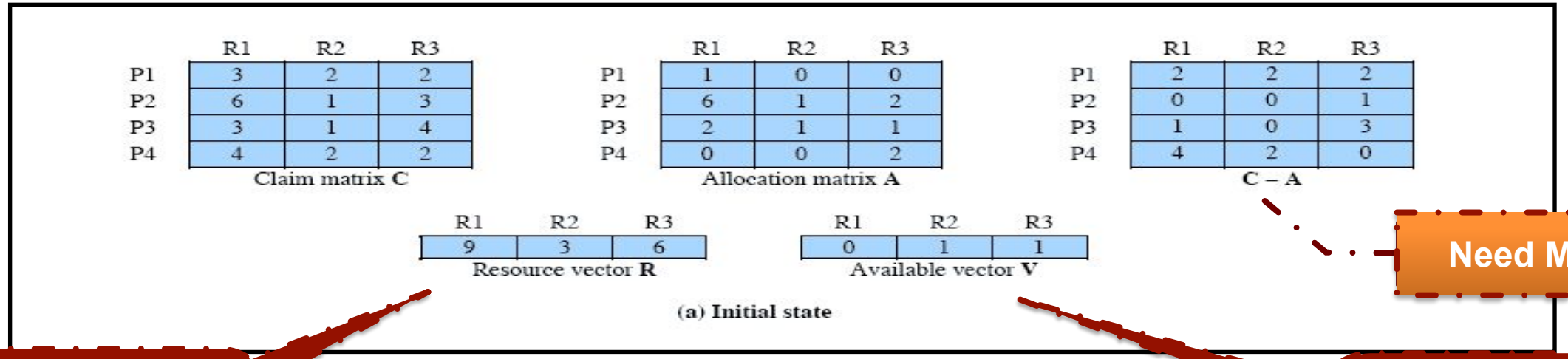
AVOIDANCE STRATEGY: RESOURCE ALLOCATION DENIAL

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

DEADLOCK AVOIDANCE

BANKER'S ALGORITHM: DETERMINATION OF A SAFE STATE

- The state of a system: four processes and three resources
- Allocations have been made to the four processes



	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

Need Matrix

Amount of existing resources

Resources available after allocation

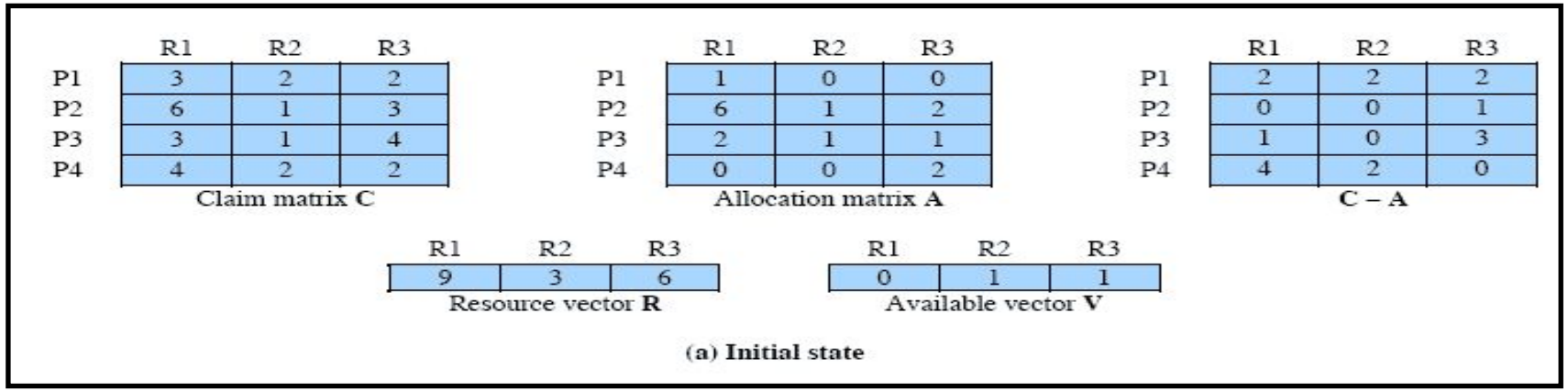
Process P2 makes the remaining resource request of R3

$$C_{ij} - A_{ij} <= V_j \text{ for all } j$$

MONASH University

DEADLOCK AVOIDANCE

BANKER'S ALGORITHM: 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	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

P2 runs to completion and releases its resources

DEADLOCK AVOIDANCE

BANKER'S ALGORITHM: DETERMINATION OF A SAFE 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

	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

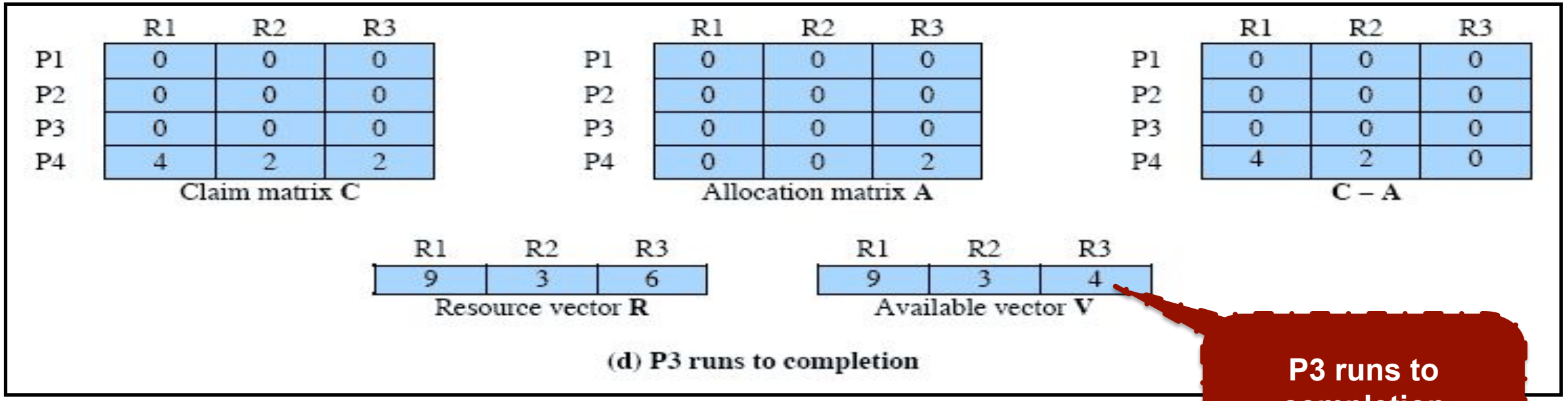
Available vector **V**

(c) P1 runs to completion

P1 makes the request for the remaining resources

DEADLOCK AVOIDANCE

BANKER'S ALGORITHM: 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

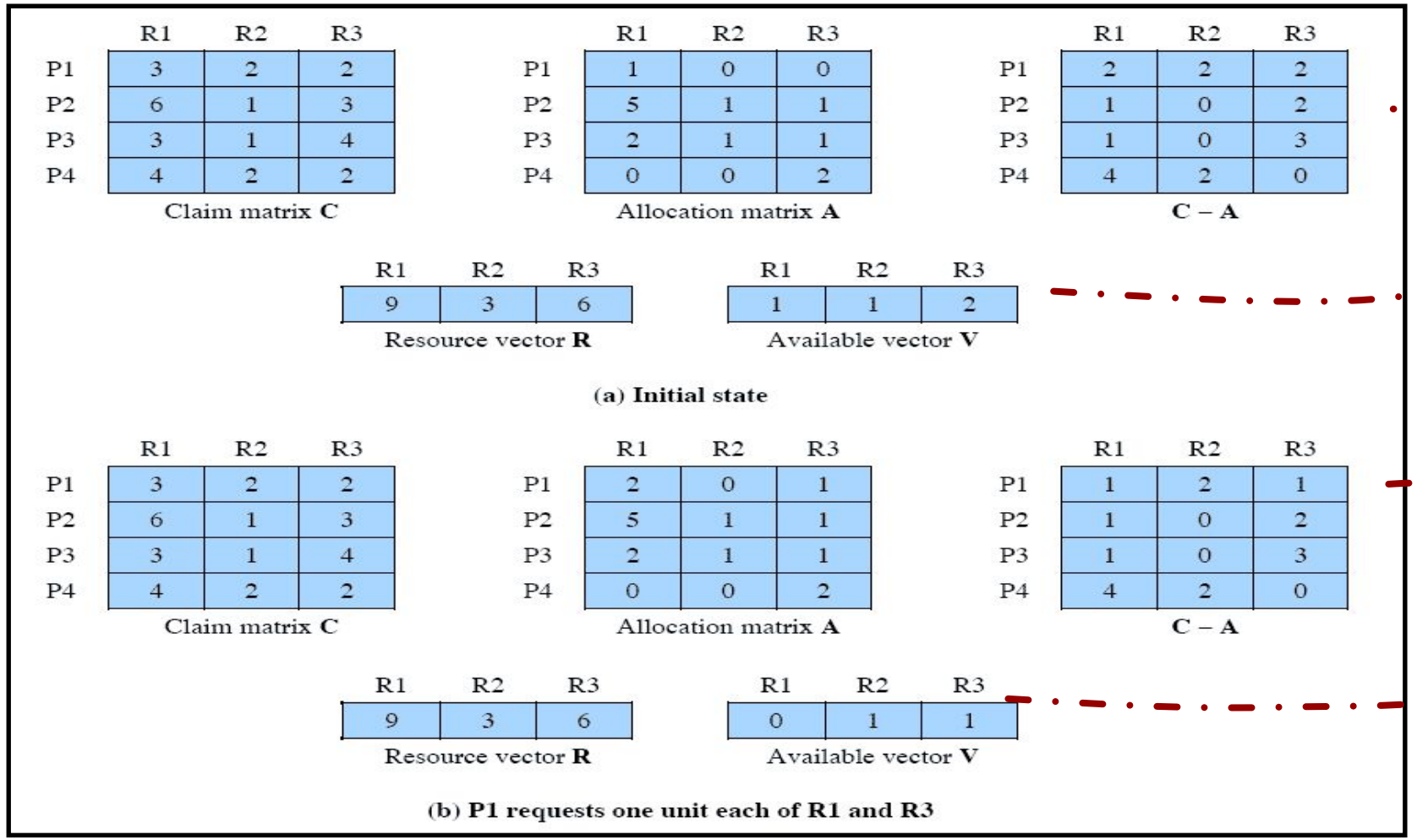
P3 runs to completion

The state defined originally is a safe state

MONASH University

DEADLOCK AVOIDANCE

BANKER'S ALGORITHM: DETERMINATION OF AN UNSAFE 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

SAFE-STATE
P2 1-R1, 2-R3

UNSAFE-STATE
P1 1-R1, 1-R3

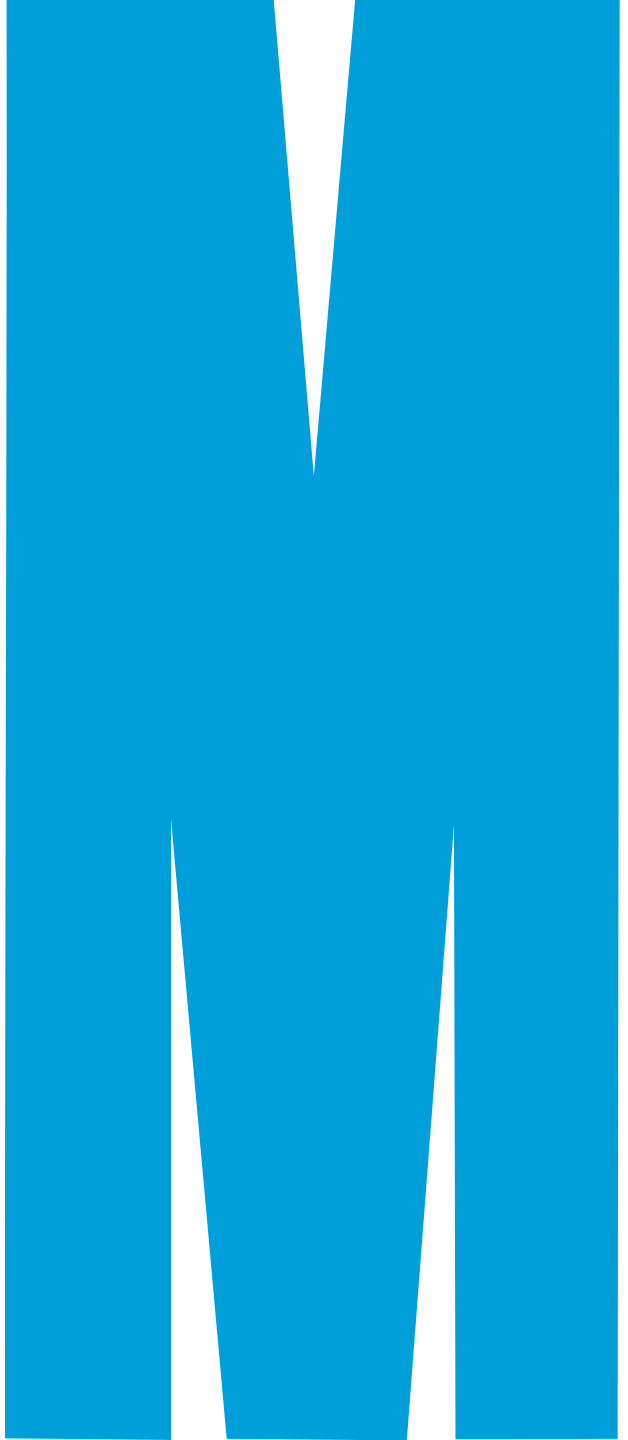
MONASH University

DEADLOCK AVOIDANCE

RESTRICTIONS

- Maximum resource requirement for each process must be stated in advance.
- Processes under consideration must be independent and with no synchronisation requirements.
- There must be a fixed number of resources to allocate.
- No process may exit while holding resources.

DEADLOCK DETECTION



DEADLOCK DETECTION

WHAT IS DEADLOCK DETECTION

- 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** (checking at each resource request):
  - it leads to early detection
  - the algorithm is relatively simple

- **Disadvantage**:
  - frequent checks consume processor time (classical polling problem)

DEADLOCK DETECTION

DEADLOCK DETECTION ALGORITHM

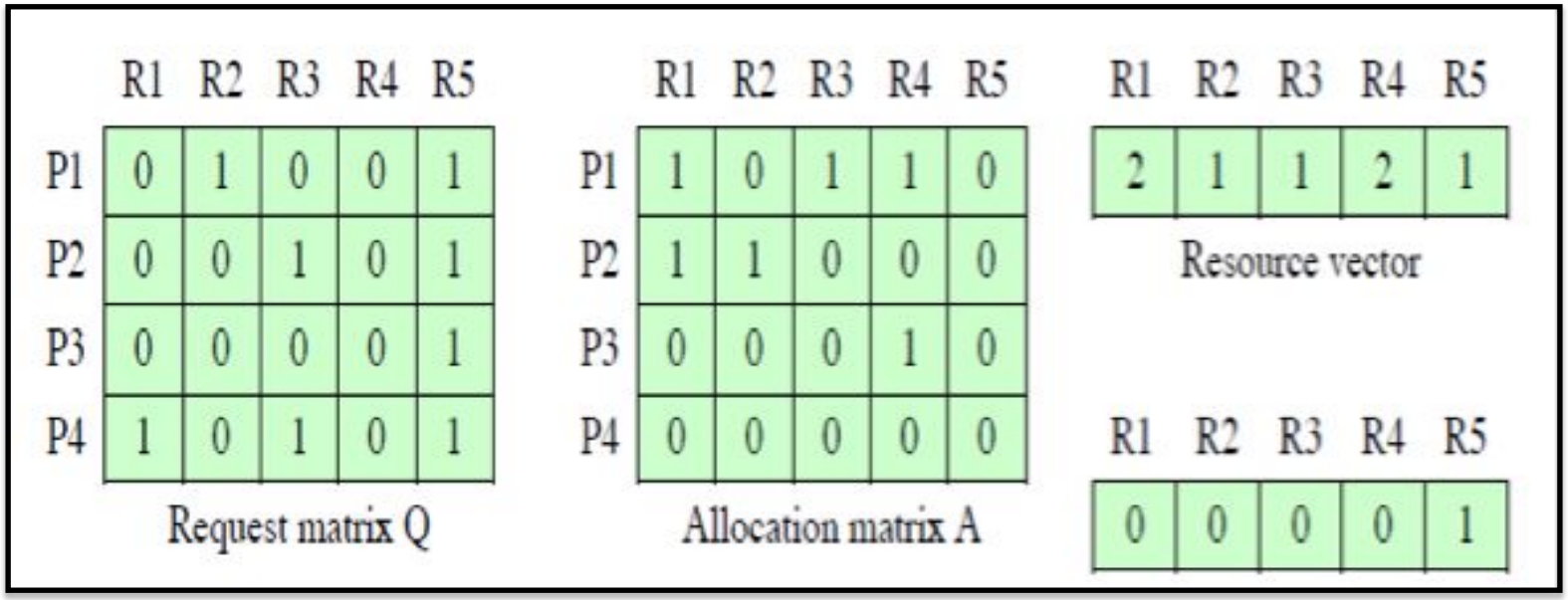
1. Mark each process that has a row in the allocation matrix of all zeros

2. Initialise a temporary vector **W** to equal the **Available** vector.

3. Find the index i such that process i is currently unmarked and the i-th row of **Q** (**request matrix**) is less than or equal to **W**. If no such row is found, terminate the algorithm.
4. If such a row is found, mark process i and add the corresponding row of the allocation matrix to **W**. Return to step 3.

DEADLOCK DETECTION

EXAMPLE: DEADLOCK DETECTION ALGORITHM



	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)

A deadlock exists if and only if there are unmarked processes at the end of the algorithm

DEADLOCK DETECTION

RECOVERY STRATEGIES AFTER DEADLOCK DETECTION

- Abort all deadlocked processes.
- Back up each deadlocked process to some previously defined checkpoint and restart all processes.
- Successively abort deadlocked processes until deadlock no longer exists.
- Successively preempt resources until deadlock no longer exists.

DEADLOCK DETECTION

PREVENTION VS DETECTION

- Deadlock prevention strategies are **conservative.**
  - limit access to resources by imposing restrictions on processes
- Deadlock detection strategies do the **opposite.**
  - resource requests are granted whenever possible

Summary

- So far we have discussed
  - Concurrency Problems
  - Semaphores and Mutex
  - Deadlock Basics
  - Dealing with Deadlock

- Next week
  - Virtual and Shared Memory

- Reading
  - Tanenbaum, Chapter 2 (4th Edition)
  - Stallings, Chapter 6 (7th Edition)