MONASH University

MONASH
INFORMATION
TECHNOLOGY

FIT2100 Semester 2 2020

Lecture 8:
Advanced Concurrency

(Reading: Stallings, Chapter 5 and Chapter 6)

- WEEK 9

GROUP OF EIGHT AUSTRALIA

Lecture 5: Learning Outcomes

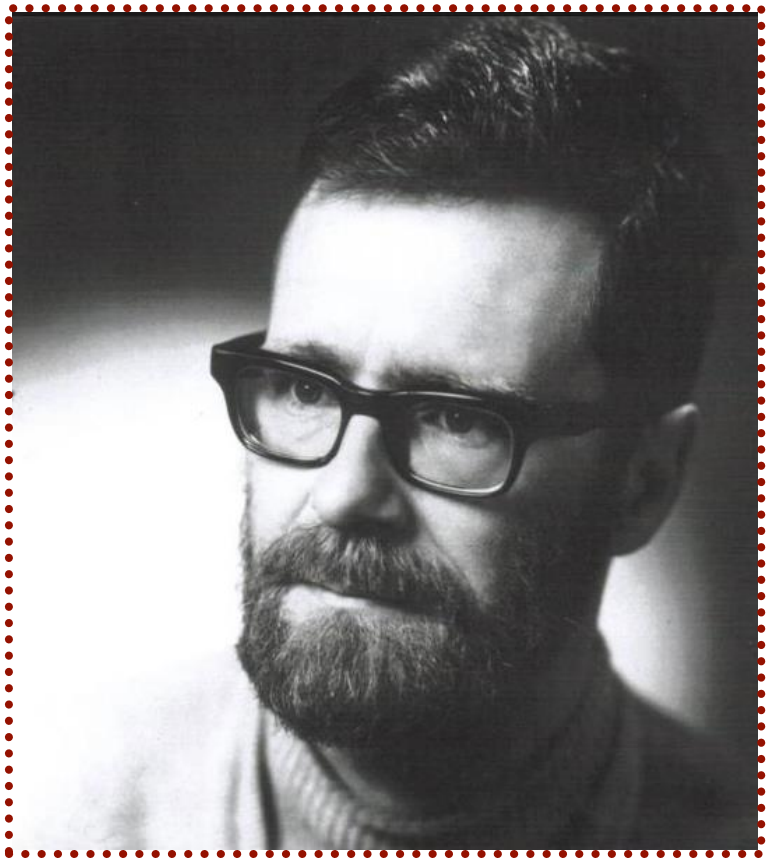
- Upon the completion of this lecture, you should be able to:
  - Discuss different concurrency mechanisms
  - Describe how semaphores support mutual exclusion
  - Understand the conditions of deadlock
  - Explain three common approaches to dealing with deadlock

MONASH University

What we have understood about the problem of *concurrency*?

Edsger W. Dijkstra

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



The Problem of Concurrency

- 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: control mechanisms

- To avoid race conditions when two operations run in parallel, **critical 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
- Semaphores
- Mutexes

What are *semaphores*?
How can semaphores be used to control concurrency problems?

The Concept of Flags

- Set a FLAG when a processing is using the shared resource — others can check that flag and decide to enter or wait in their critical section

- Binary value: FLAG is **ON** (used) or **OFF** (not-used) — as a form of ***semaphore***

Periodic testing for the availability of the flag is wastage of resource; rather the process that returns the flag can send a signal to the process that is waiting for that resource.

The Concept of Semaphores

- An integer variable used for signaling among processes
- Only three operations are allowed on a semaphore:
  - **initialisation**, **increment** or **decrement**

atomic operations

- Two types of semaphores:
  - Binary semaphores which takes on only the values 0 and 1
  - Counting (general) semaphores

Semaphores: Operations

A variable that has an integer value upon which only three operations are defined.

There is no way to inspect or manipulate semaphores other than these three operations.

1. May be *initialised* to a non-negative integer value.
2. The **semWait** operation *decrements* the value.
   - ✦ If the value becomes negative, then the process executing the semWait is blocked. Otherwise, the process continues execution.
3. The **semSignal** operation *increments* the value.
   - ✦ If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.

Counting Semaphores: Definition

```
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0) {
            /* place this process in s.queue */;
            /* block this process */;
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0) {
            /* remove a process P from s.queue */;
            /* place process P on ready list */;
      }
}
```

Binary Semaphores: Definition

```
struct binary_semaphore {
      enum {zero, one} value;
      queueType queue;
};
void semWaitB(binary_semaphore s)
{
      if (s.value == one)
            s.value = zero;
      else {
               /* place this process in s.queue */;
               /* block this process */;
      }
}
void semSignalB(semaphore s)
{
      if (s.queue is empty())
            s.value = one;
      else {
               /* remove a process P from s.queue */;
               /* place process P on ready list */;
      }
}
```

Continue execution

Semaphores: 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

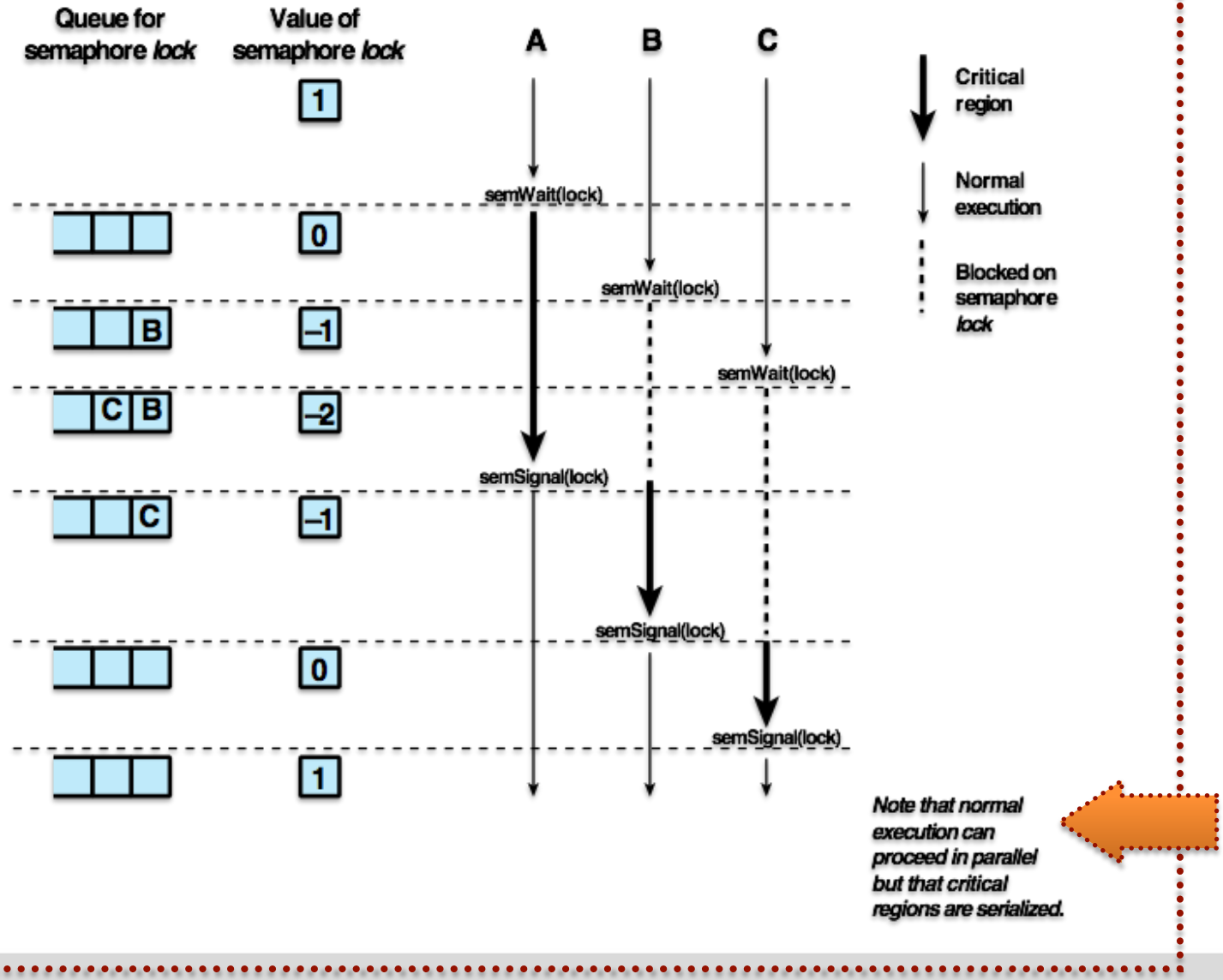
You do not know whether another process is waiting so the number of *unblocked processes may be zero or one*

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.

Mutual Exclusion: Shared Data Protected by a Semaphore



MONASH
University

WHAT IS A MUTEX?

Mutex: Mutual Exclusion Lock

- A related concept to binary semaphores
  - Very similar, but with a specific use case.

- Mutex 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:**
  - The process/thread that locks the mutex must be the one to unlock it

What is a *deadlock*? (Another concurrency problem)

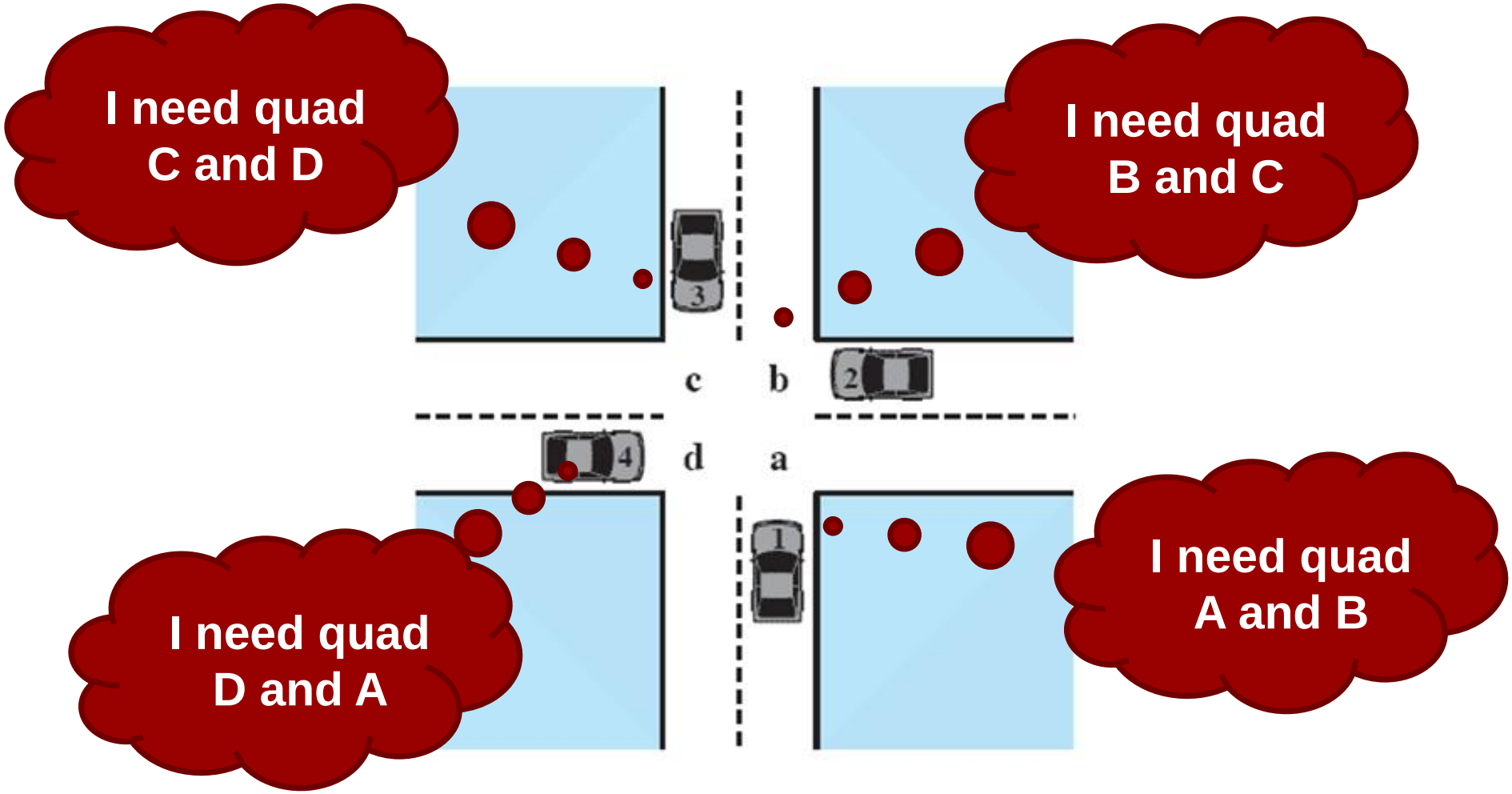
The Concept of Deadlock

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other.

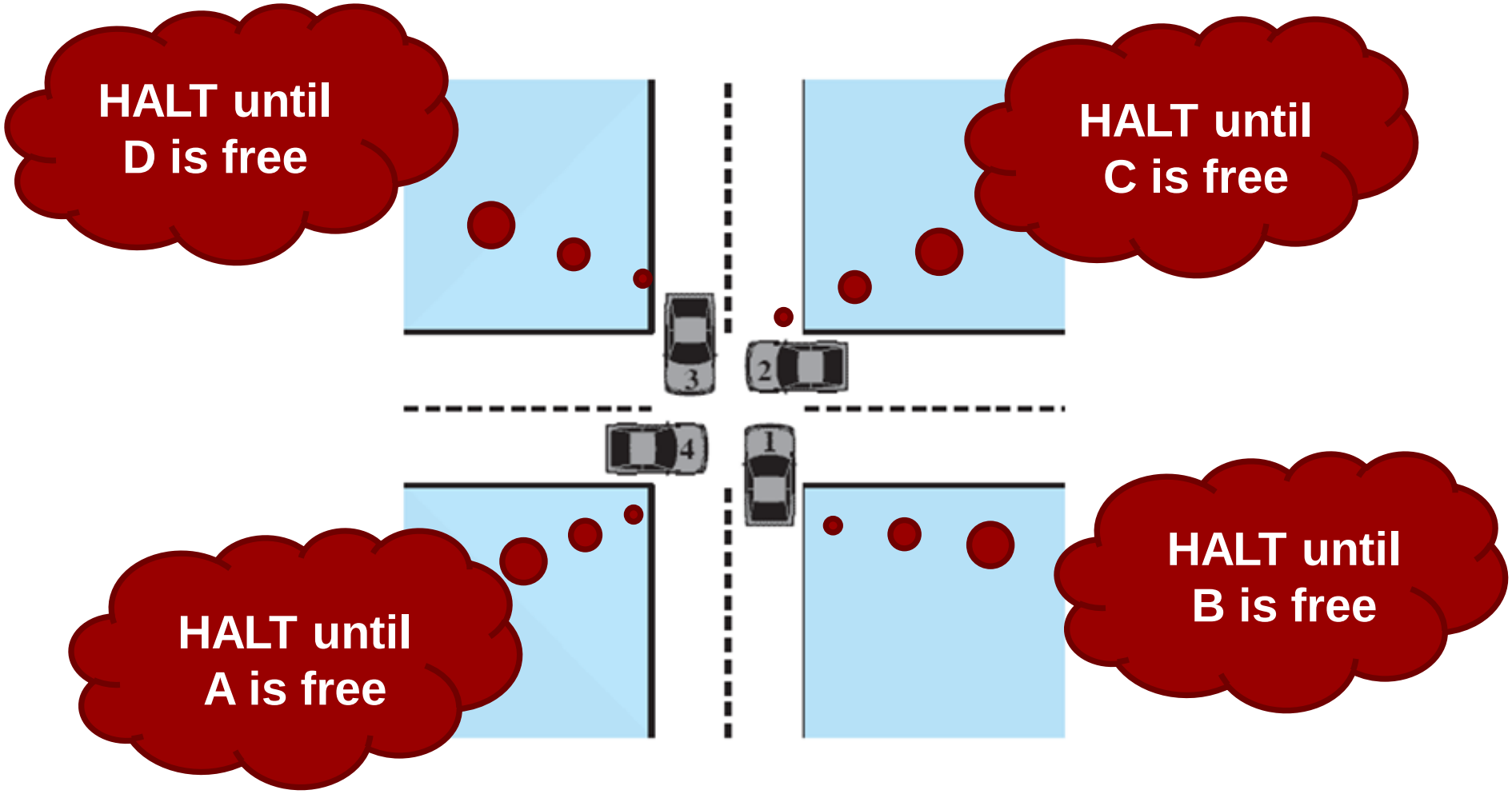
- A set of processes is **deadlocked** when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set.

No efficient solution in general.

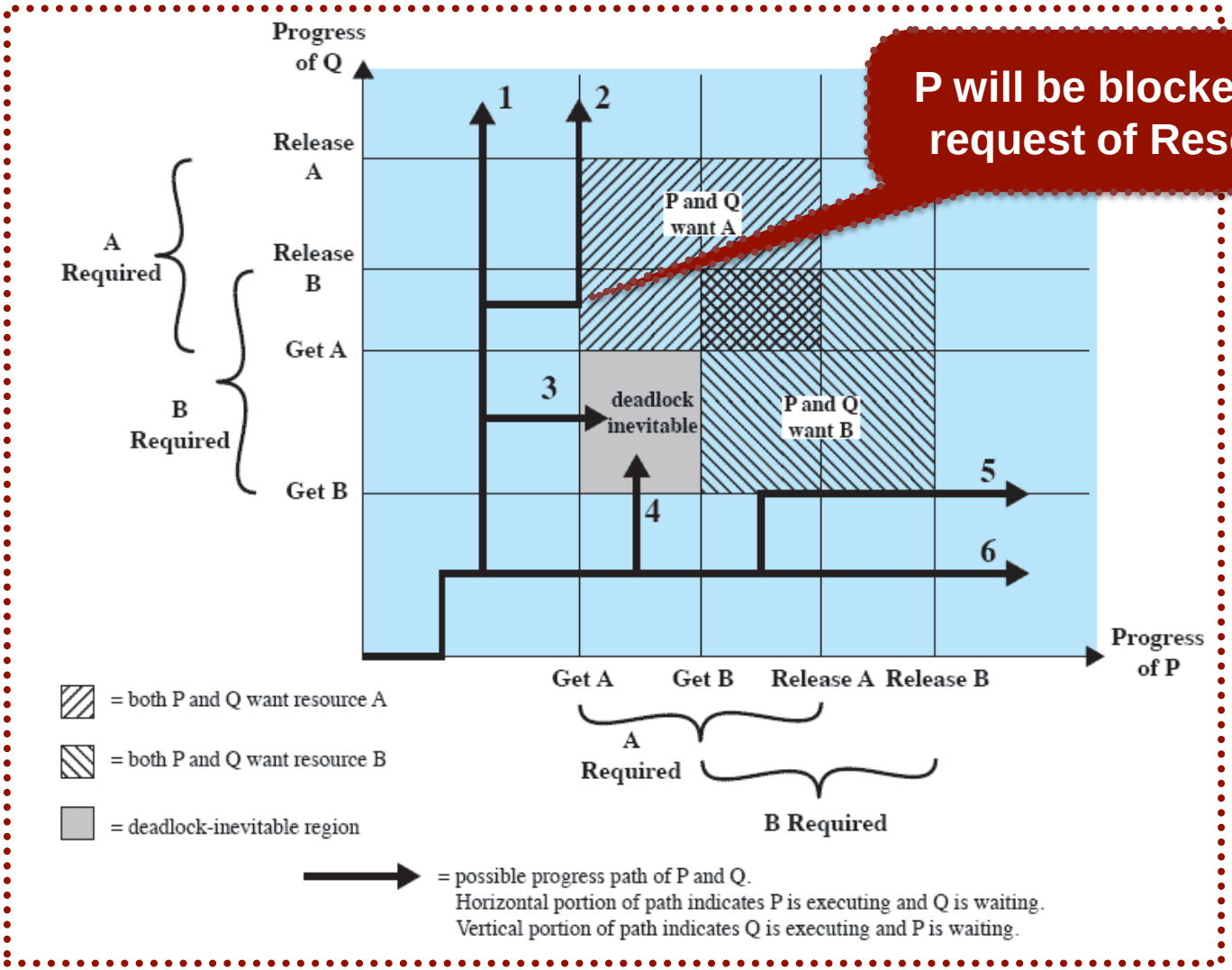
Potential Deadlock



Actual Deadlock

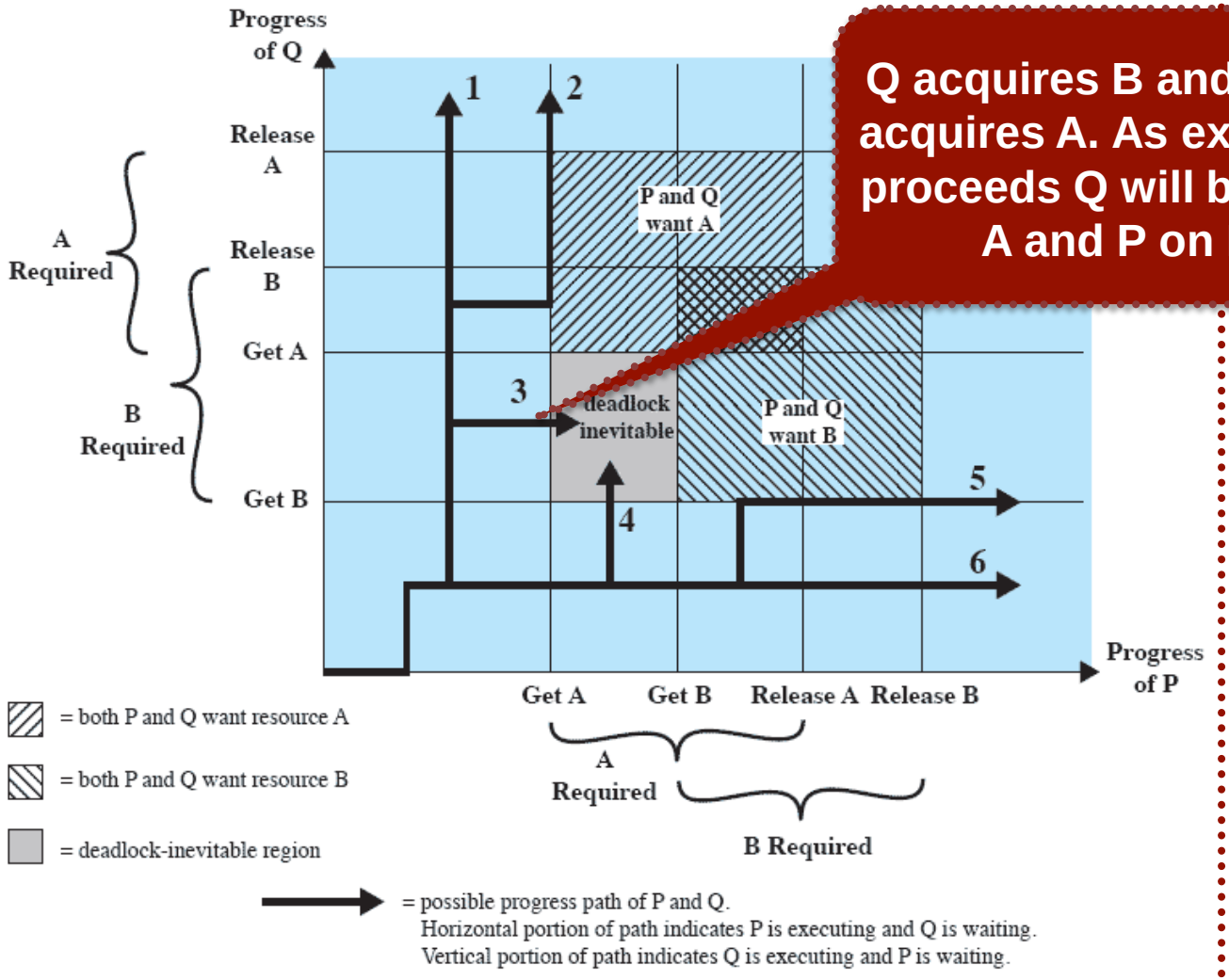


Deadlock: Joint Program Diagram



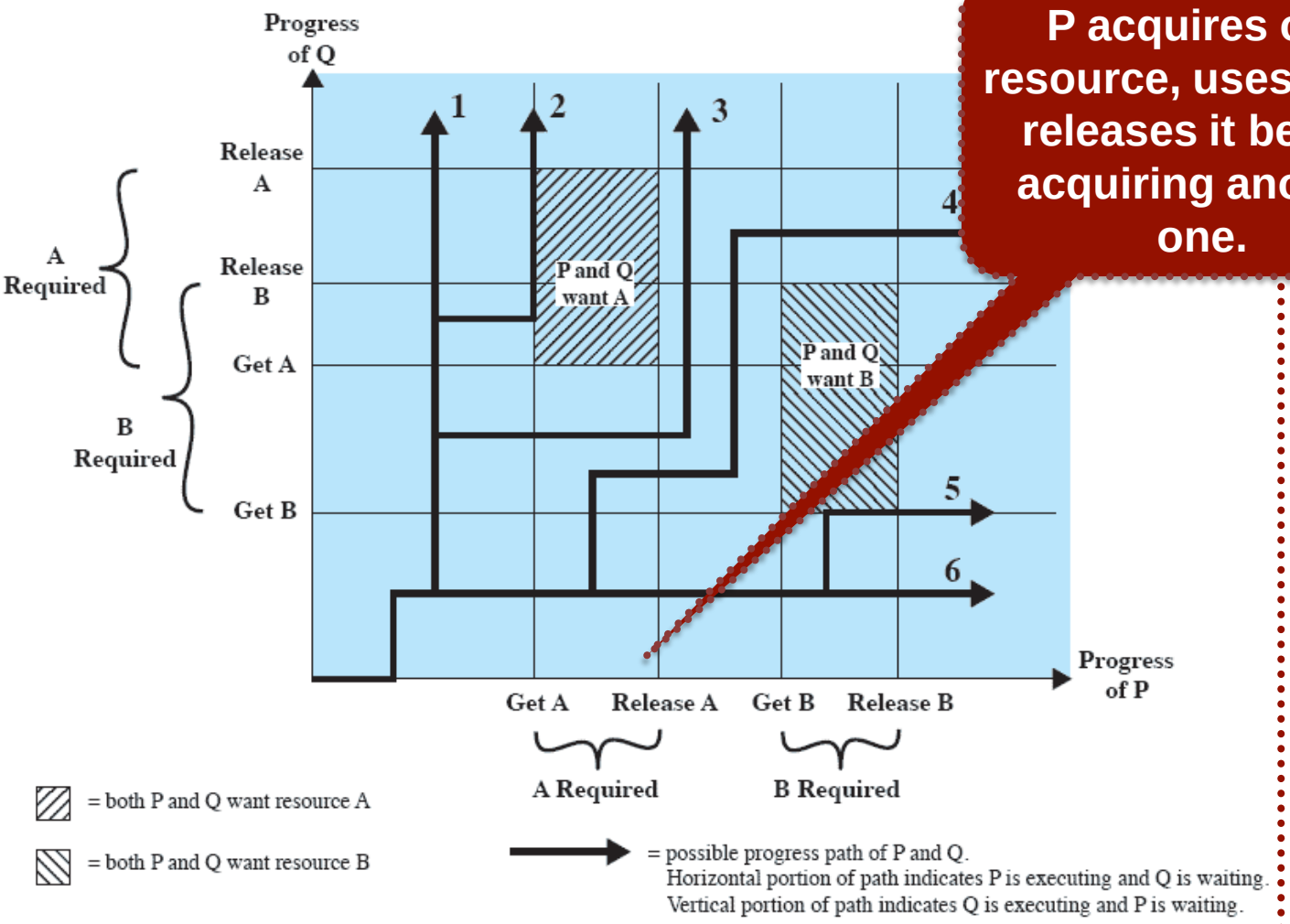
P will be blocked on the request of Resource A

Deadlock: Joint Program Diagram



Q acquires B and then P acquires A. As execution proceeds Q will block on A and P on B.

No Deadlock



P acquires one resource, uses it and releases it before acquiring another one.

MONASH University

What resources do processes compete for?

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

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 =>…

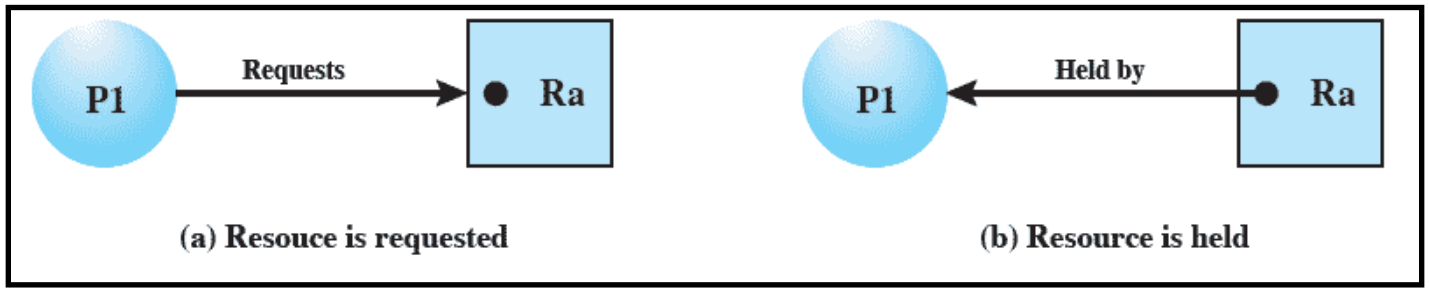
Consumable Resources: Two Processes Communicating

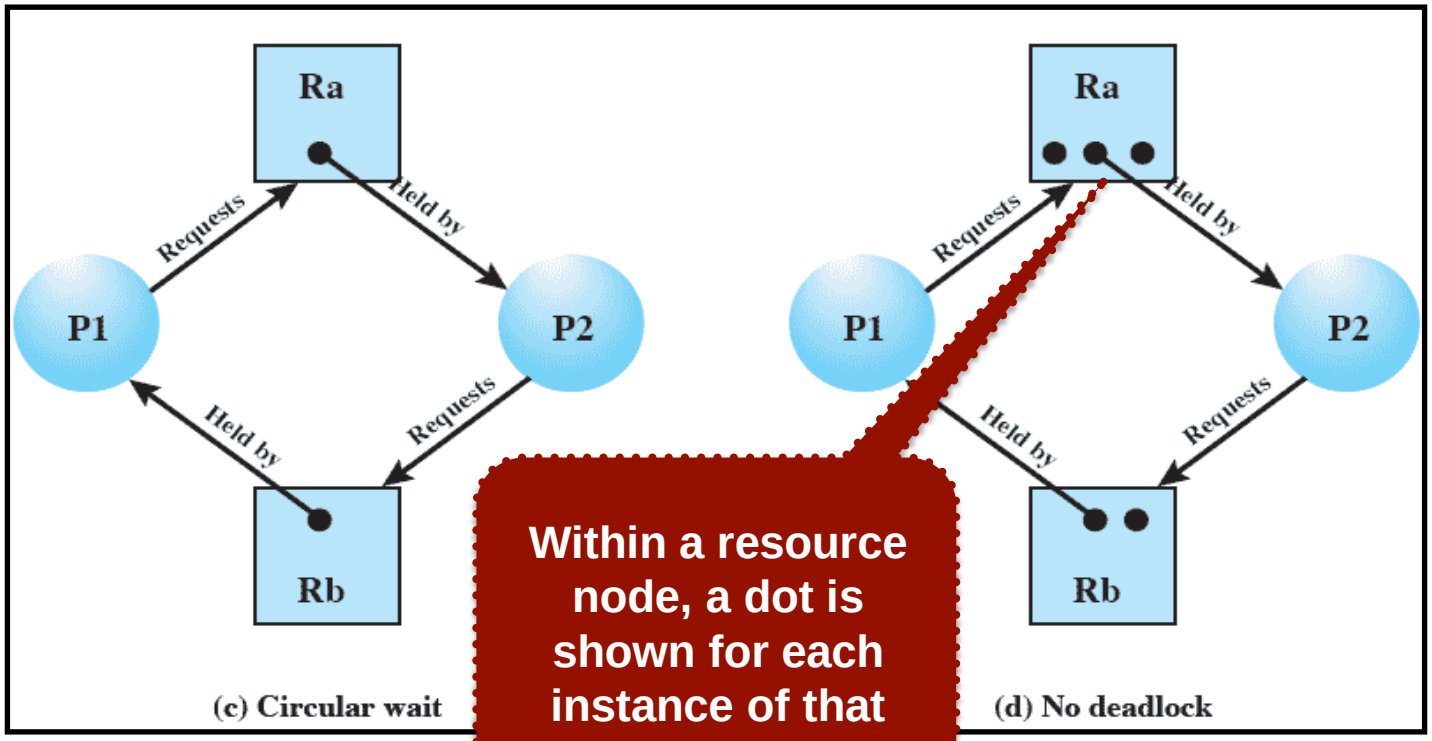
- Consider a pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process:

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).

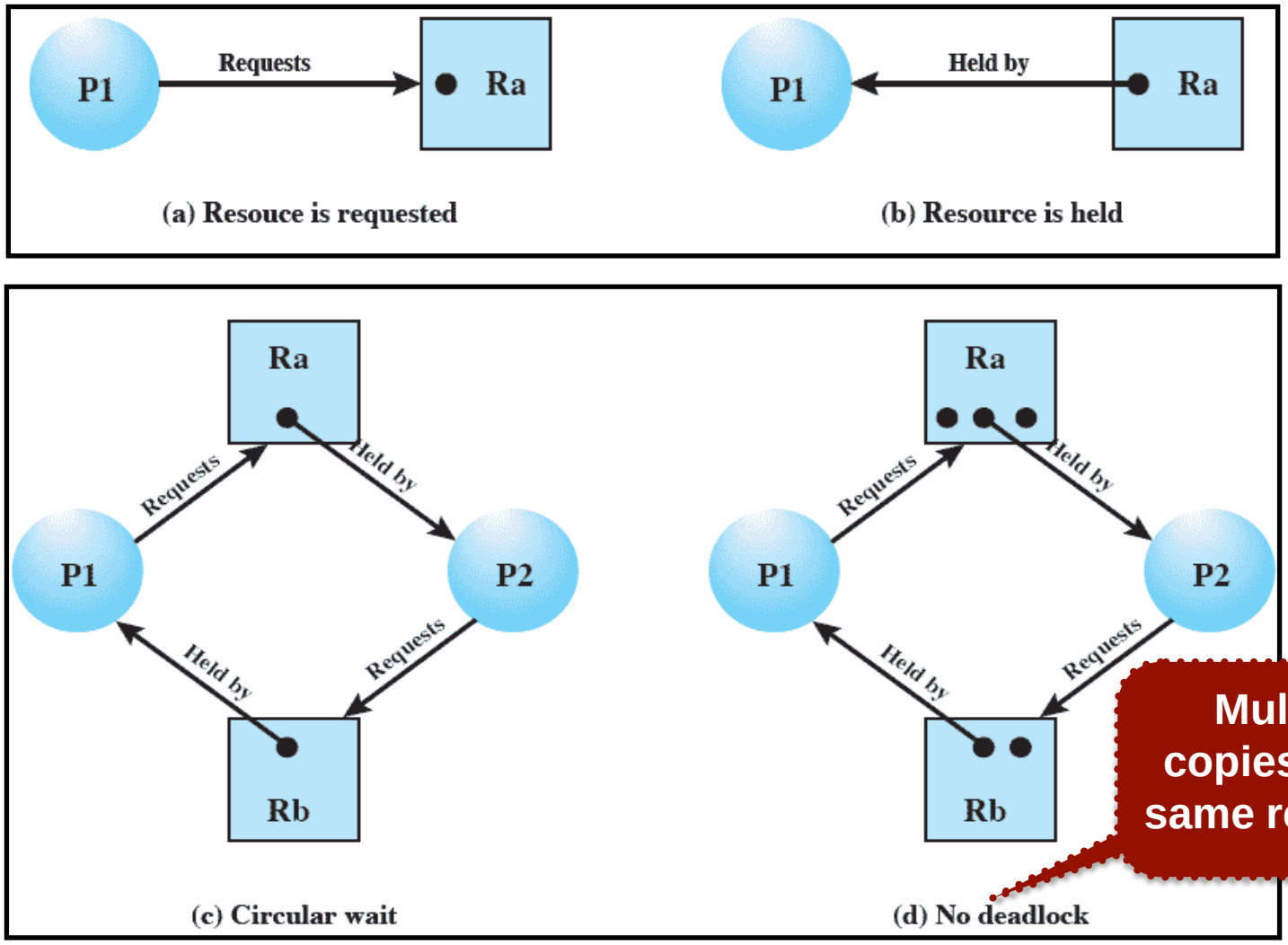
Resource Allocation Graph





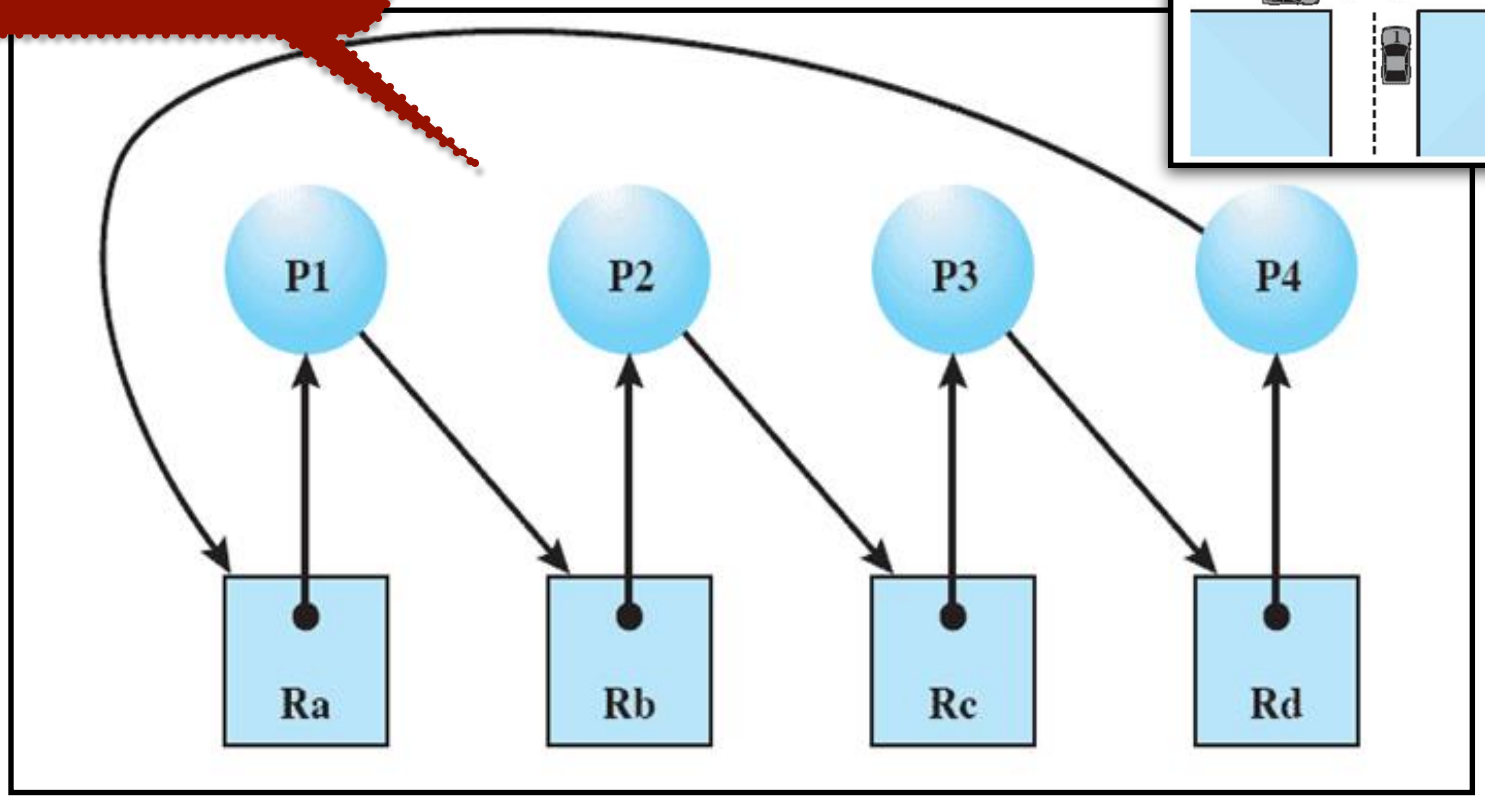
Within a resource node, a dot is shown for each instance of that resource

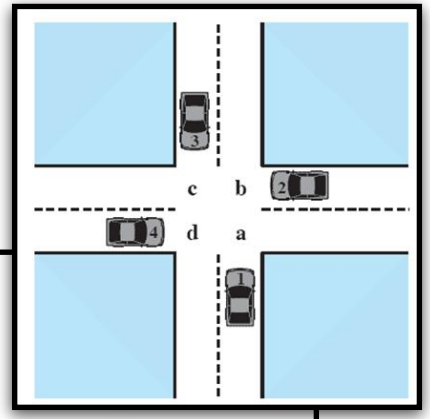
Resource Allocation Graph



Deadlock: Example

Circular chain of processes and resources that results in deadlock





MONASH University

What are conditions for a deadlock to occur?

Deadlock: The Four Conditions

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

These three can lead to the possibility of a deadlock while with the fourth one indicates the existence of a deadlock

Deadlock: The Four Conditions

Direct condition for deadlock exists

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

First three conditions are indirect (necessary) for a possibility of deadlock

MONASH
University

What are three common approaches to dealing with deadlock?

Dealing 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.

Three general strategies

Deadlock: Prevention Strategy

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

- Two main methods:
  - Indirect — prevent the occurrence of one of the three necessary conditions
  - Direct — prevent the occurrence of a circular wait

Deadlock: Condition Prevention

Should not be disallowed

Might not know all the resources needed in advance

Mutual Exclusion

Hold and Wait

if access to a resource requires mutual exclusion then it must be **supported by the OS**

require 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

Only practical when states of resources can be easily saved and restored later

❑ 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

Slow down processes and denying resource access

❑ Circular Wait

- Define a linear ordering of resource types

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

Deadlock: Avoidance Approaches

Deadlock Avoidance

Process Initiation Denial

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

Resource Allocation Denial

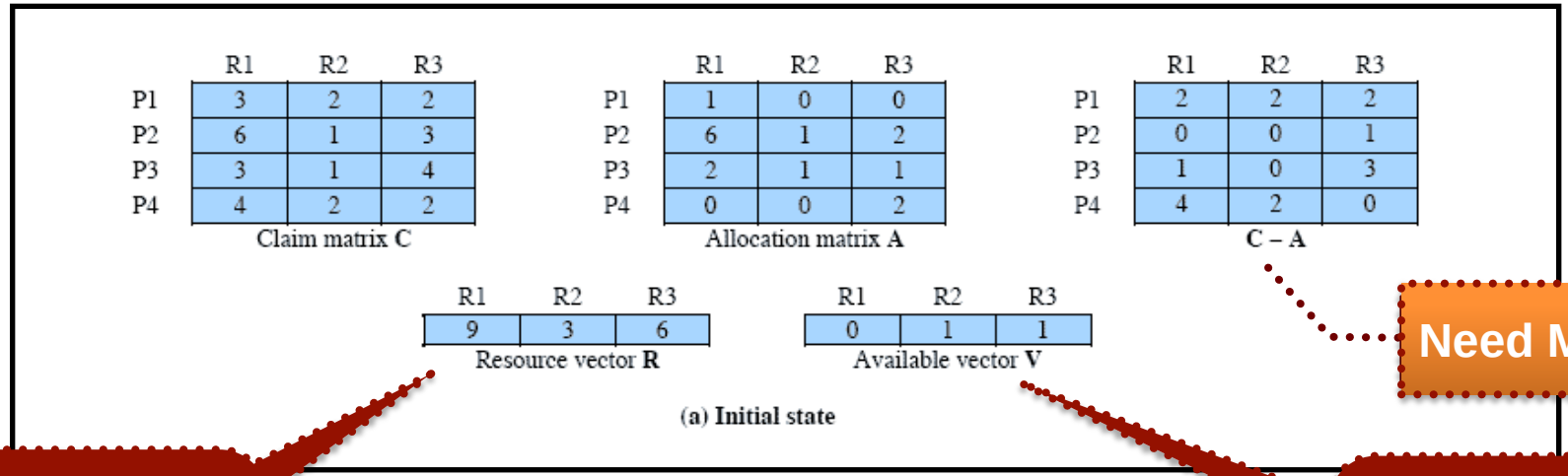
- do not grant an incremental resource request to a process if this allocation might lead to deadlock

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

Avoidance Strategy: Determination of a Safe State

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



Need Matrix

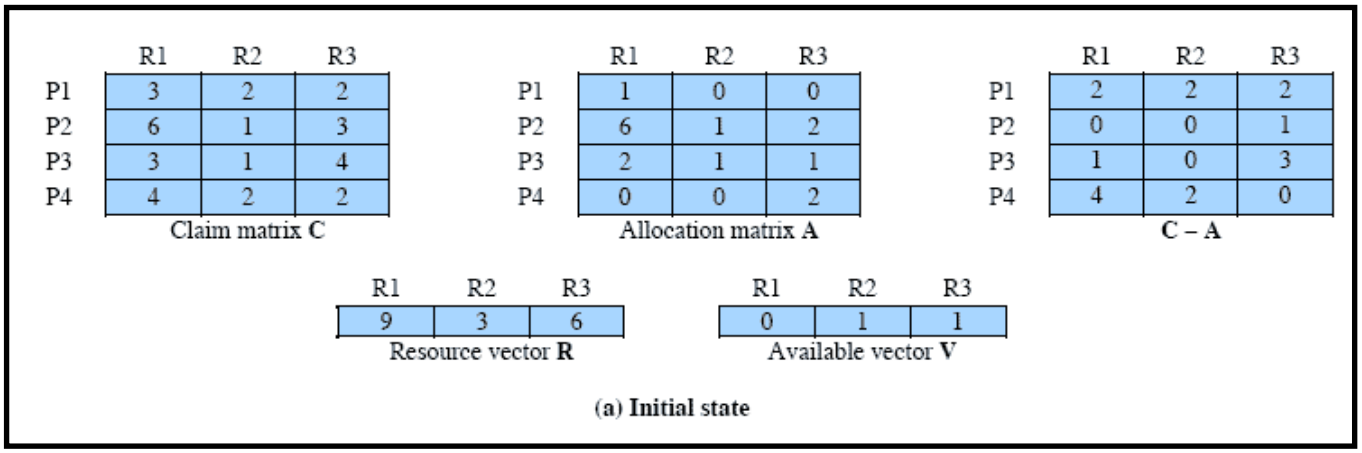
Amount of existing resources

Resources available after allocation

Process P2 makes the remaining resource request of R3

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

Avoidance Strategy: 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

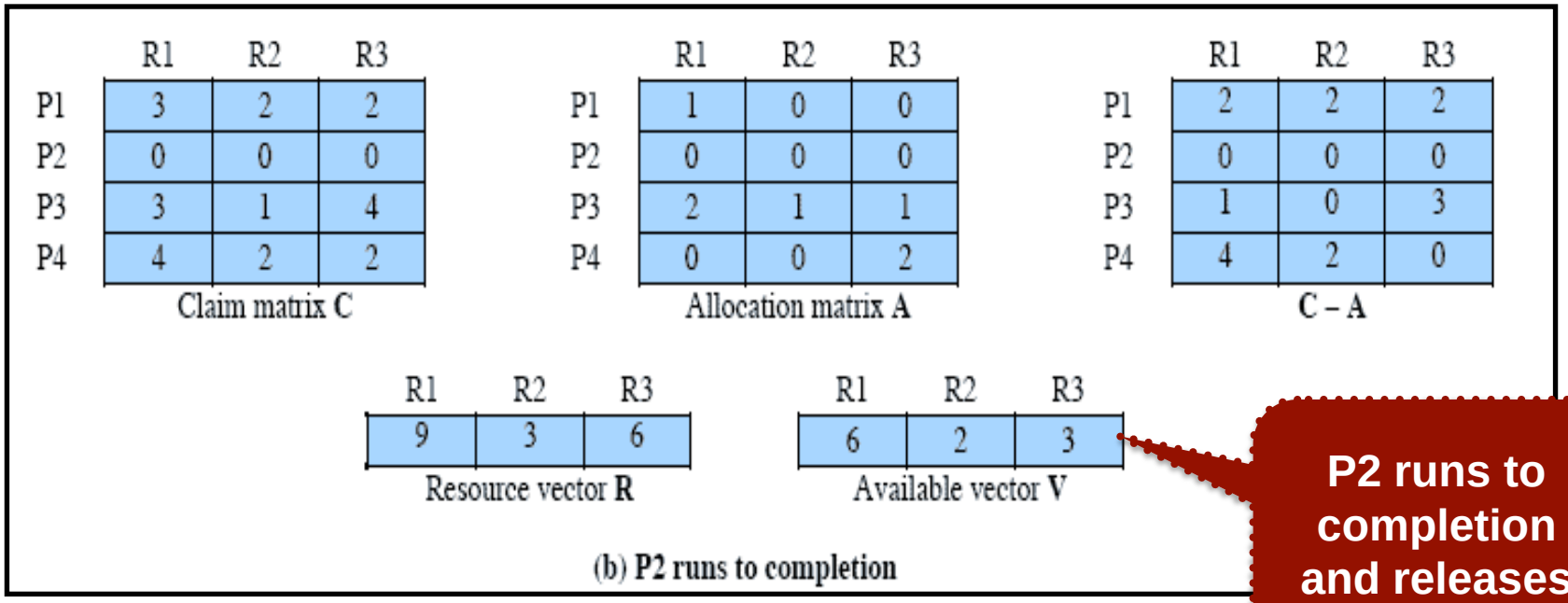
R1	R2	R3
9	3	6

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

Avoidance Strategy: 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

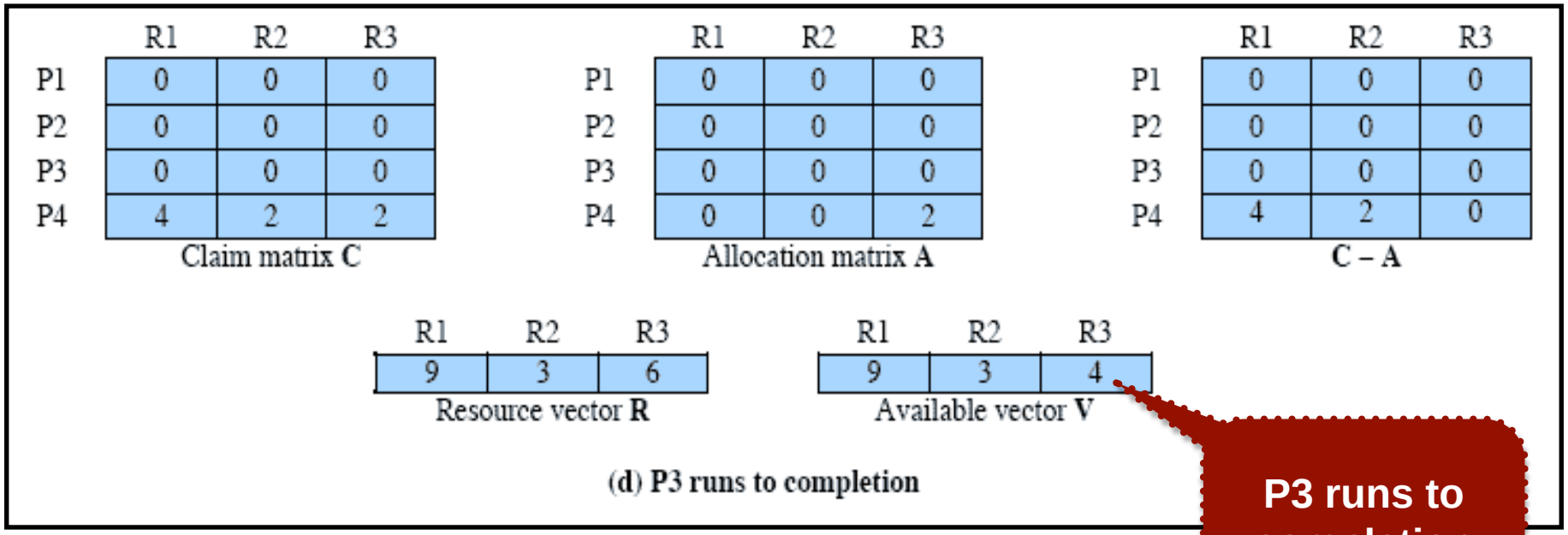
R1	R2	R3
7	2	3

Available vector V

(c) P1 runs to completion

P1 makes the request for the remaining resources

Avoidance Strategy: 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

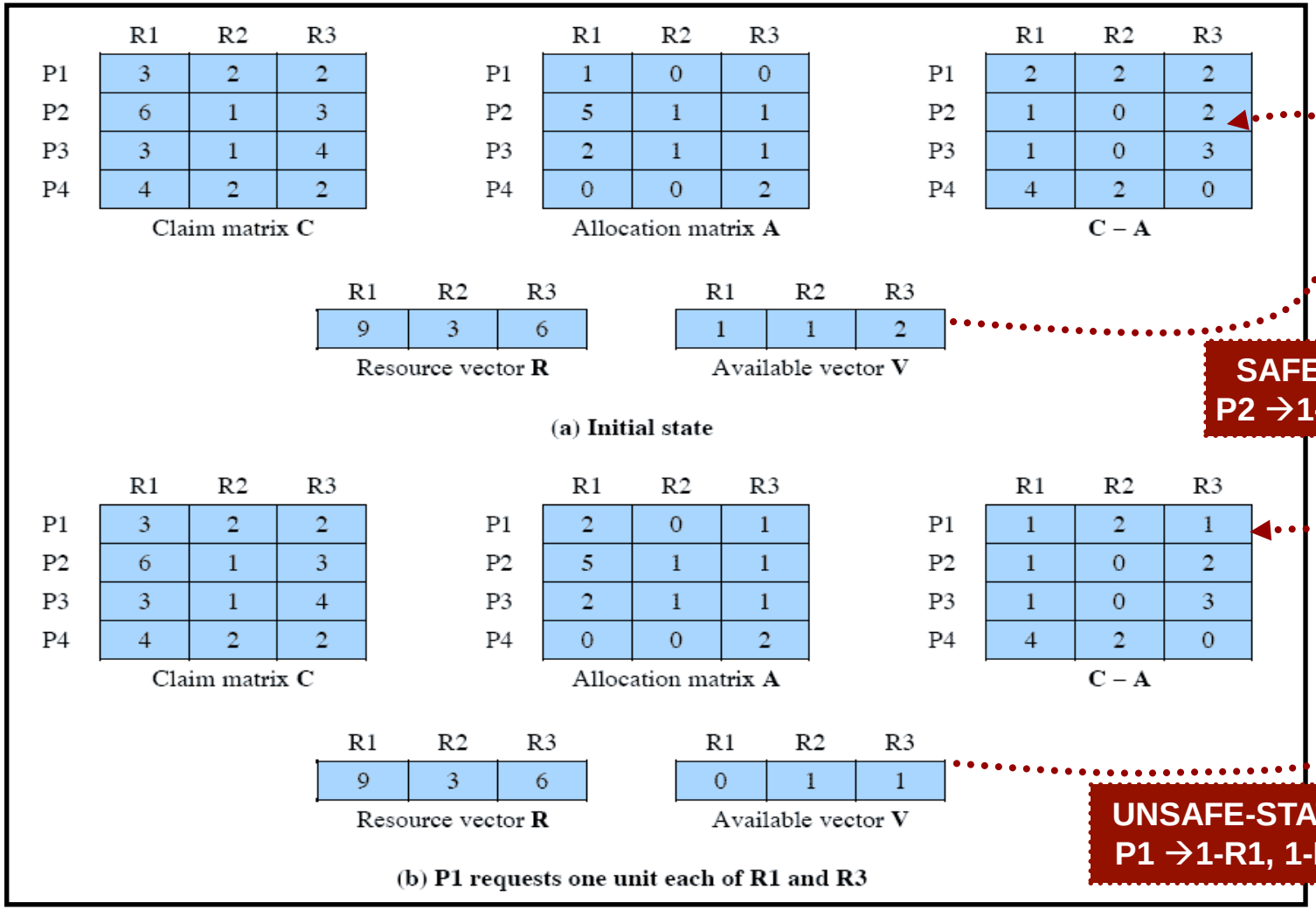
Available vector **V**

(d) P3 runs to completion

P3 runs to completion

The state defined originally is a safe state

Avoidance Strategy: Determination of an Unsafe State



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: Prevention Strategy vs Detection Strategy

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

Deadlock: 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.

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

- **Disadvantage**:
  - frequent checks consume considerable processor time

Deadlock: Detection Algorithms

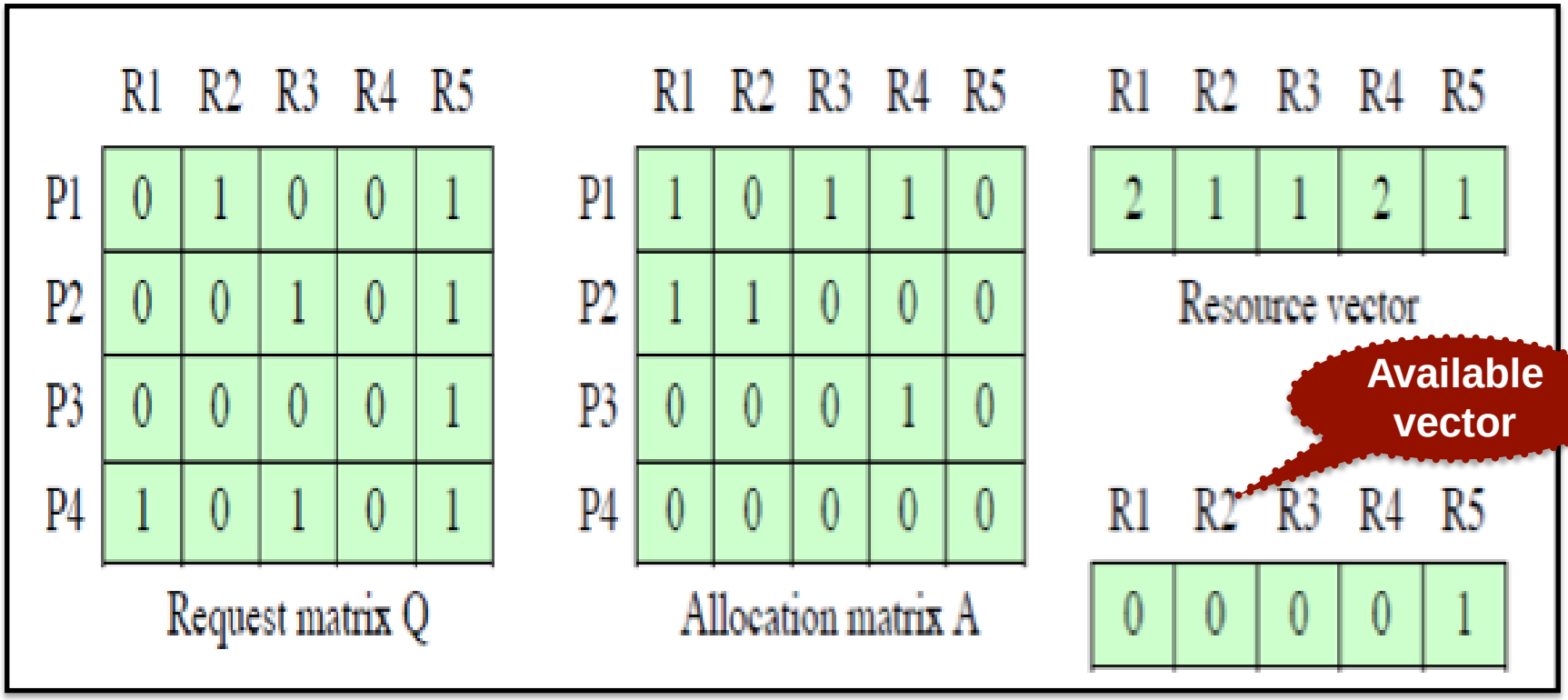
1. Mark each process that has a row in the allocation matrix of all zeros

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**. That is, $Q_{ik} <= W_k$ for **1 <= k <= m**. 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**. That is, set $W_k = W_k + A_{ik}$ for **1 <= k <= m**. Return to step 3.

Deadlock Detection: Example



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

Deadlock Detection: Recovery Strategies

- 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: Prevention, Avoidance, Detection

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

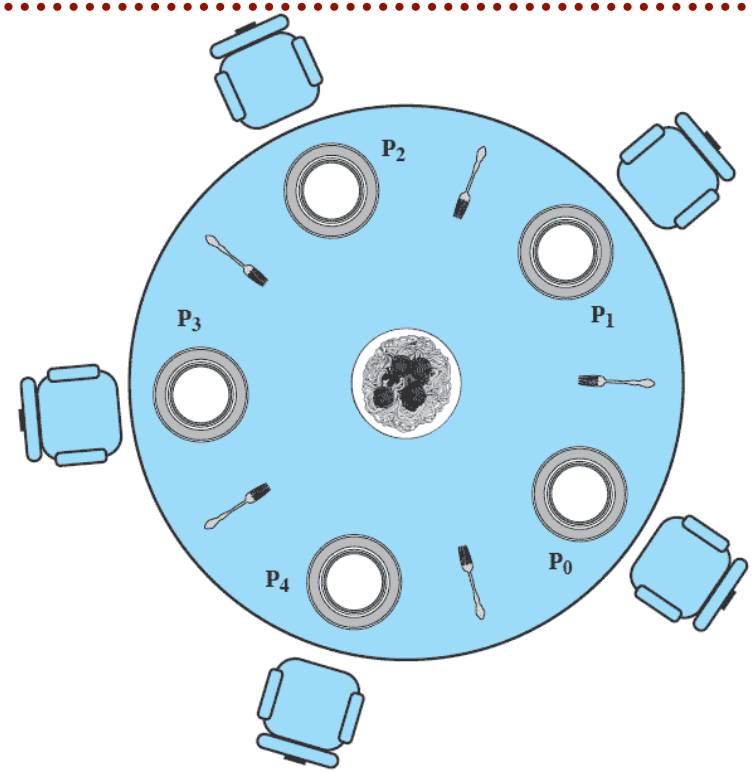
MONASH University

The Dining Philosophers problem

*Reading from Stallings, Chapter 6: 6.6

The Dining Philosophers Problem

- Each philosopher needs 2 forks to eat
- No two philosophers can use the same fork at the same time — satisfy mutual exclusion
- No philosopher must starve to death — avoid deadlock and starvation



First Solution: Five Seated Philosophers

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

Deadlock occurs if all philosophers want to eat at the same time

Second Solution: Four Seated Philosophers

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

Maximum four seated philosophers are allowed

Summary of Lecture 8

- Semaphores are use for signaling among processes and can be readily used to enforce a mutual exclusion discipline.
- Deadlock is a situation of blocking a set of processes that either compete for system resources or communicate with each other.

Reading from Stallings, Chapter 5: 5.1, 5.3
Chapter 6: 6.1, 6.2, 6.3, 6.4, 6.6