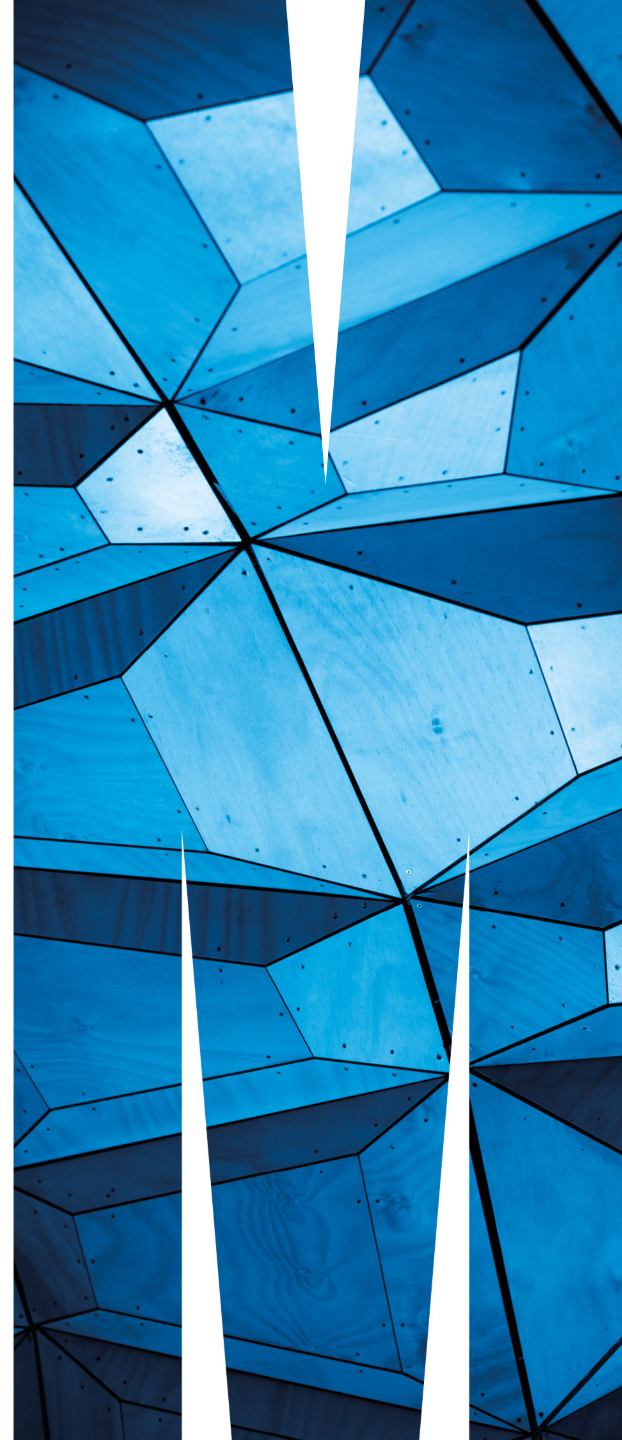
MONASH University

MONASH
INFORMATION
TECHNOLOGY



FIT2100 Semester 2 2017

Lecture 5: Concurrency (Part 2)

(Reading: Stallings, Chapter 5 and Chapter 6)

Jojo Wong

GROUP
OF EIGHT
AUSTRALIA

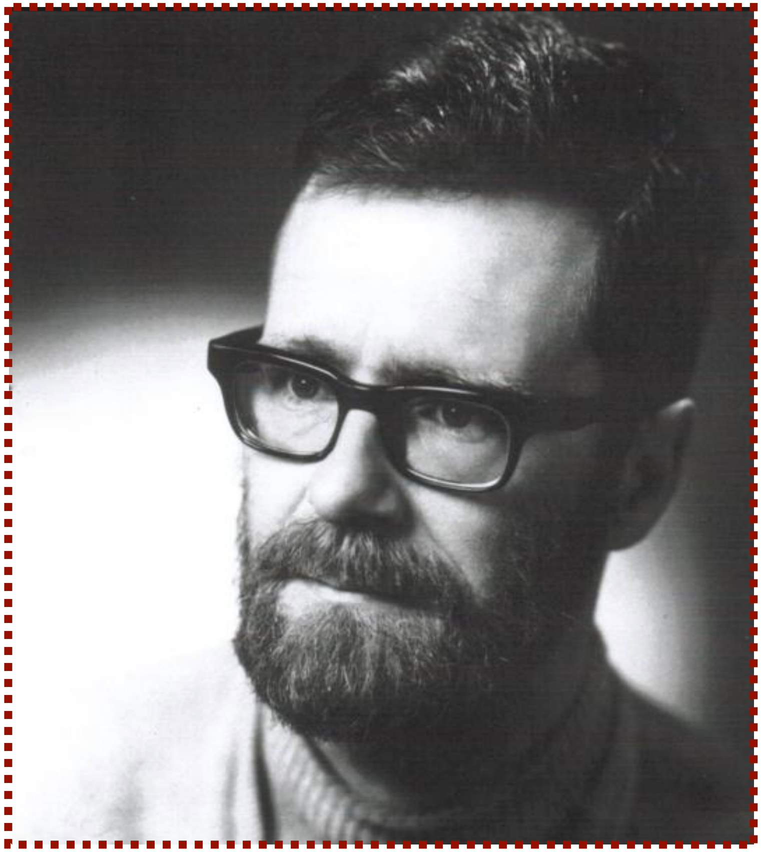
Lecture 5: Learning Outcomes

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

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.

data inconsistency

- 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.

deadlock and starvation

What are the common concurrency mechanisms?

Concurrency: Control Mechanisms

Semaphore	An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a **counting semaphore** or a **general semaphore.**
Binary Semaphore	A semaphore that takes on only the values 0 and 1.
Mutex	Similar to a binary semaphore. 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).
Condition Variable	A data type that is used to block a process or thread until a particular condition is true.
Monitor	A programming language construct that encapsulates variables, access procedures, and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are *critical sections*. A monitor may have a queue of processes that are waiting to access it.
Event Flags	A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).
Mailboxes/Messages	A means for two processes to exchange information and that may be used for synchronization.
Spinlocks	Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.

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 process
- 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 takes integer values

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.

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*

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

Mutex: Mutual Exclusion Lock

- A related concept to binary semaphores

Unix/Linux Implementation

- Mutex is a programming flag:
  - set to 0 when it is locked
  - set to 1 when it is unlocked

- Different from binary semaphores:
  - The process that locks the mutex must be the one to unlock it

Semaphores: Strong vs. Weak

- A queue is used to hold processes waiting on a semaphore.

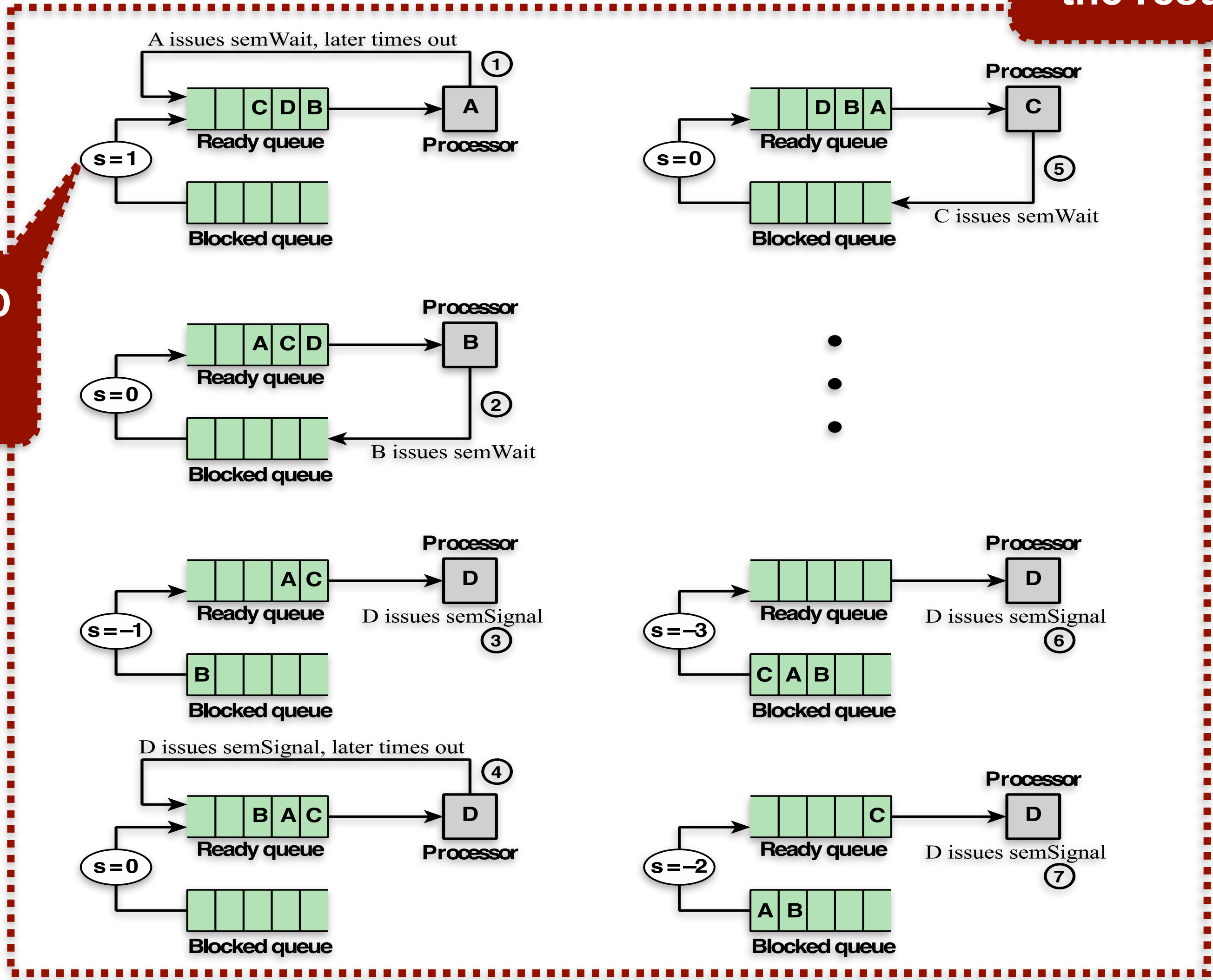
- Strong semaphores:
  - The process that has been blocked the longest is released from the queue first (FIFO).

- Weak semaphores:
  - The order in which processed are removed from the queue is not specified.

Strong Semaphore Mechanism: Example

A, B and C rely on the result from D

One of the D results is available

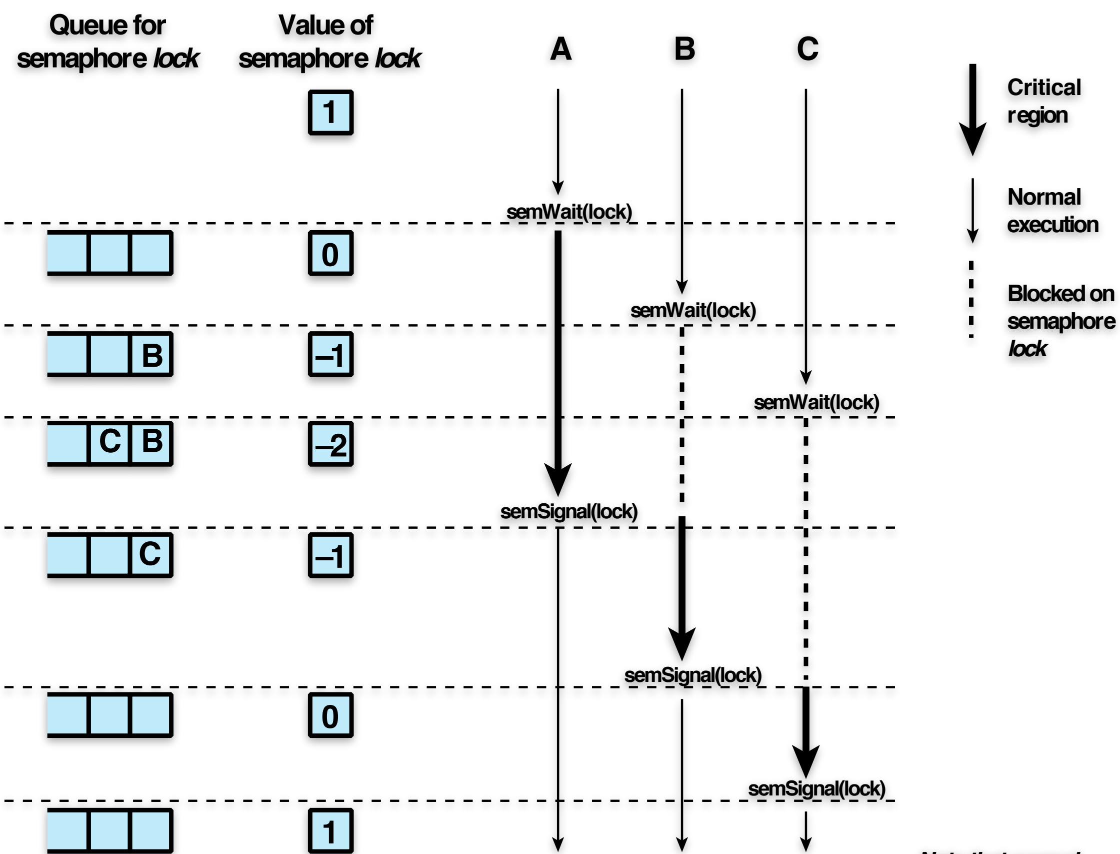


Mutual Exclusion: Using Semaphores

```
/* 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



Note that normal execution can proceed in parallel but that critical regions are serialized.

MONASH University

MONASH University

What is the producer/consumer problem?
(The classical concurrency problem)

The Producer/Consumer Problem

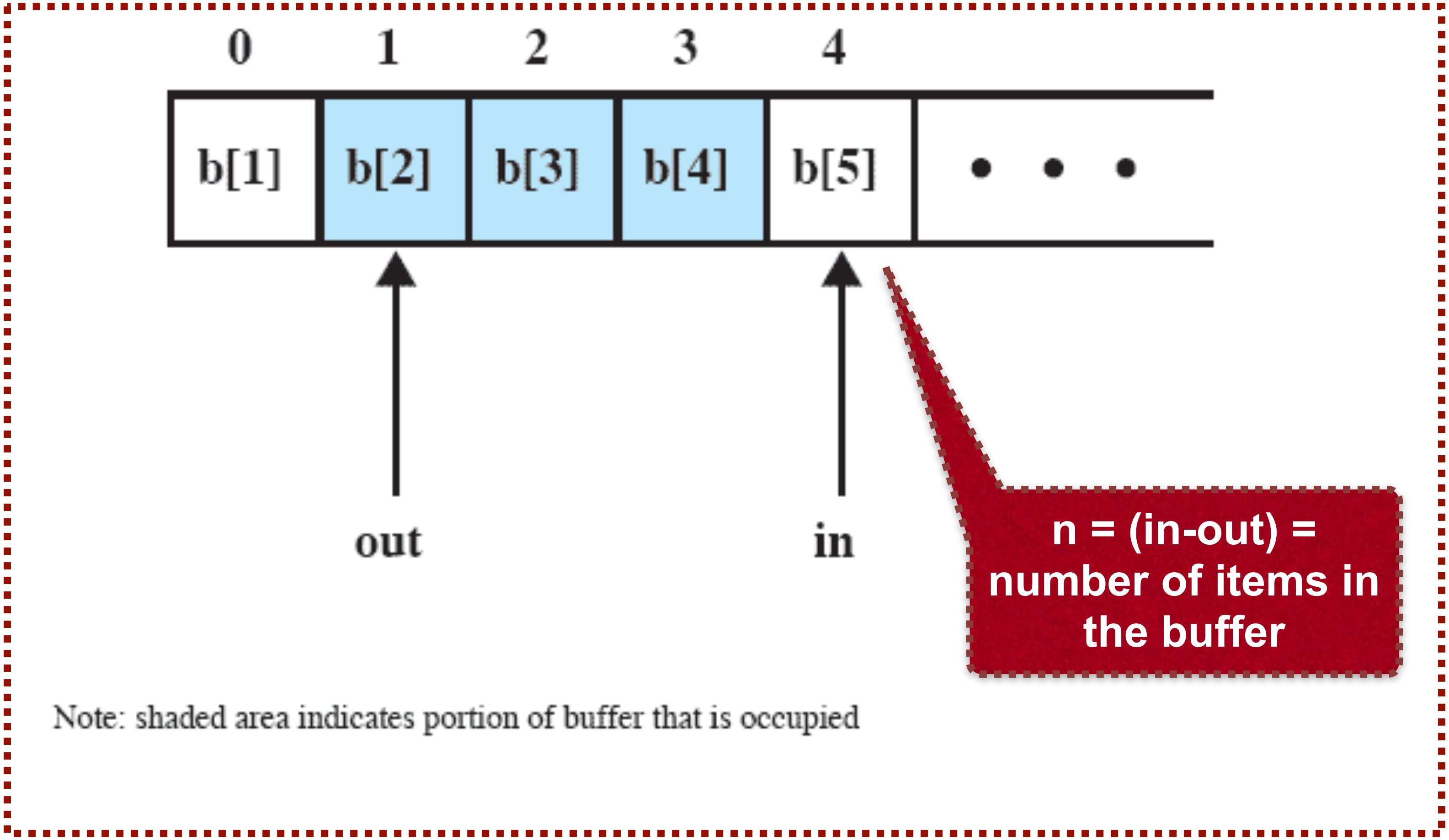
General Situation

- One or more producers are generating data and placing these in a buffer
- A single consumer is taking items out of the buffer one at a time
- Only one producer or a consumer may access the buffer at any one time

The Problem

- Ensure that the producer cannot add data into a full buffer
- Consumer cannot remove data from an empty buffer

The Producer/Consumer Problem: Infinite Buffer



Solving with Binary Semaphores: Incorrect Solution

n = number of items in the buffer

There is, however, a flaw in this program. When the consumer has exhausted the buffer, it needs to reset the delay semaphore so that it will be forced to wait until the producer has placed more items in the buffer. This is the purpose of the statement: if n == 0 semWaitB (delay) .

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     semWaitB(delay);
     while (true) {
          semWaitB(s);
          take();
          n--;
          semSignalB(s);
          consume();
          if (n==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

Solving with Binary Semaphores: Incorrect Solution

A semaphore is initialised to 1.

semWait operation decrements the semaphore value.

The number of items in the buffer=n is

The semaphore delay is used to force the consumer to semWait if the buffer is empty.

semSignal operation increments the semaphore value.

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	**if** (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	**if** (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		**if** (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		**if** (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	–1	0
21		semiSignlaB(s)	1	–1	0

NOTE: White areas represent the critical section controlled by semaphore s.

MONASH University

Solving with Binary Semaphores: Incorrect Solution

In line 14, the consumer fails to execute the semWaitB operation. The consumer exhausts the buffer and set n to 0 (line 8), but the producer has incremented n before the consumer can test it in line 14. The result is a semSignalB not matched by a prior semWaitB . The value of –1 for n in line 20 means that the consumer has consumed an item from the buffer that does not exist. It would not do simply to move the conditional statement inside the critical section of the consumer because this could lead to deadlock (e.g., after line 8 of the Table).

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	**if** (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	**if** (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		**if** (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		**if** (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	–1	0
21		semiSignlaB(s)	1	–1	0

NOTE: White areas represent the critical section controlled by semaphore s.

Solving with Binary Semaphores: Incorrect Solution

n = number of items in the buffer

Deadlock (the producer will be waiting for the buffer — s to be released by the consumer, but consumer is blocked on delay)

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
      while (true) {
            produce();
            semWaitB(s);
            append();
            n++;
            if (n==1) semSignalB(delay);
            semSignalB(s);
      }
}
void consumer()
{
      semWaitB(delay);
      while (true) {
            semWaitB(s);
            take();
            n--;
            semSignalB(s);
            consume();
            if (n==0) semWaitB(delay);
      }
}
void main()
{
      n = 0;
      parbegin (producer, consumer);
}
```

Solving with Binary Semaphores: Correct Solution

A fix for the problem is to introduce an auxiliary variable that can be set in the consumer's critical section for use later on. A careful trace of the logic should convince you that deadlock can no longer occur.

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     int m; /* a local variable */
     semWaitB(delay);
     while (true)  {
          semWaitB(s);
          take();
          n--;
          m = n;
          semSignalB(s);
          consume();
          if (m==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

```
void consumer()
{
     semWaitB(delay);
     while (true) {
          semWaitB(s);
          take();
          n--;
          semSignalB(s);
          consume();
          if (n==0) semWaitB(delay);
     }
}
```

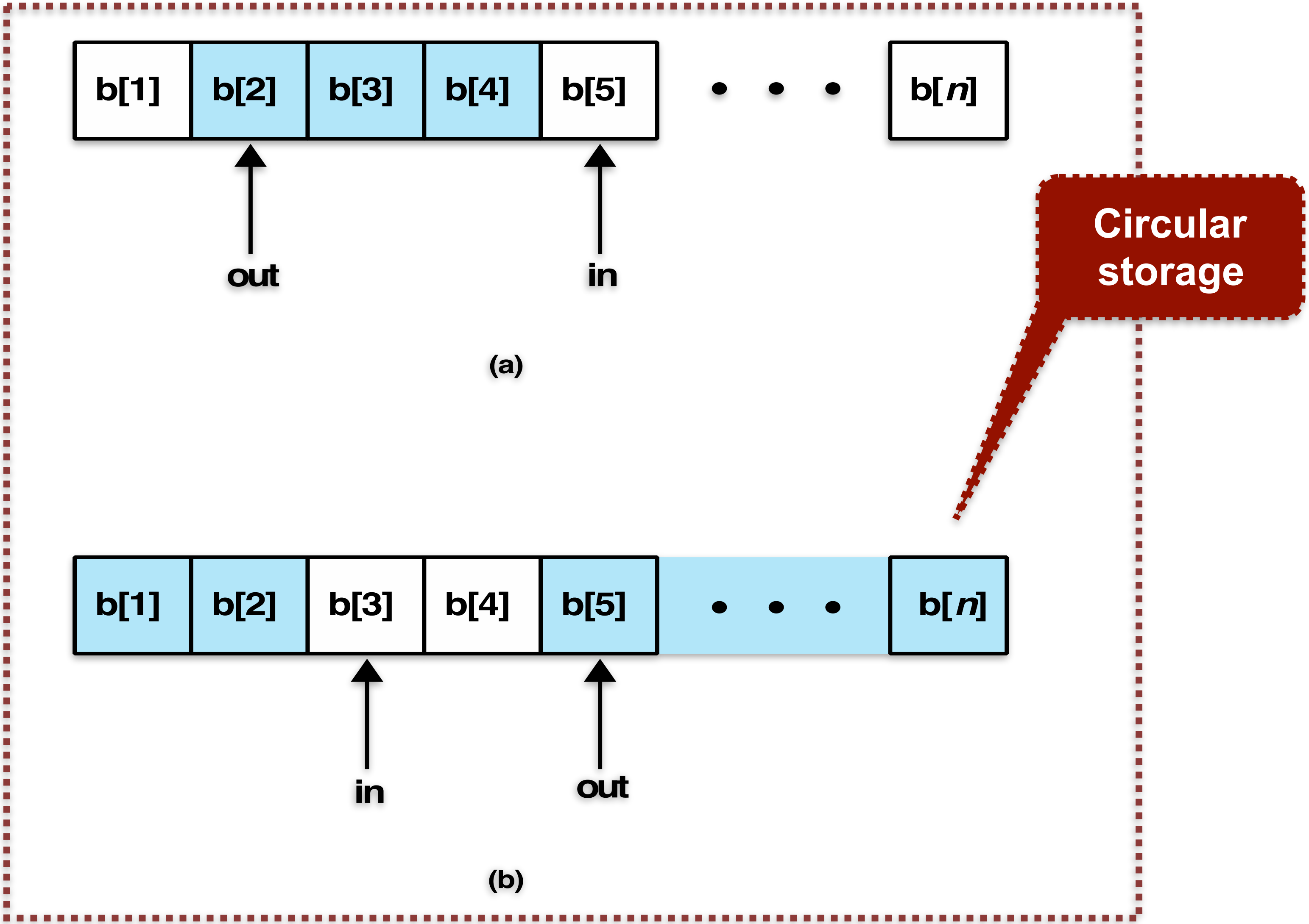
Solving with Counting Semaphores

```
/* program  producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
     while (true) {
          produce();
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Can we swap the order of semSignal(s) and

How about swapping the order of semWait(n) and semWait(s)?

The Producer/Consumer Problem: Finite Buffer



Solving with Counting Semaphores: Finite Buffer

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
     while (true) {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

producer must wait if there is no empty space

consumer must wait if there are no items in the buffer

What is a deadlock?
(Another concurrency problem)

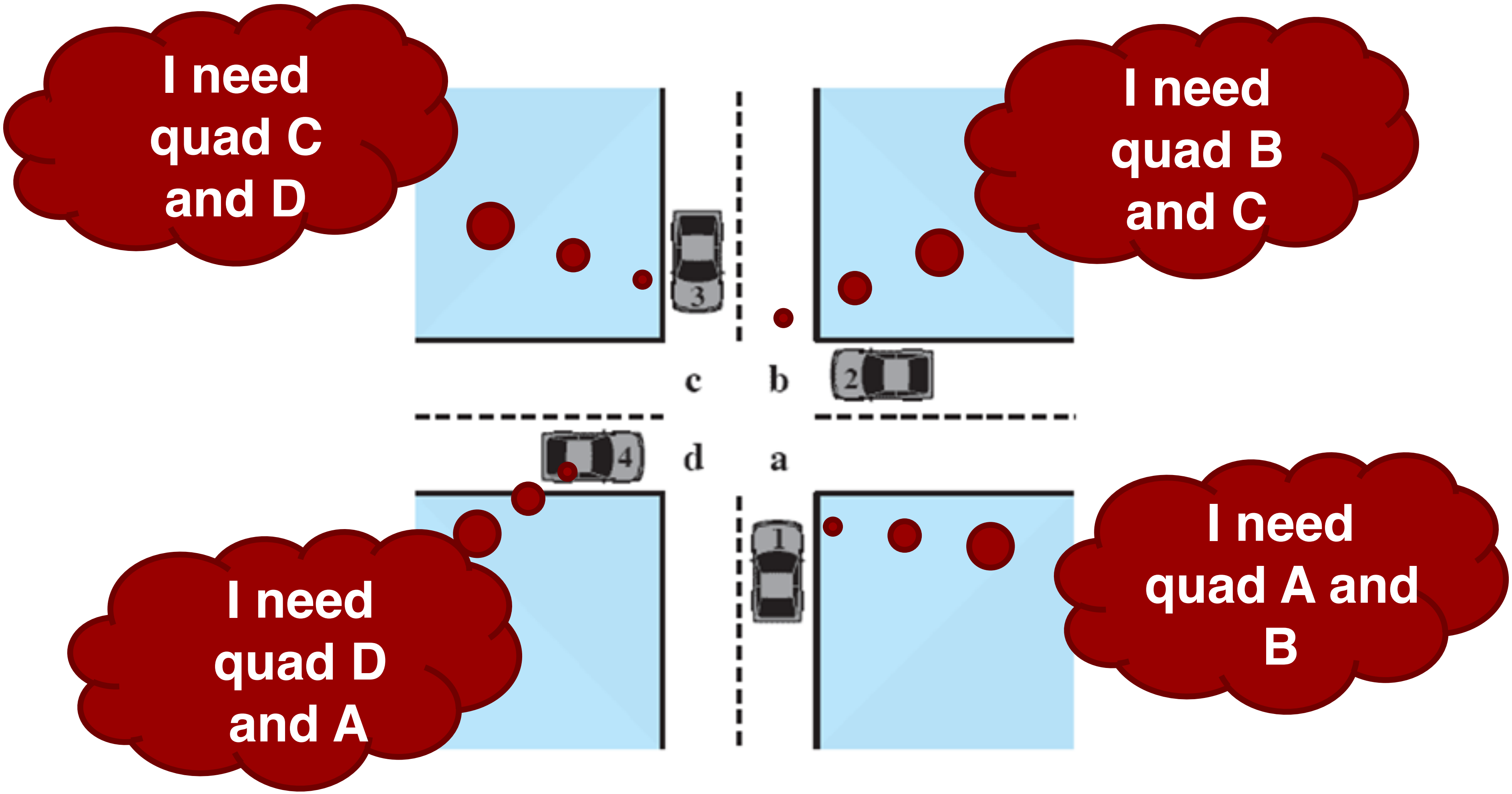
The Concept of Deadlock

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

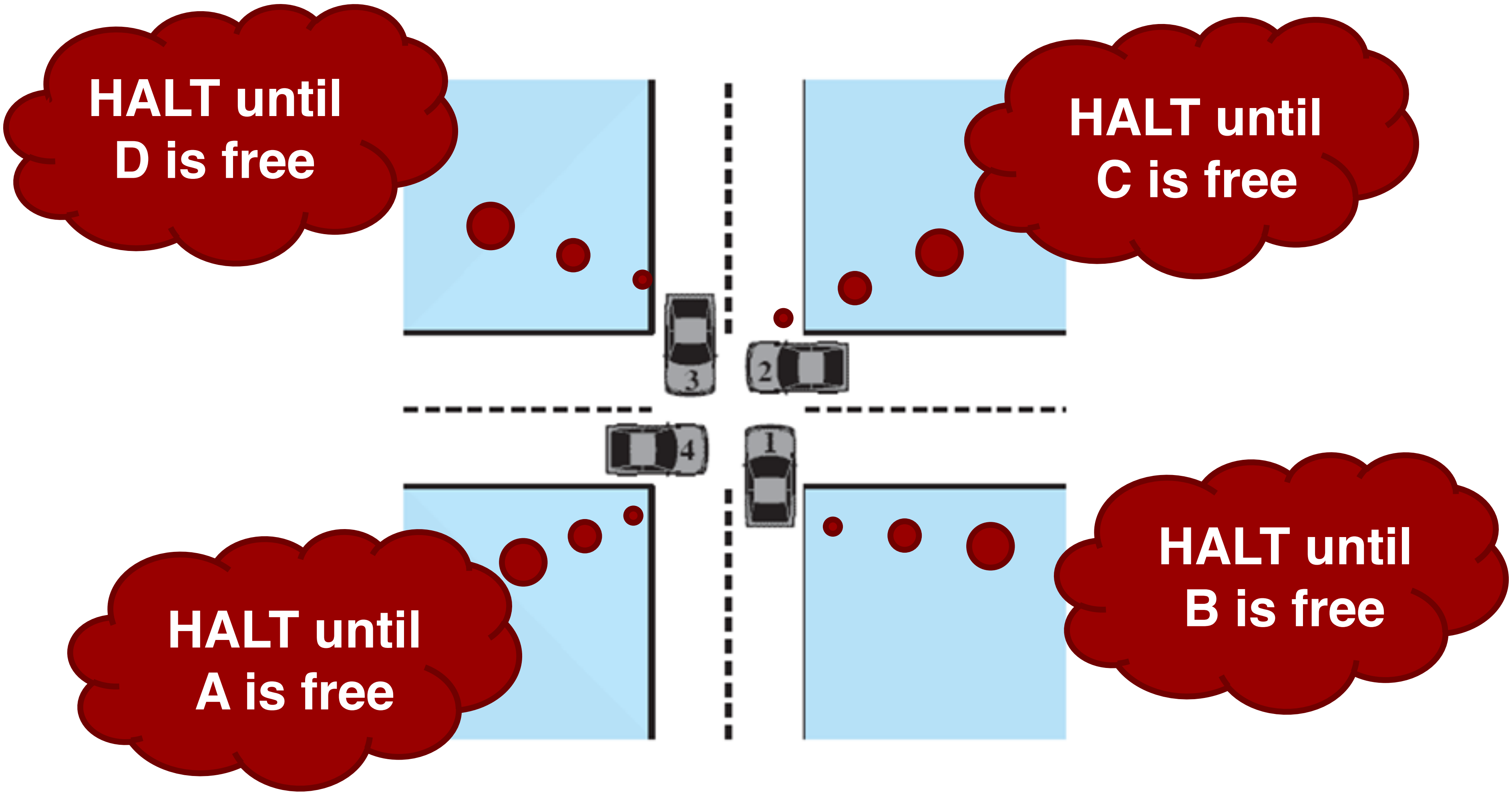
- A set of processes is **deadlocked** when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set.

No efficient solution in general.

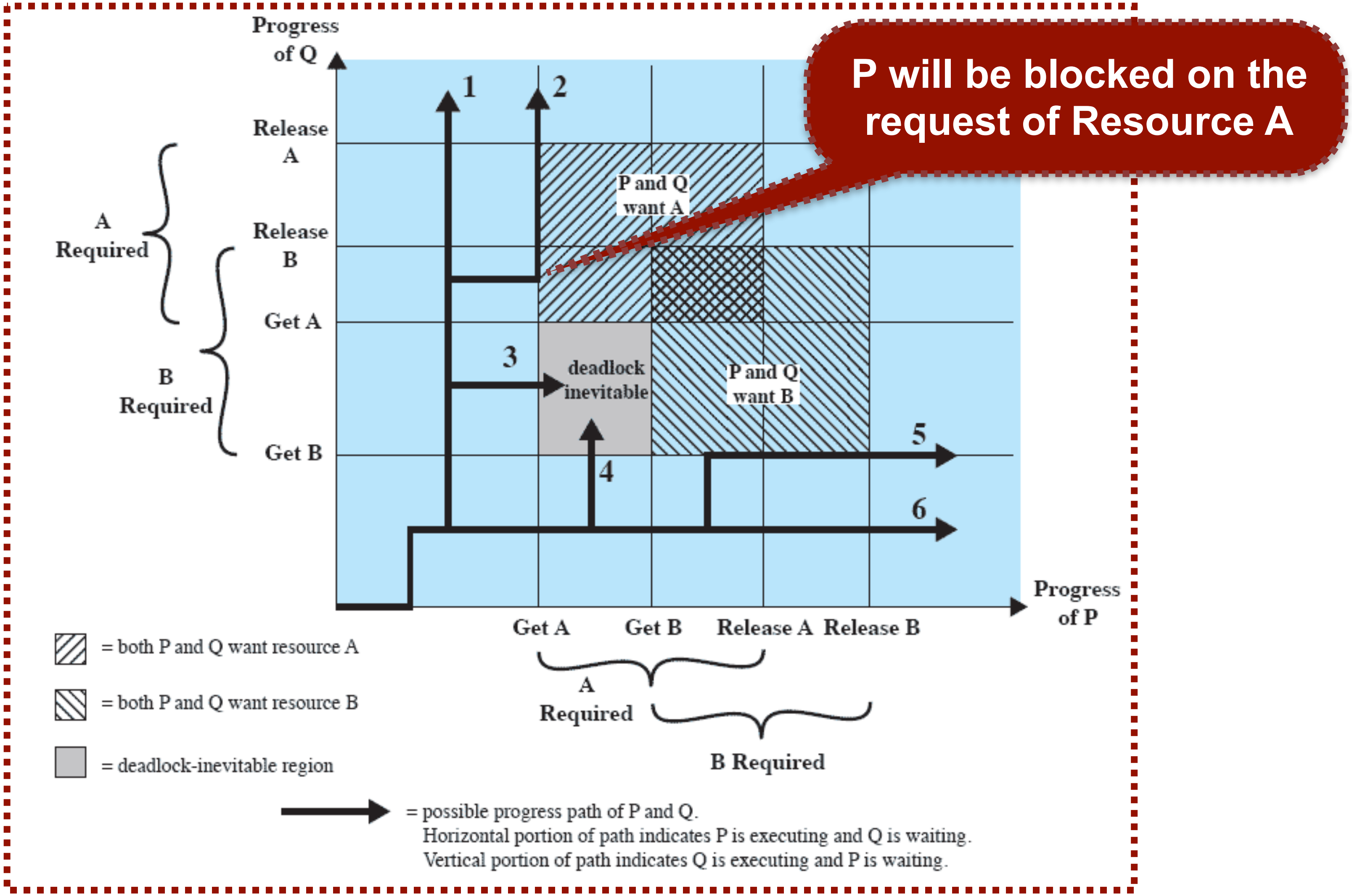
Potential Deadlock



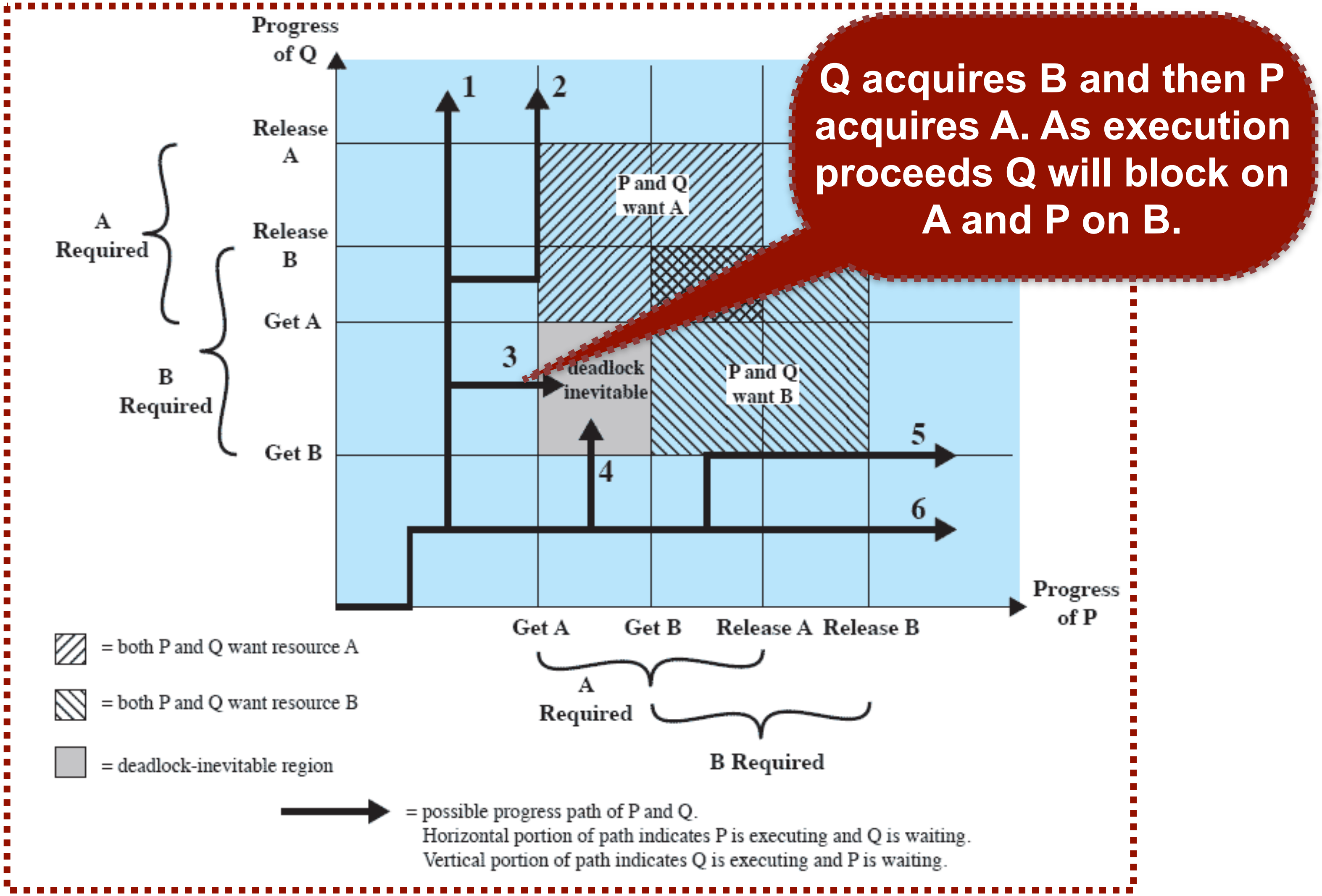
Actual Deadlock



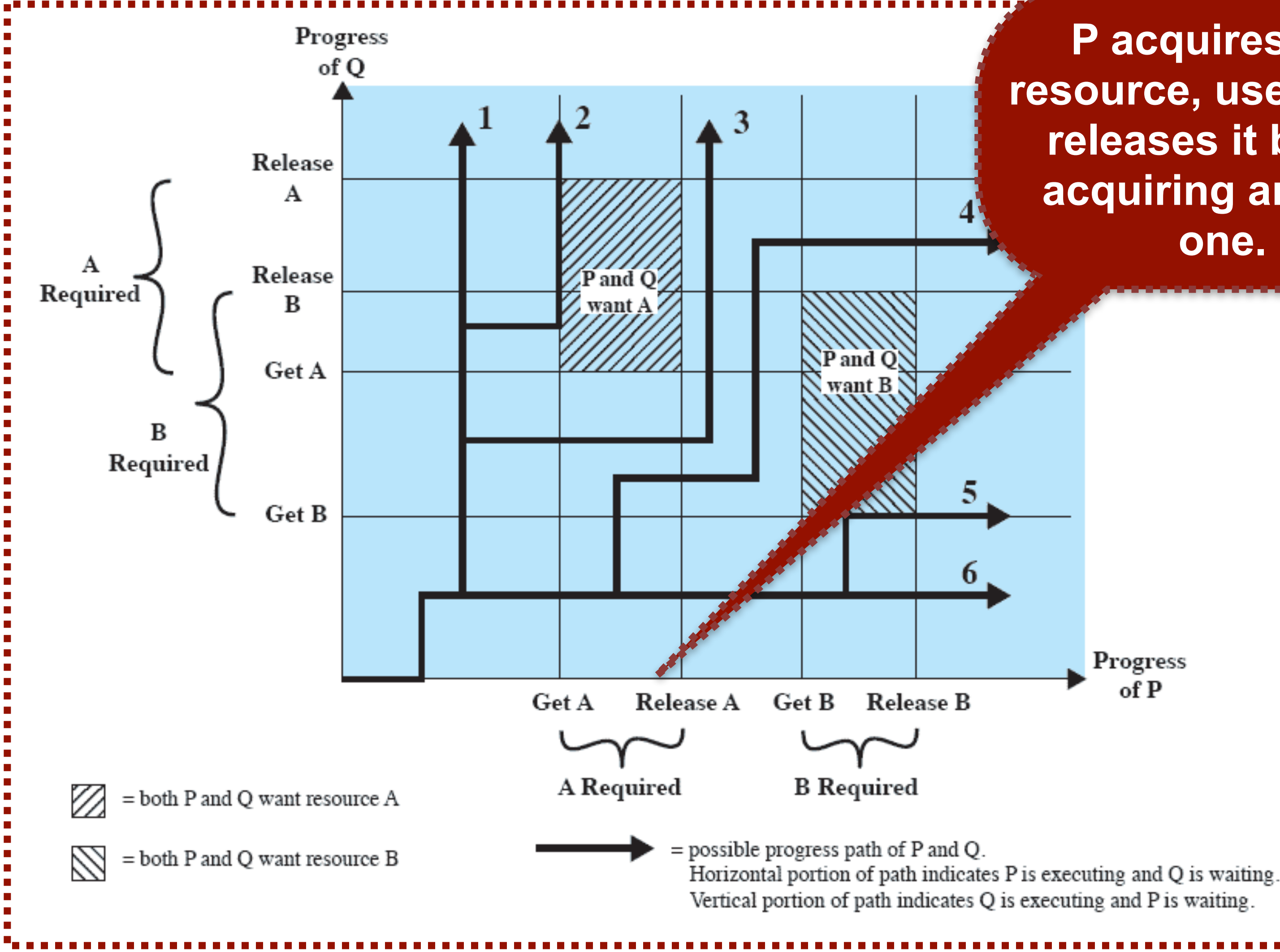
Deadlock: Joint Program Diagram



Deadlock: Joint Program Diagram



No Deadlock



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

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 =>…

MONASH University

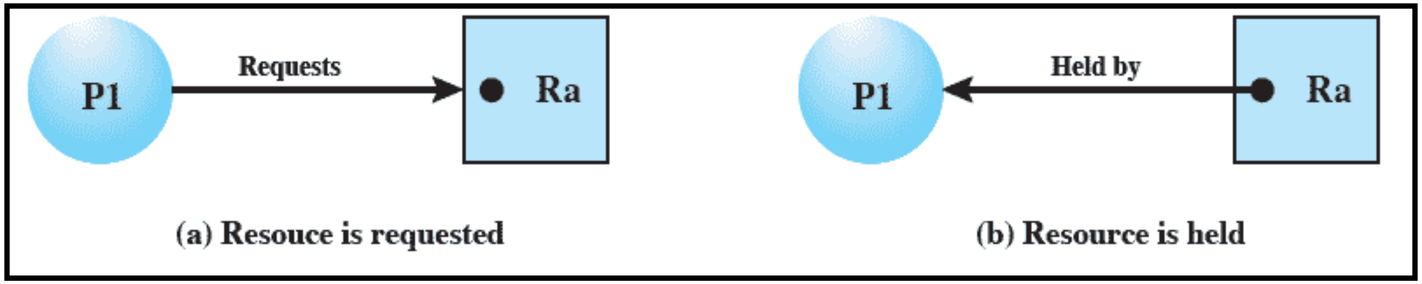
Consumable Resources: Two Processes Communicating

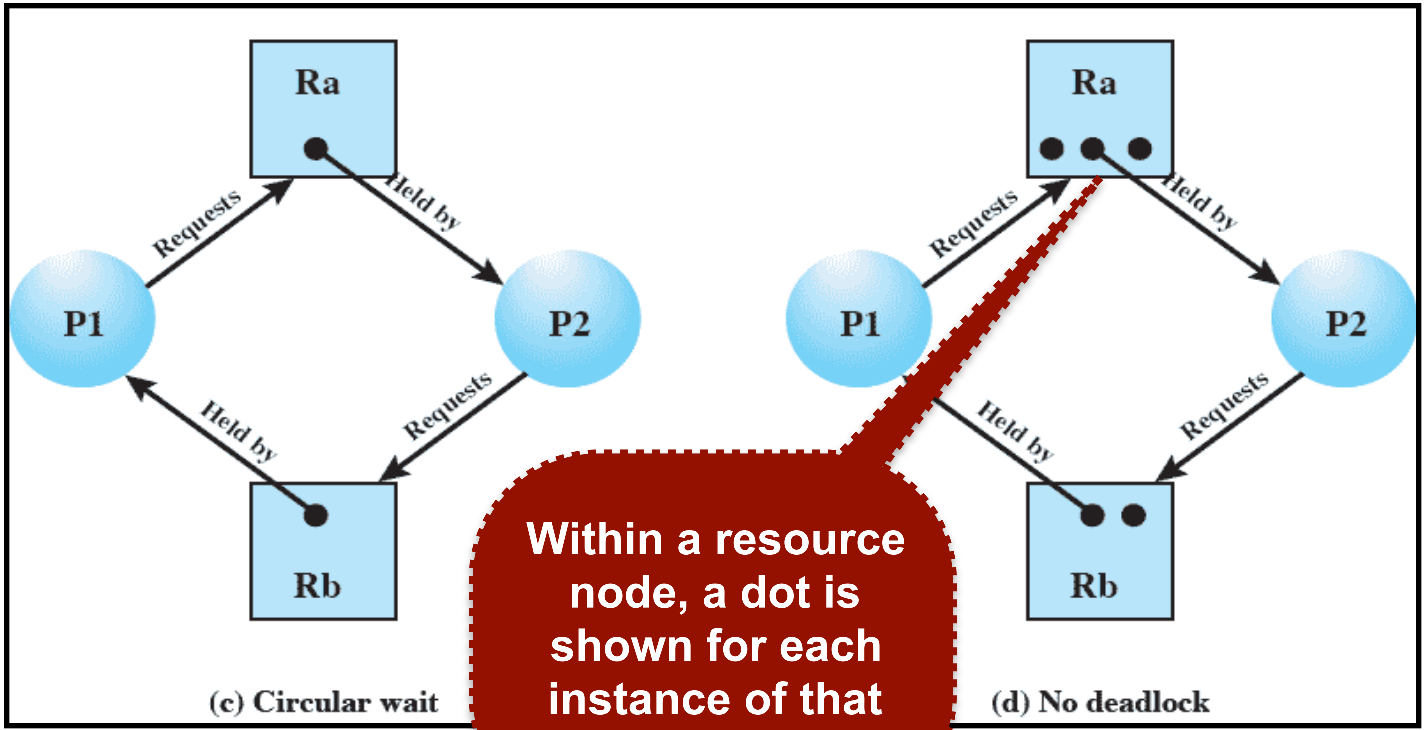
- Consider a pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process:

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



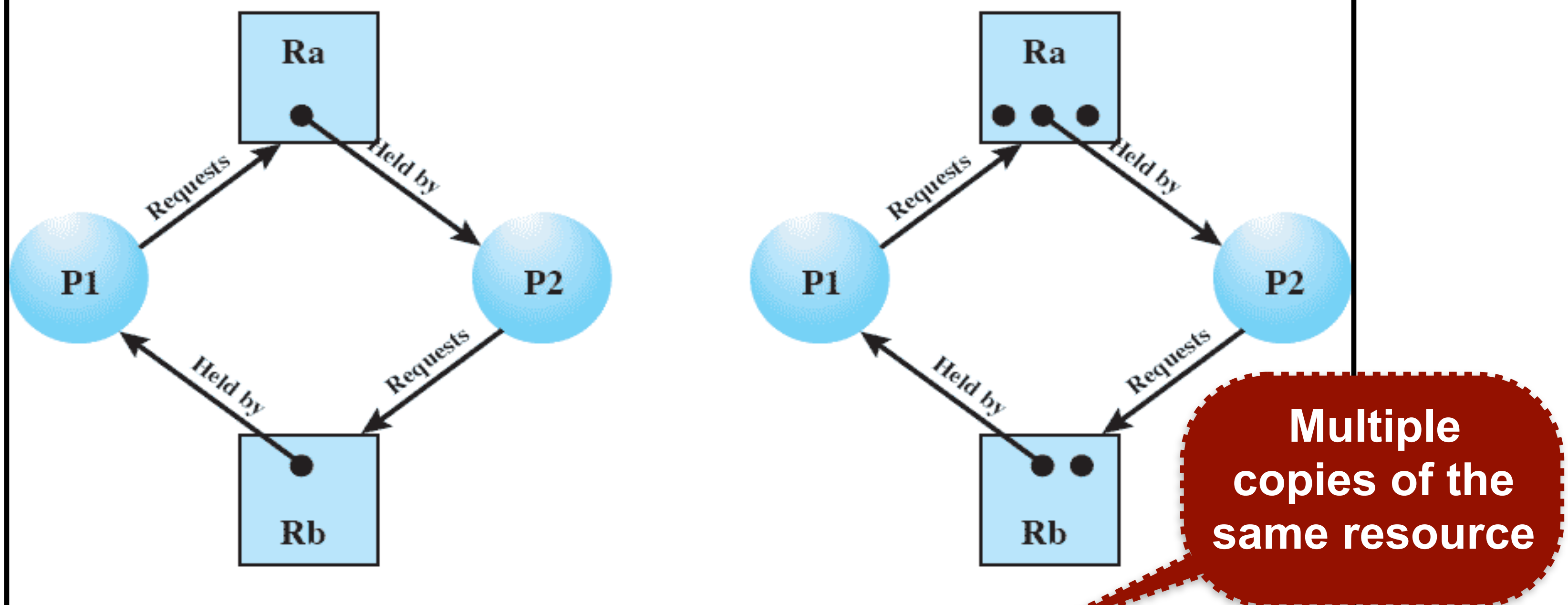


Within a resource node, a dot is shown for each instance of that resource

Resource Allocation Graph

(a) Resouce is requested

(b) Resource is held

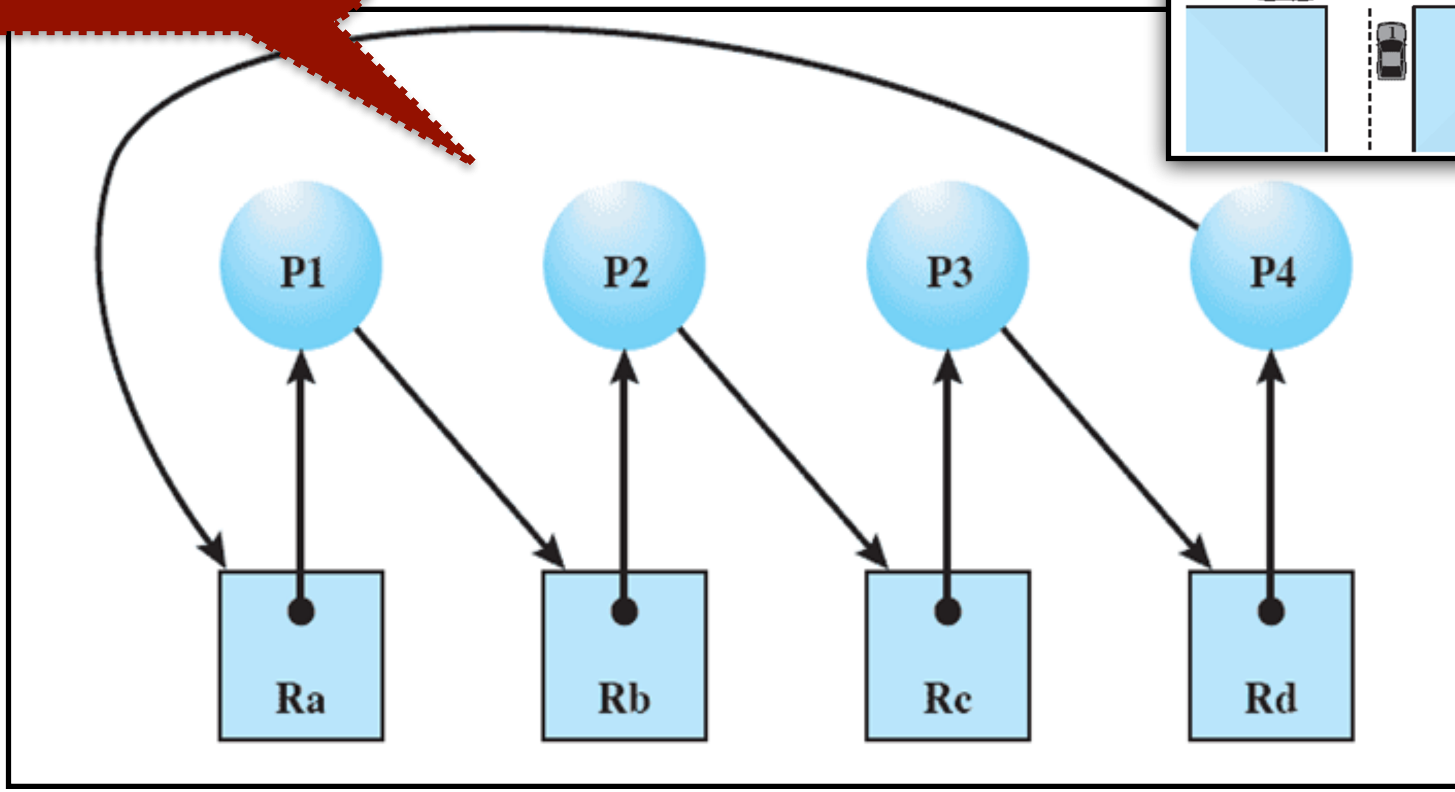


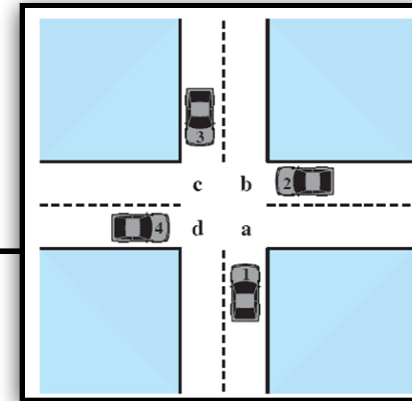
(c) Circular wait

(d) No deadlock

Deadlock: Example

Circular chain of processes and resources that results in deadlock





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 3 conditions are indirect (necessary) for a possibility of deadlock

MONASH University

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, 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

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

Mutual Exclusion

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

Might not know all the resources needed in advance

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

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 some level of 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

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

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

Need Matrix

Resource vector **R**

R1	R2	R3
9	3	6

Available vector **V**

R1	R2	R3
0	1	1

(a) Initial state

Amount of existing resources

Resources available after allocation

Process P2 makes the remaining resource request of R3

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

Avoidance Strategy: Determination of a Safe State

(a) Initial state

Claim matrix C

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

Allocation matrix A

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

C – A

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

Resource vector R

R1	R2	R3
9	3	6

Available vector V

R1	R2	R3
0	1	1

(b) P2 runs to completion

Claim matrix C

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

Allocation matrix A

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

C – A

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

Resource vector R

R1	R2	R3
9	3	6

Available vector V

R1	R2	R3
6	2	3

P2 runs to completion and releases its resources

Avoidance Strategy: Determination of a Safe State

Claim matrix C

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

Allocation matrix A

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

C – A

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

Resource vector R

R1	R2	R3
9	3	6

Available vector V

R1	R2	R3
6	2	3

(b) P2 runs to completion

Claim matrix C

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

Allocation matrix A

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

C – A

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

Resource vector R

R1	R2	R3
9	3	6

Available vector V

R1	R2	R3
7	2	3

(c) P1 runs to completion

P1 makes the request for the remaining resources

Avoidance Strategy: Determination of a Safe State

Claim matrix C

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

Allocation matrix A

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

C – A

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

Resource vector **R**

R1	R2	R3
9	3	6

Available vector **V**

R1	R2	R3
9	3	4

(d) P3 runs to completion

P3 runs to completion

The state defined originally is a safe state

Avoidance Strategy: 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**

SAFE-STATE
P2 → 1-R1, 2-R3

(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**

UNSAFE-STATE
P1 → 1-R1, 1-R3

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

Deadlock Avoidance: Implementation Logic

```
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

Deadlock Avoidance: Implementation Logic

```
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)

Assume that process will complete and will release its resources

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 resource

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

Request matrix Q

	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

Allocation matrix A

	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

Resource vector

R1	R2	R3	R4	R5
2	1	1	2	1

Available vector

R1	R2	R3	R4	R5
0	0	0	0	1

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

Rollback and restart mechanisms required

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

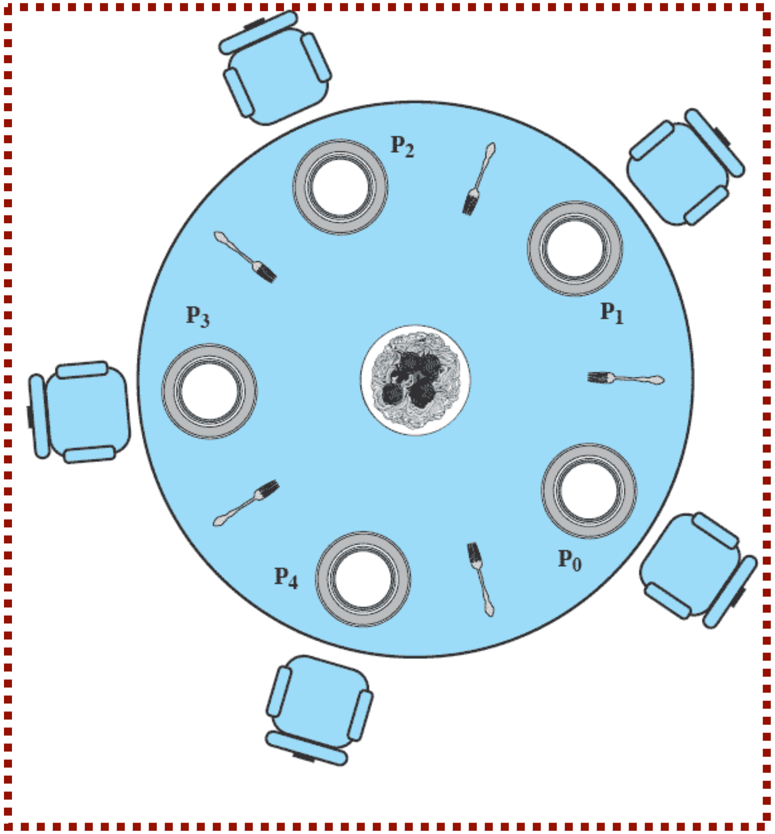
Which processes to abort or to preempt the resources?

- Successively preempt resources until deadlock no longer exists

What is the dining philosophers problem?
(The synchronisation and deadlock problem)

The Dining Philosophers Problem

- 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 5

- 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.
- Deadlock can be dealt with by three approaches — prevention, avoidance, and detection.

Next week: Uniprocessor scheduling algorithms