

Process Synchronisation continued and an introduction to Deadlock

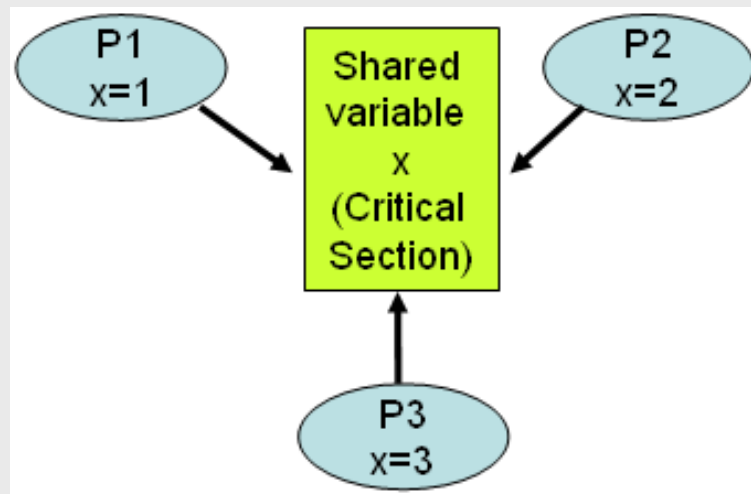
Mohammad S. Hasan

Staffordshire University, UK

Lecture Outline

- ◆ **Critical Section problem**
- ◆ **Semaphores as a classical solution to the critical section problem**
- ◆ **Semaphores as process synchronisation and co-ordination primitives**
- ◆ **The use of Resource Allocation Graphs**
- ◆ **Introduction to the concept of deadlock**

Critical Section Problem



Two or more entities share one common resource

Entities

- Vehicle, cars, trains

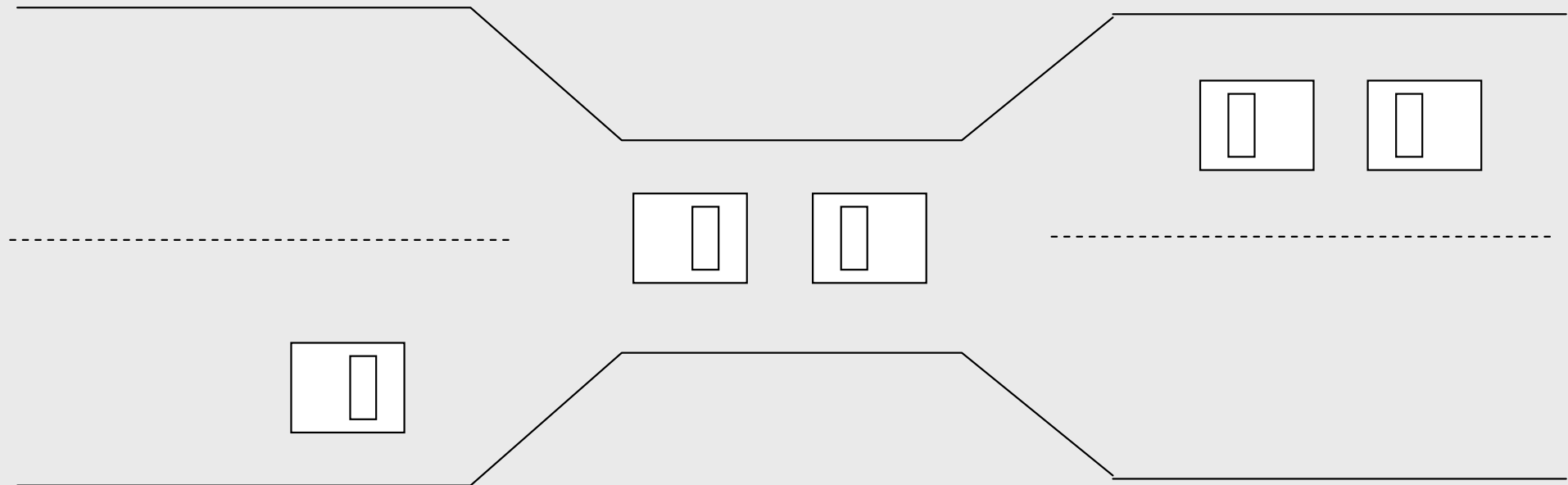
- Processes

Resource

- Narrow bridge

- Memory location, hard disk

Bridge Crossing Example



Bridge Crossing Example (Cont.)

- ◆ Traffic can flow across bridge in one direction only
- ◆ Each section of a bridge can be viewed as a resource.
- ◆ If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- ◆ Several cars may have to be backed up if a deadlock occurs.
- ◆ Starvation is possible.

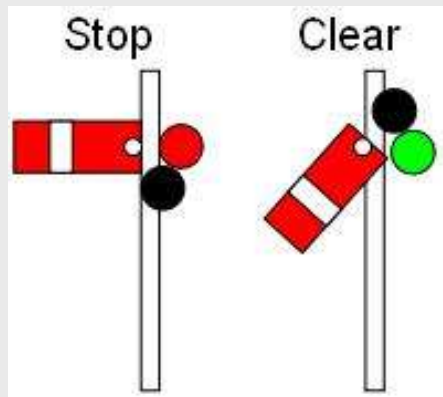
More on Critical Section Problem – Shared train / tube line



Imagine

- One tube line to Heathrow airport
- A train is coming from the airport
- Other trains trying to go to Heathrow airport
- Have to wait at the signal
- Can go to Heathrow when signal is GREEN

Semaphore



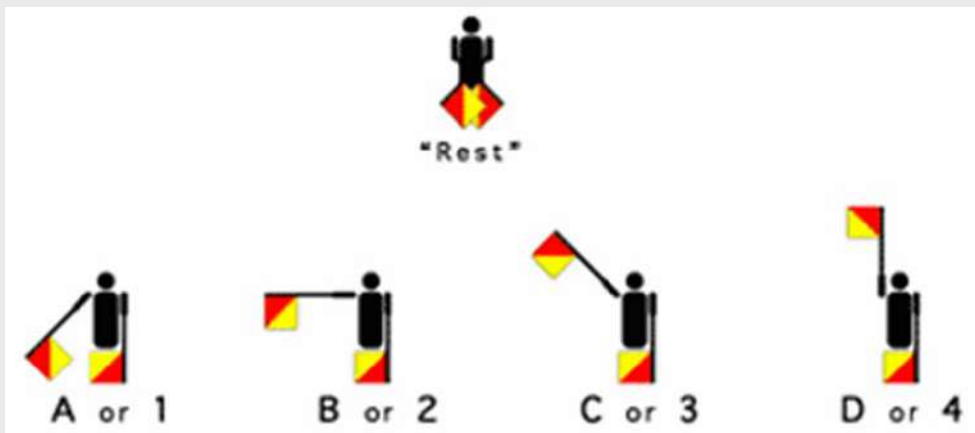
◆ To show the status of the shared resource

◆ Shared Train line

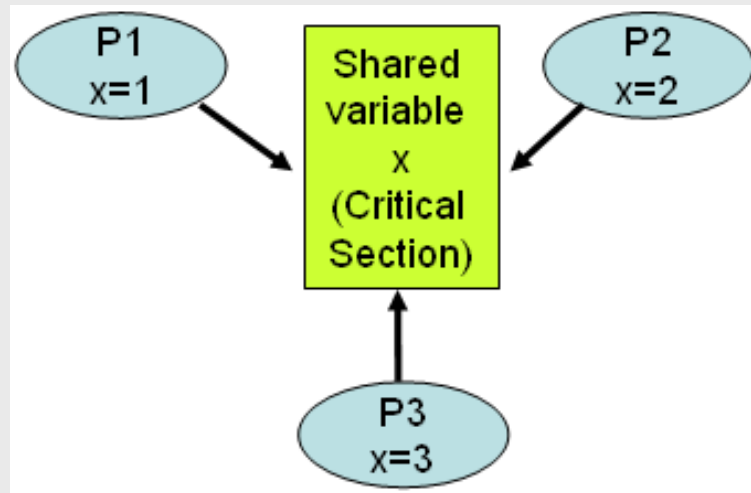
- Wait for a passing train
- Go ahead

◆ Degree of requests

- Rest = Shared resource free
- A = one entity waiting
- B = two entities waiting
- C = three entities waiting etc.



Semaphore as solution to Critical Section Problem

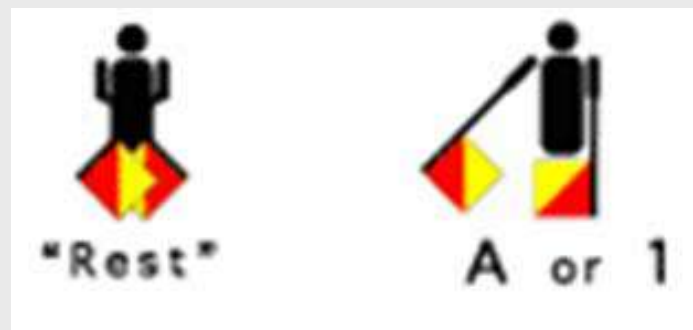


◆ When no process is writing to variable “x”

● OS shows “Rest” to all processes

◆ When P1 is writing to variable “x”

● OS shows “A or 1” to all other processes e.g. P2 and P3 trying to write to “x”



Semaphore in OS – Original Definition

- ◆ More general solution to critical section problems
- ◆ Semaphore S – integer variable that can only be accessed via two indivisible (atomic) operations

wait (S): while $S \leq 0$ do *no-op*;
 $S := S - 1$;

signal (S): $S := S + 1$;

Example: Critical Section of n Processes

◆ Shared variables

- **var mutex : semaphore**
- **initially mutex = 1**

◆ Process P_i

◆ Note:

- *mutex* is the short version of Mutual Exclusion

◆ Generic structure of a process

```
repeat
    wait(mutex);
    critical section
    signal(mutex);
    remainder section
until false;
```

Semaphore (Cont.)

- ◆ The original semaphore definition requires *busy waiting*.
 - Wasting CPU cycles that could be used productively by other processes.
 - Often called a *spinlock* (process 'spins' while waiting for lock).

Semaphore Implementation

- ◆ Rather than busy waiting we can define a semaphore to block a process that cannot proceed placing it on a queue of processes associated with the semaphore.

- **signal(S)** then removes a process from waiting queue

- ◆ Define a semaphore as a record

```
type semaphore = record  
    value: integer  
    L: list of process;  
end;
```

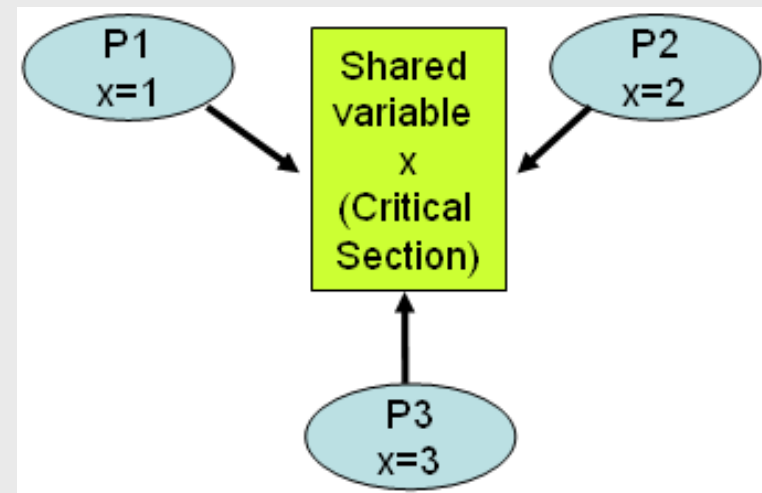
Semaphore Implementation (Cont.)

- With semaphore defined as in previous slide we specify two simple operations:

- block(P)** suspends the process that invokes it.
- wakeup(P)** resumes the execution of a blocked process P.

- Process P1 is writing to “x”, process P2 and P3 trying to write to “x”

- P2 and P3 will be blocked
- P2 and P3 will be in the list



Semaphore Implementation (Cont.)

◆ Semaphore “wait(S)” operation now defined as

```
wait(S):      S.value := S.value – 1;  
              if S.value < 0  
              then begin  
                  add this process to S.L;  
                  block;  
              end;
```

Semaphore Implementation (Cont.)

◆ Semaphore “signal(S)” operation now defined as

```

signal(S):      S.value := S.value + 1;
                  if S.value ≤ 0
                    then begin
                      remove a process P from S.L;
                      wakeup(P);
                    end;

```

Semaphores for general synchronisation

◆ Example synchronisation problem

● Execute B in process Pj only after A executed in process Pi

◆ Solve problem by using a semaphore initialised to 0 (in this example we will call it *flag*)

◆ Code:

Pi

..

A

signal(flag)

Pj

..

wait(flag)

B

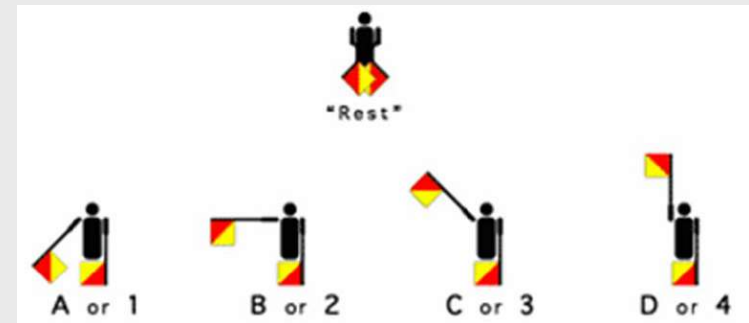
Two Types of Semaphores

Counting semaphore

- integer value can range over an unrestricted domain.
- Useful for controlling access to a type of resource with some given number of instances of that resource e.g. with n resources of a given type, with semaphore value set to n , process will wait when n instances of resource are already allocated.

Binary semaphore

- integer value can range only between 0 and 1.
- Useful for implementing mutual exclusion as we have seen.

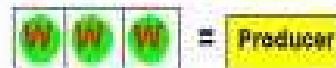


Classical Synchronisation Problem - Bounded Buffer

Producer:

1. Make new widget → 

2. If buffer is full, go to sleep



3. Put widget in buffer



4. If buffer was empty, wake consumer



Buffer

Consumer:

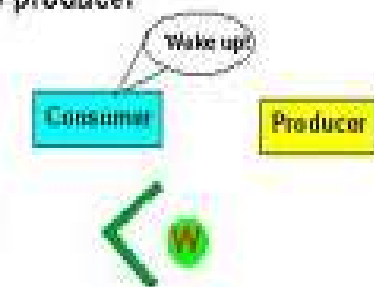
1. If buffer is empty, go to sleep



2. Take widget from buffer



3. If buffer was full, wake producer



4. Consume the Widget

Classical Synchronisation Problem - Bounded Buffer (Cont.)

◆ Shared data - as we have seen before

- buffer for shared data - an array organised as a circular buffer

- n is the size of the buffer

◆ BUT IN ADDITION we now have semaphores:

◆ full, empty, mutex initialised as below:

<code>full = 0;</code>	<code>// gives number of elements in buffer</code>
<code>empty = n;</code>	<code>// gives no. of empty spaces in buffer</code>
<code>mutex = 1;</code>	<code>// guarantees mutual exclusion</code>

Bounded-Buffer Problem (Cont.)

◆ Producer process

repeat

...

produce an item

...

wait(empty);

wait(mutex);

put item in buffer

signal(mutex);

signal(full);

until false;

◆ Consumer process

repeat

wait(full)

wait(mutex);

...

remove an item from buffer

...

signal(mutex);

signal(empty);

...

consume the item

...

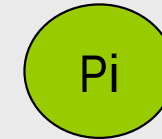
until false;

Resource-Allocation Graphs

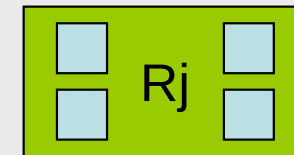
- ◆ A set of vertices V and a set of edges E .
- ◆ V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system.
 - $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system.
- ◆ request edge – directed edge $P_i \rightarrow R_j$
- ◆ assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

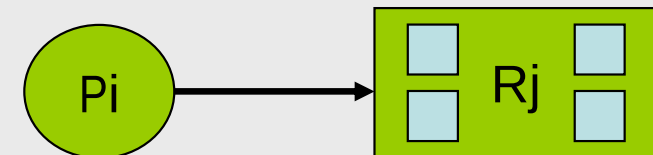
◆ A Process



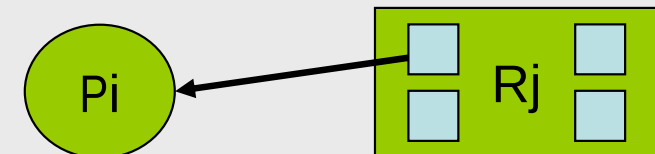
◆ Resource Type with 4 instances



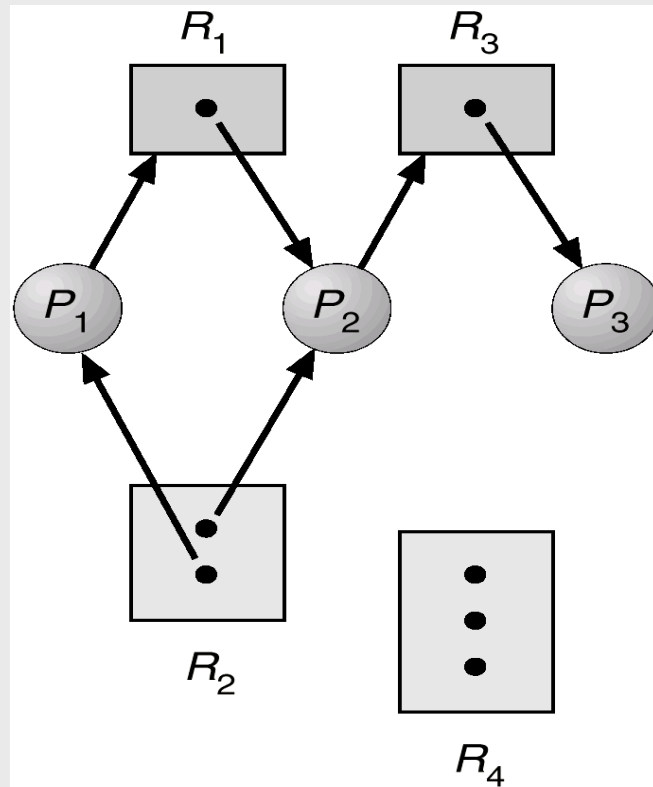
◆ P_i requests instance of R_j



◆ P_i is holding an instance of R_j



Example of a Resource Allocation Graph



◆ P1

- is allocated one R2 instance
- is requesting for R1

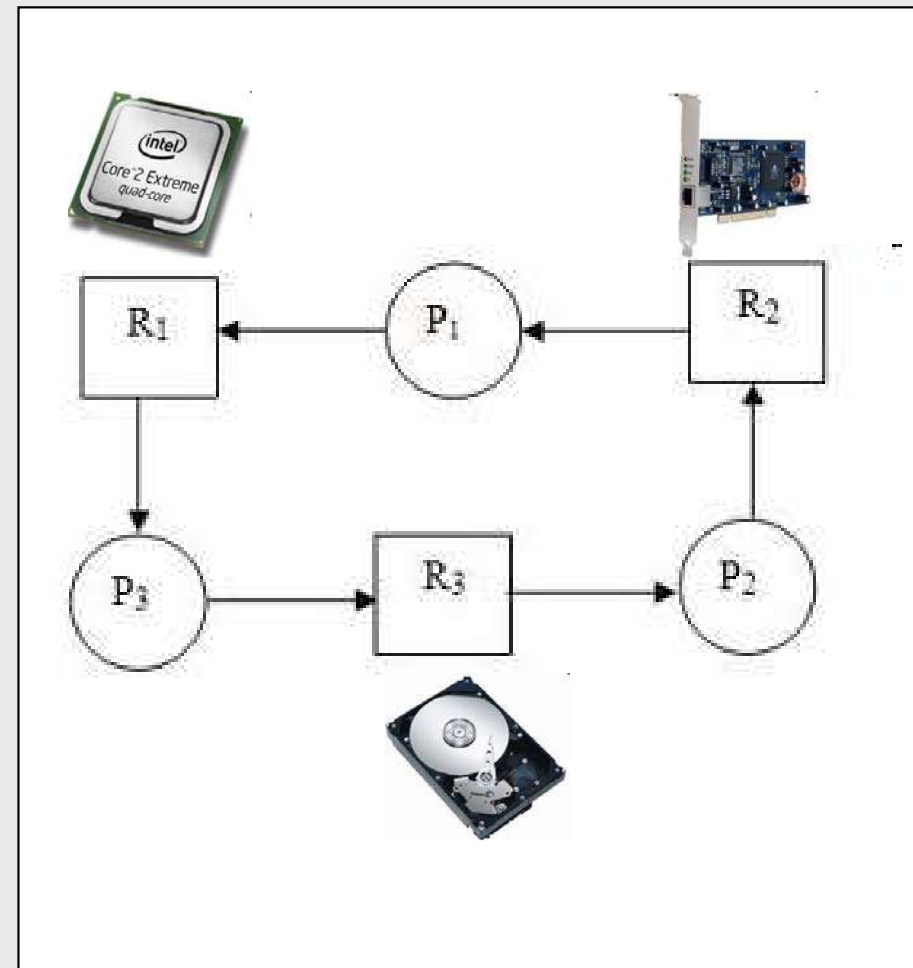
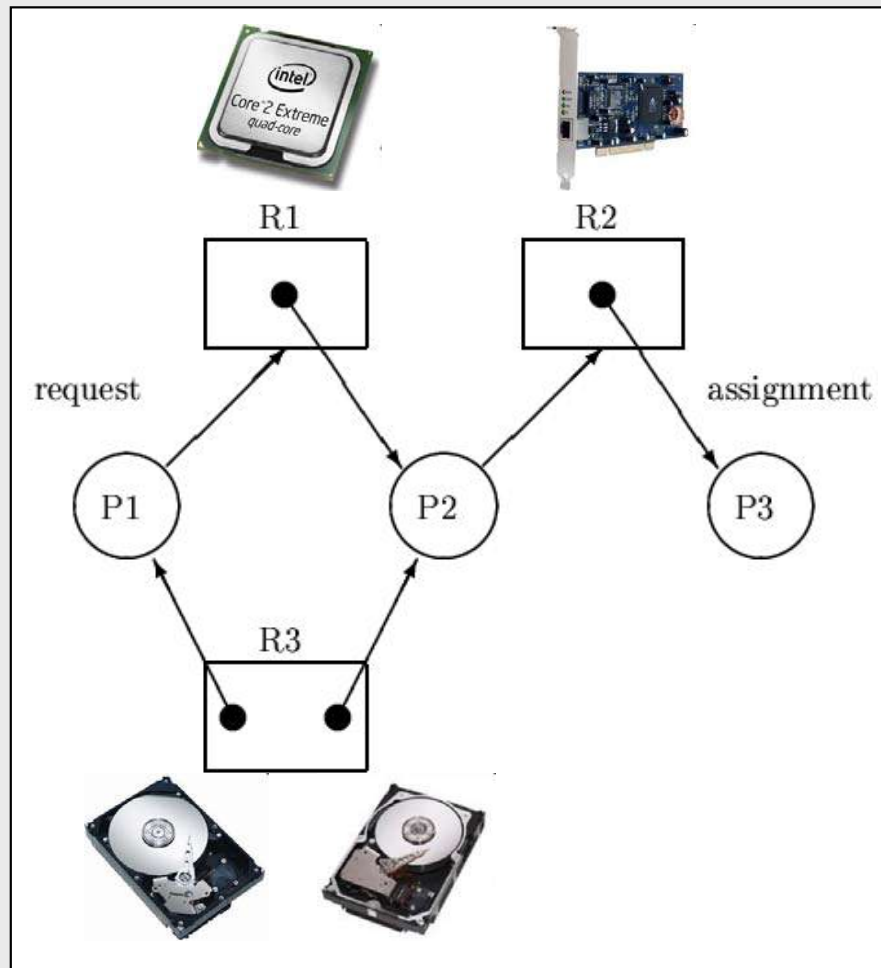
◆ P2

- is allocated R1 and one R2 instance
- is requesting for R3

◆ P3

- is allocated R3

More on Resource Allocation Graph



Deadlock and Starvation



◆ Deadlock

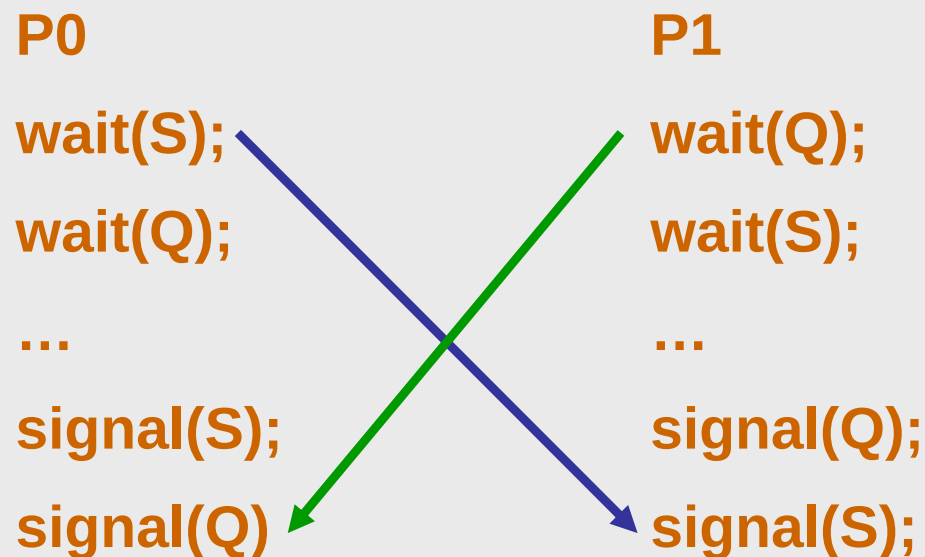
- two or more processes are waiting indefinitely for an event that can only be caused by one of the processes that is waiting

◆ Starvation

- indefinite blocking of a process on a queue

Semaphore and Deadlock

- ❖ Care needs to be taken in use of semaphores, otherwise Deadlocks can occur
- ❖ For example, let S and Q be two semaphores initialised to 1, then process P0 and P1 are in Deadlock



The Deadlock Problem – in general

- ◆ A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

◆ Example

- System has 2 disk drives
- P1 and P2 each holds one disk drive and each needs another one.
- semaphores A and B, initialised to 1

◆ Process Structure

P0

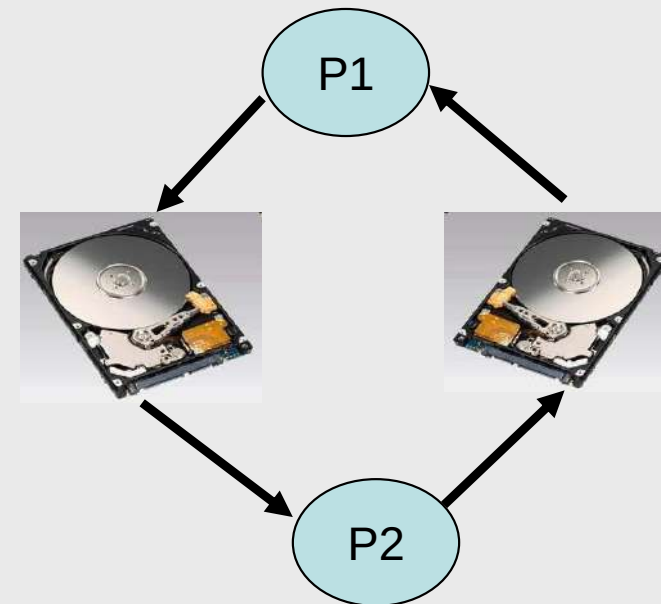
wait (A)

wait (B);

P1

wait(B);

wait(A);



Deadlock – 4 necessary conditions

◆ **Deadlock can arise if four conditions hold simultaneously:**

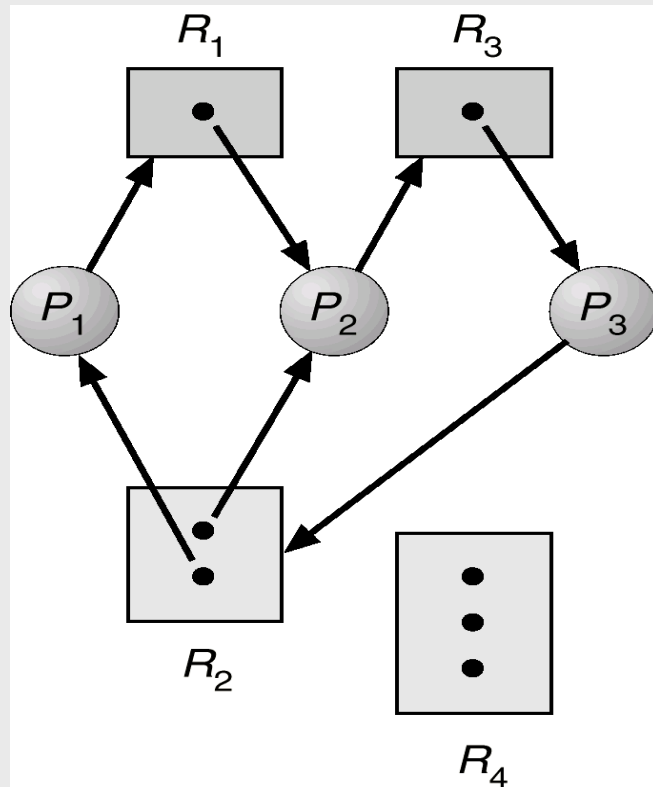
- **1. Mutual exclusion: only one process at a time can use a resource.**
- **2. Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes.**

Deadlock – 4 necessary conditions (Cont.)

- **3. No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **4. Circular wait:** there exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0 .

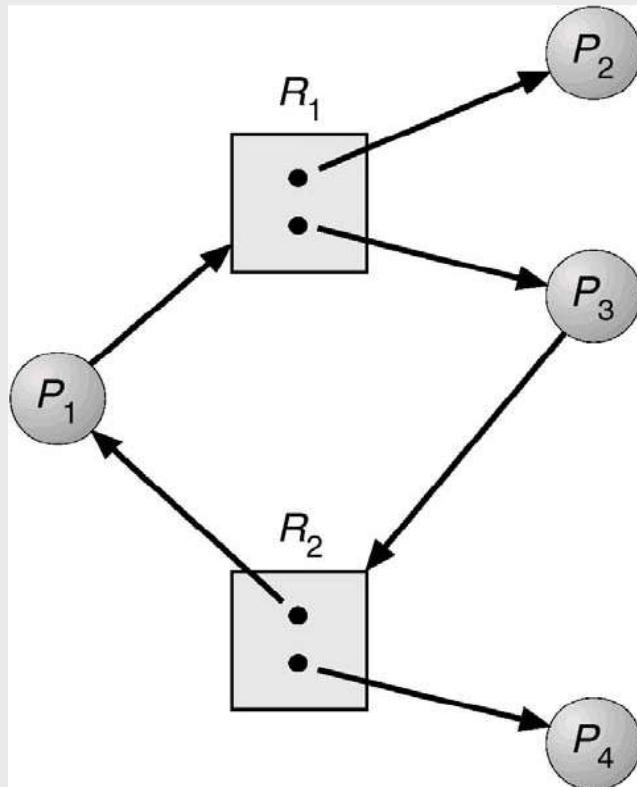
Resource Allocation Graph showing a Deadlock

◆ Circular Wait



- **P1 is waiting for R1 held by P2**
- **P2 is waiting for R3 held by P3**
- **P3 is waiting for R2 instances held by P1 and P2**

Resource Allocation Graph with a cycle but no Deadlock



◆ **P2 completes**

● **P2 releases one instance of R_1**

● **Released instance is assigned to P_1**

◆ **P1 completes**

◆ **P4 completes**

● **P4 releases one instance of R_2**

● **Released instance is assigned to P_3**

◆ **P3 completes**