Operating Systems Concepts

- Chapter 4: **Concurrency**
  - How do I get my computer to do more than one thing at a time?
  - Eg. Background printing
  - Big idea: processes
  - Big issue: synchronisation, mutual exclusion, deadlock

Olav Beckmann
Huxley 449
http://www.doc.ic.ac.uk/~ob3

Acknowledgements: There are lots. See end of Chapter 1.

Home Page for the course:
http://www.doc.ic.ac.uk/~ob3/Teaching/OperatingSystems/
This is only up-to-date after I have issued printed version of the notes, tutorials, solutions etc.

Operating Systems Concepts: Chapter 4: Concurrency and processes

- A key function of an OS is to support concurrency:
  - allow several different applications to run on the same machine at the same time
  - allow some activities to occur "in the background"
  - allow several users to share a machine
- This section of the course concerns
  - How concurrent processes are implemented
  - Issues in writing programs to work properly when running concurrently
- Background: try the Windows NT command "taskmgr" and the Unix/Linux command "ps -aux"

Using a Textbook

- Tanenbaum Chapter 2
- Nutt Chapters 8, 9 and 10
- Stallings Chapters 5 and 6
- Optional second-term course, "Concurrent and Distributed Programming"
- Jeff Magee and Jeff Kramer's book, Concurrency: State Models & Java Programs
  - http://www-dse.doc.ic.ac.uk/~jnm/book/index.html

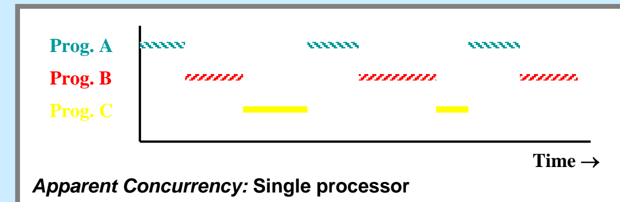
Process Control Overview

- Modes of execution
  - Processes divided by their class user or system (kernel or OS)
  - *Privileges* and *trust*
  - Less privileged mode – user mode
- Allows us to protect the operating system and key operating system tables from interference by user programs
  - E.g. in kernel mode the software has complete control of processor

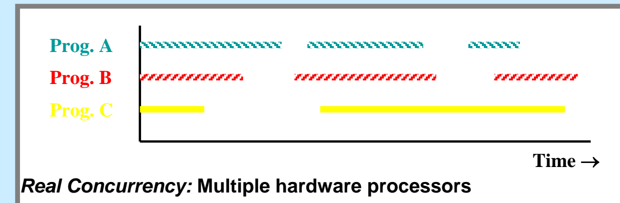
Processes cont

- When program is running and needs to access system,
  - mode is changed (change mode routine)
  - and changed back when finished
- OS checks that the change mode is allowed
- Typical kernel functions are
  - process management, *Concurrency*
  - memory man,
  - IO man,
  - Security,
  - and support functions: such as interrupt handling.

Concurrent Processes



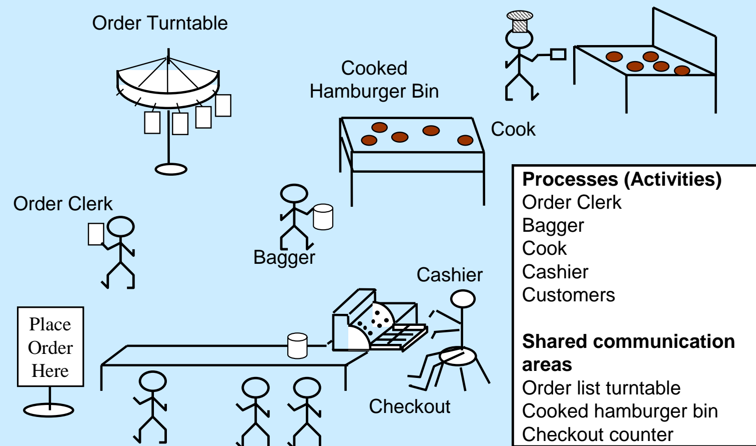
Apparent Concurrency: **Single processor**



Real Concurrency: **Multiple hardware processors**

Each activation of a program is termed a *Process*

An Example of Concurrent Activity

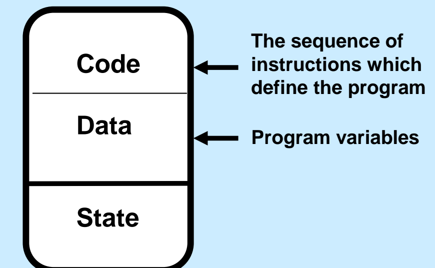


Processes (Activities)
Order Clerk
Bagger
Cook
Cashier
Customers

Shared communication areas
Order list turntable
Cooked hamburger bin
Checkout counter

Program = a sequence of instructions
Process = an activity consisting of the execution of a program

A Process is represented in a computer by:



The sequence of instructions which define the program

Program variables

What does the process state consist of ?

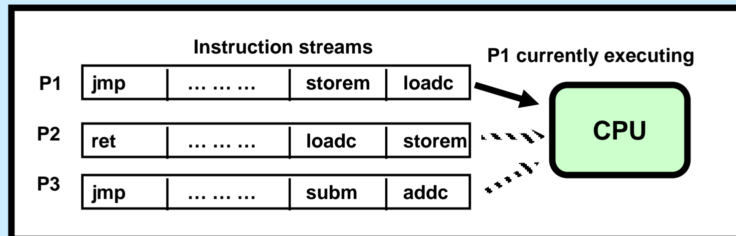
Processor:

- The agent which executes a program.
- A process runs on a processor.

Concurrent Processes:

- Activation of more than one process.

Apparent concurrency can be achieved by switching a processor between a set of processes - *instruction interleaving.*



Recall… An interrupt can occur after the execution of any instruction

```
PC = 0;
do {
  fetch( );
  PC=PC+1;
  execute( );
  if (InterruptRequest &&
      InterruptEnabled) {
    InterruptEnabled = false;
    Mem[0] = PC;
    PC = Mem[1];
  }
} forever;
```

Saving the PC during an interrupt, and restoring it on return

Save the current PC somewhere (so we can return later), and branch to the address of the interrupt handler

Recall…

```
PC = 0;
do {
  fetch( );
  PC=PC+1;
  execute( );
  if (InterruptRequest &&
      InterruptEnabled) {
    InterruptEnabled = false;
    Mem[0] = PC;
    PC = Mem[1];
  }
} forever;
```

```
void InterruptHandler() {
  saveProcessorState();
  char ch= *KBD_PORT_ADDRESS;
  KbdBuffer.add(scanToAscii(ch));
  restoreProcessorState();
  rti; // restore PC from Mem[0] and
       // re-enable interrupts
}
```

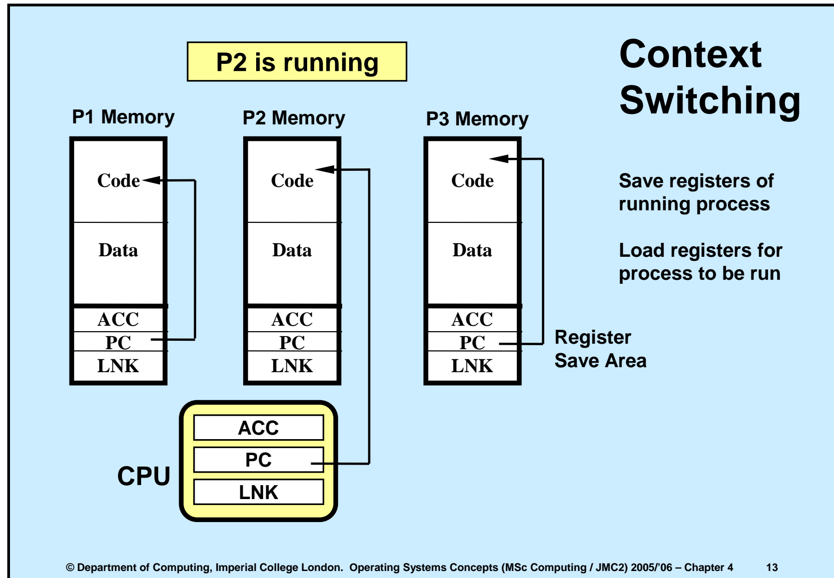

Switching between processes

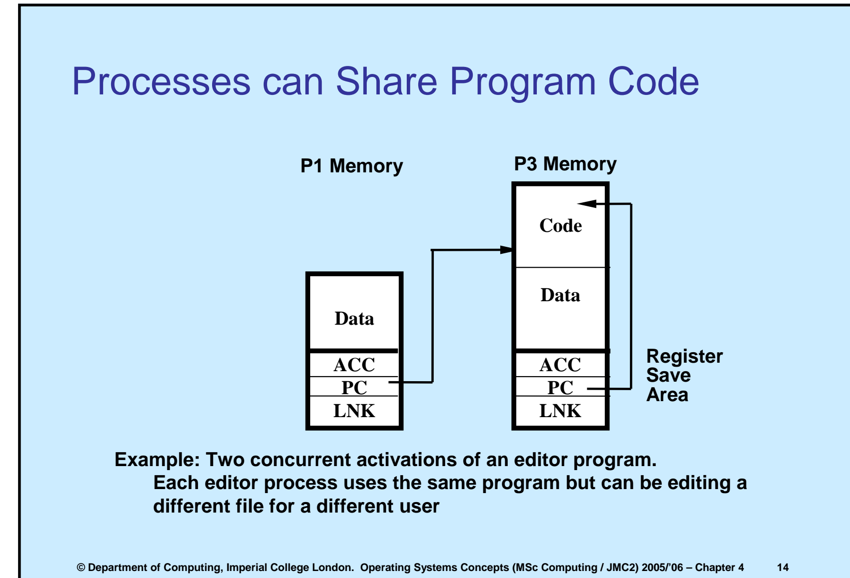
```
PC = 0;
do {
  fetch( );
  PC=PC+1;
  execute( );
  if (InterruptRequ
      InterruptEna
    InterruptEnab
    Mem[0] = PC;
    PC = Mem[1];
  }
} forever;
```

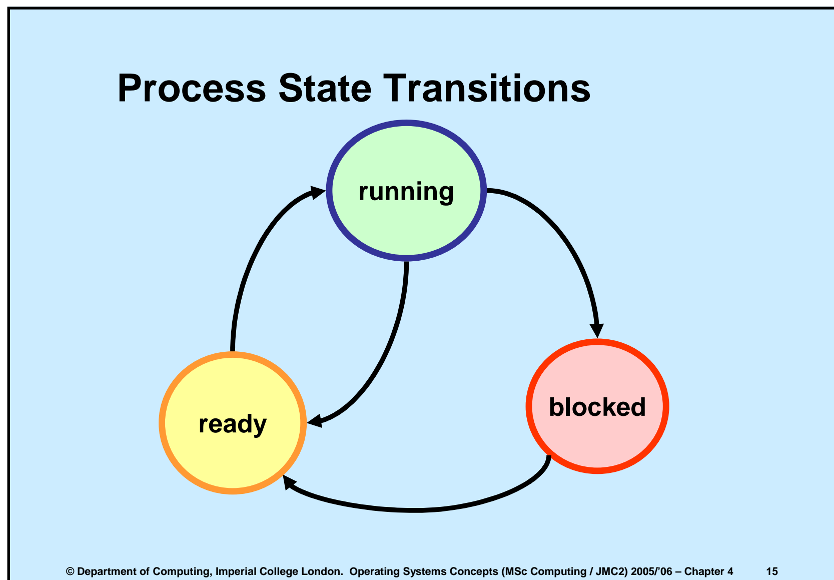
```
void InterruptHandler() {
  saveProcessorState();
```

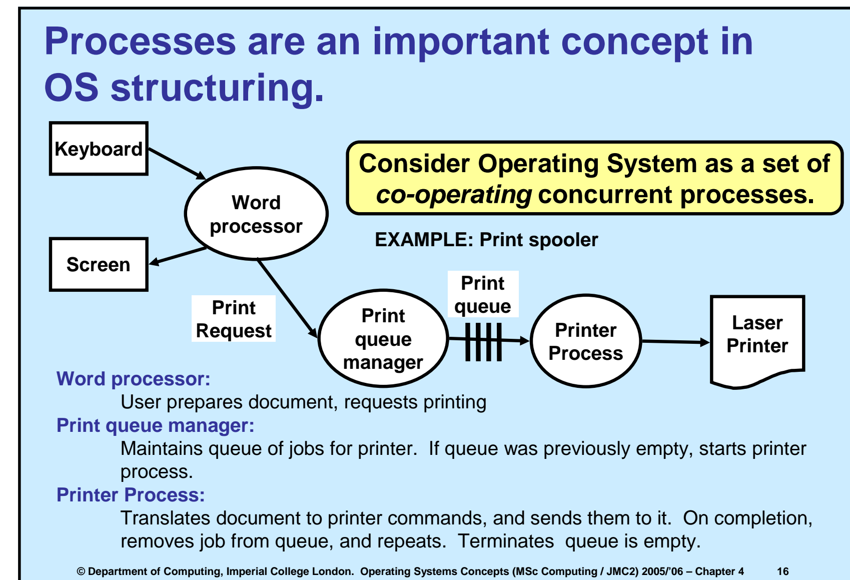
1. Handle the interrupt
2. Choose which processor state to return to

```
  rti; // restore PC from Mem[0] and
       // re-enable interrupts
}
```


Context Switching
P2 is running
P1 Memory
P2 Memory
P3 Memory
Code
Data
ACC
PC
LNK
Save registers of running process
Load registers for process to be run
Register Save Area
CPU
ACC
PC
LNK
© Department of Computing, Imperial College London. Operating Systems Concepts (MSc Computing / JMC2) 2005/'06 – Chapter 4 13

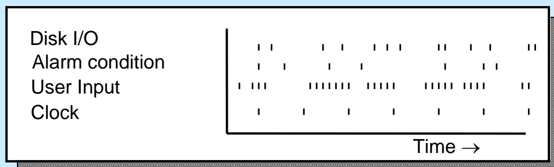

Processes can Share Program Code
P1 Memory
P3 Memory
Code
Data
ACC
PC
LNK
Register Save Area
Example: Two concurrent activations of an editor program.
Each editor process uses the same program but can be editing a different file for a different user
© Department of Computing, Imperial College London. Operating Systems Concepts (MSc Computing / JMC2) 2005/'06 – Chapter 4 14


Process State Transitions
running
ready
blocked
© Department of Computing, Imperial College London. Operating Systems Concepts (MSc Computing / JMC2) 2005/'06 – Chapter 4 15


Processes are an important concept in OS structuring.
Keyboard
Screen
Word processor
Consider Operating System as a set of co-operating concurrent processes.
EXAMPLE: Print spooler
Print Request
Print queue manager
Print queue
Printer Process
Laser Printer
Word processor:
User prepares document, requests printing
Print queue manager:
Maintains queue of jobs for printer. If queue was previously empty, starts printer process.
Printer Process:
Translates document to printer commands, and sends them to it. On completion, removes job from queue, and repeats. Terminates queue is empty.
© Department of Computing, Imperial College London. Operating Systems Concepts (MSc Computing / JMC2) 2005/'06 – Chapter 4 16

Non- Determinism

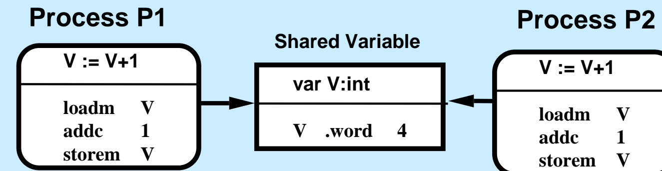
- Operating Systems are ***non-deterministic*** in that they must respond to events (I/O) which occur in an unpredictable order.
- Events (or interrupts) cause process switches.
  - e.g. an I/O completion interrupt will cause the OS to switch to an I/O process.



- **The way a system switches between processes cannot be pre-determined, since the events which cause the switches are not deterministic.**
  - **e.g. cannot tell when a user will type the next character**
- **The interleaving of instructions, executed by a processor, from a set of processes is non-deterministic.**

Process Interaction

EXAMPLE - Updating a shared variable (e.g. bank balance)

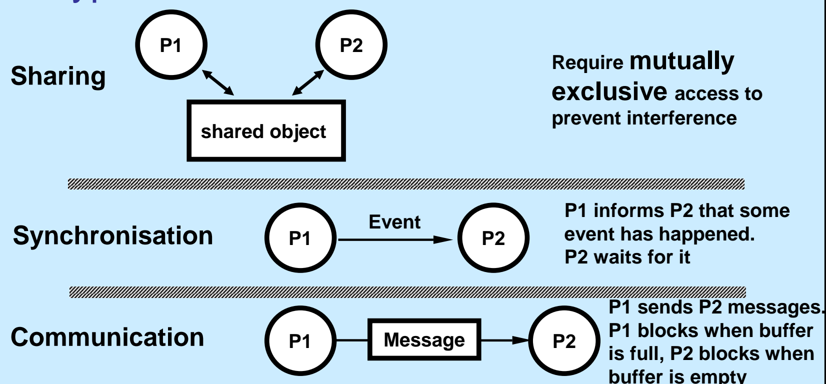


Remember instructions can be arbitrarily interleaved.

Process	Instruction	Accumulator	Value of V
P1	loadm V	4	4
P2	loadm V	4	4
P1	addc 1	5	4
P2	addc 1	5	4
P1	storem V	5	5
P2	storem V	5	5

PROCESS INTERACTION MUST BE CONTROLLED

Types of Process Interaction



Mutual Exclusion, **Synchronisation** and **Communication** are closely related.

Critical Sections

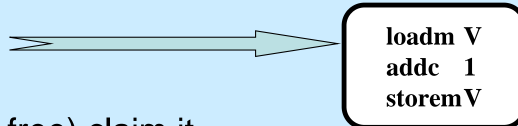
- A critical section is a sequence of instructions which must be executed by at most one process at a time
- Analogy: a bridge strong enough for only one vehicle
- A code section is critical if it

1. **Reads a memory location which is shared with another process**
2. **Updates a shared memory location with a value which depends on what it read**

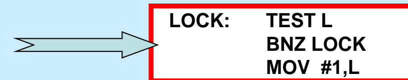
Critical Sections - Examples

if (hotel room is available) book it

- v = v+1



- If (lock is free) claim it



Protecting critical sections - achieving mutual exclusion

- We need to make sure that at most one process can execute the critical section at once
- "Mutual exclusion" – the presence of one process in the critical section ensures that all others are excluded
- So when a process tries to enter a critical section, it may have to wait until it has been vacated

Locks

Process Interaction Mechanism #1

lock L = binary value

L= 0 lock open/free
L= 1 locked

Primitive Operations

```
LOCK(L) ::=      wait until L == 0 then do L := 1
UNLOCK(L) ::=    L := 0
```

Mutual Exclusion using locks:

```
void increment(int &V) {
    LOCK(L)
    // access shared object
    V = V+1;
    UNLOCK(L)
}
```

"critical section"

- If the lock L is initially 0:
  - first process to perform the LOCK operation sets it to 1.
- Subsequent processes will be *blocked* at the LOCK operation
  - so cannot access shared object until first process releases the lock.
- In this way, locks can be used to implement mutual exclusion.
- Only one process can be executing in its critical section at any one time.

(That's what *critical section* means.)

A Non-Implementation of Locks

You might try to implement locks like this:

```
LOCK:   TEST L
        BNZ LOCK
        MOV  #1,L

UNLOCK: MOV #0, L
```

TEST	sets the condition register Z bit if L is 0.
BNZ	jumps if Z is not set.
MOV #n,L	sets L to constant n

This does not work, because the instruction sequence for LOCK is interruptible. Imagine two processes **P1** & **P2** trying to **LOCK L** - initially **0**.

EXERCISE: **show how a bad execution sequence can let both processes through the lock together:**

Process	Instruction	Z bit	Value of L
P1	TEST L	Z	0
P2	TEST L	Z	0
P2	MOV #1,L		1
P1	MOV#1,L		1

A Better Implementation of Locks

You might try to implement locks like this:

```
LOCK:   STI          // enable interrupts
        CLI          // disable interrupts
        TEST L
        BNZ LOCK
        MOV  #1,L
        STI          // re-enable interrupts
```

```
UNLOCK:  MOV #0, L
```

- Interrupts must be enabled (at least briefly) while looping waiting for the lock to become available
- Interrupts must be disabled during the critical section – between reading L (**TEST L**) and writing to L (**MOV #1,L**).
- This is a common technique for achieving mutual exclusion but has some serious problems…

Disabling interrupts to achieve mutual exclusion

Problems with using interrupt disable/enable:

1. **If your critical section is long**, the interrupt response time will be adversely affected – you may miss an important interrupt
2. **In a multiprocessor**, the critical section could be executed by another processor – disabling interrupts can't stop it
3. **If you make a mistake**, and forget to re-enable interrupts… your machine will become unresponsive

TEST & SET Instruction

Another approach is to use a single *indivisible* (non-interruptible) *test & set* instruction

E.g. for the IBM 370

```
TS  L          1) Read L and set condition code if L=0;
               2) Write value 1 into L
```

Compare: **BTS** instruction in Intel x86

How can we use this instruction?

```
LOCK:   TS L
        BNZ LOCK
```

```
UNLOCK:  MOV #0, L
```

- **TS L always sets L to 1**
  - If L was one already it does not matter that it is set to 1 again.
  - It only sets the Z condition code bit if L was 0 (free) beforehand

Locks - Summary

- It's really useful to be able to run several processes (or threads) concurrently
- If the processes share data or resources, access has to be coordinated
- Mutual exclusion: only one process is allowed access at once
- A critical section is an example of where mutual exclusion is needed
- We can achieve mutual exclusion by disabling/re-enabling interrupts – but there are drawbacks
- We can achieve mutual exclusion by claiming a lock on entry, releasing it on exit – but we still have a critical section in the lock itself

Locks – Summary, continued

- Lock can be implemented by disabling/re-enabling interrupts – but a better scheme is to use an indivisible instruction
- The problem with the lock schemes we have seen so far is that they lead to a *busy wait*: a process waiting on a lock cycles in a loop using the processor
- The next section of the chapter introduces semaphores. A semaphore improves on a lock in two ways:
  - No busy wait
  - Generalises: N states instead of 2
  - **Introduced by Edsger Dijkstra in his T.H.E. operating system (1965) (http://www.cs.utexas.edu/users/joshi/ewd/EWD.html)**

Semaphores

Process Interaction Mechanism #2

If processes are competing for some resource, a *semaphore* is a simple way of keeping track of the availability of that resource.

Binary Semaphore takes values:	**0, 1**
General Semaphore takes values:	**0, 1, 2 …**

The value of a Semaphore is a *protected* variable
– only accessible through two primitive operations:

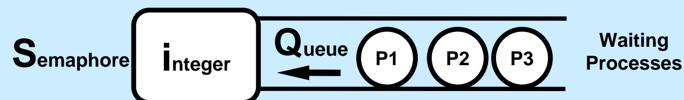
```
P(s) ::= when s > 0 do s=s-1
V(s) ::= s=s+1
```

- **P means "I want" (Please)**
- **V means "Here is" (now 'Vailable)**

- **If value is 0 when you call P, P waits until some other process - *not you!* - calls V.**
- **The P & V operations are indivisible.**
- **As with locks, can be implemented using *busy-wait***
- **Also need initialization, init(s,v) ::= s:=v**

(*Proberen*, to test, *verhogen*, to increment)

Non busy wait implementation of semaphores



P(S)	V(S)

if S.i > 0 **then**
S.i := S.i-1
else
suspend process on S.Q
end if
	if not empty(S.Q) **then**
	resume a process in S.Q
	else
	S.i := S.i +1
	end if

- **The queue is usually First In First Out (FIFO).**

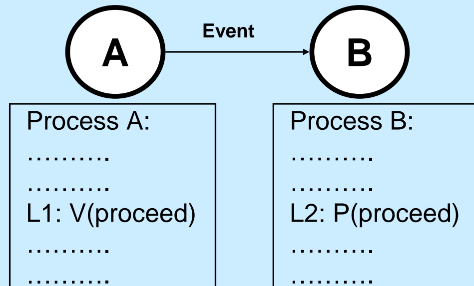
Using Semaphores:

Mutual Exclusion

```
var d: int          //shared variable
var s: semaphore
InitSema(s,1)       // initialise to 1

process p(n)
                    // s = 1
   P(s)             // s = 0
   d := d + 1
   V(s)
end p               // s = 1
```

Process can only enter critical section if s == 1.
Only one process at a time can be executing its critical section
– so get *mutual exclusion.*

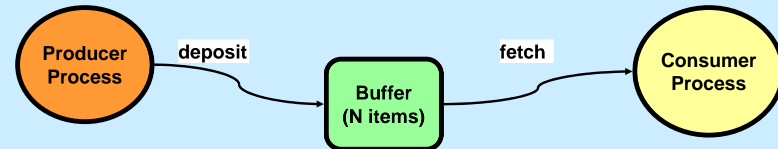
Using Semaphores: Synchronisation



- Process B must wait at L2
  - until Process A reaches L1 and signals that B can proceed by executing *V(proceed).*
- What value must the semaphore *proceed* be initialised to?

Using Semaphores: Communication

Producer - Consumer problem – important example



Three **semaphores for** ***three*** **"resources":**

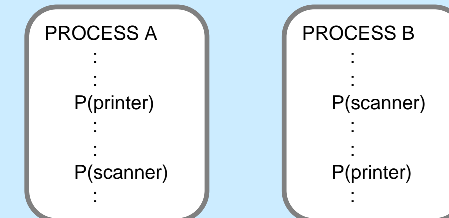
- ***Space*** **in buffer is resource needed by Producer**
  - **allow deposit only when buffer not full (items in buffer < N)**
- ***Item*** **in buffer is resource needed by Consumer**
  - **allow fetch only when buffer not empty (items in buffer > 0)**
- **Mutual exclusion for buffer access is resource needed by everyone**
  - **allow buffer access only when no one else accessing it**

Semaphore Solution

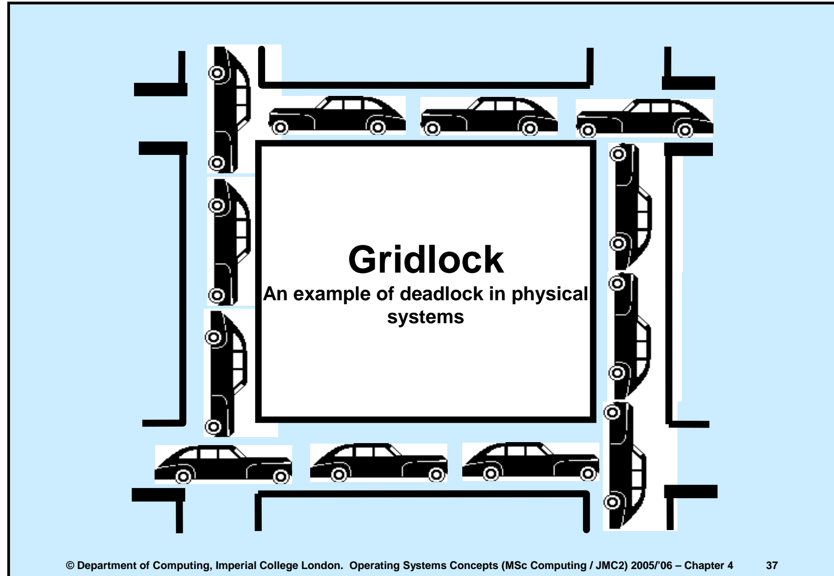
```
var mutex: semaphore   // initialise to 1
var space: semaphore   // initialise to N
var item: semaphore    // initialise to 0

process Producer
  loop
    – produce item
    P(space)  // "I want space"
    P(mutex)  // "I want mutual exclusion"
    – deposit item
    V(mutex)  // "Here is mutual exclusion"
    V(item)   // "Here is item"
  end loop
end Producer

process Consumer
  loop
    P(item)   // "I want item"
    P(mutex)  // "I want mutual exclusion"
    – fetch item
    V(mutex)  // "Here is mutual exclusion"
    V(space)  // "Here is space"
    – consume item
  end loop
end Consumer
```

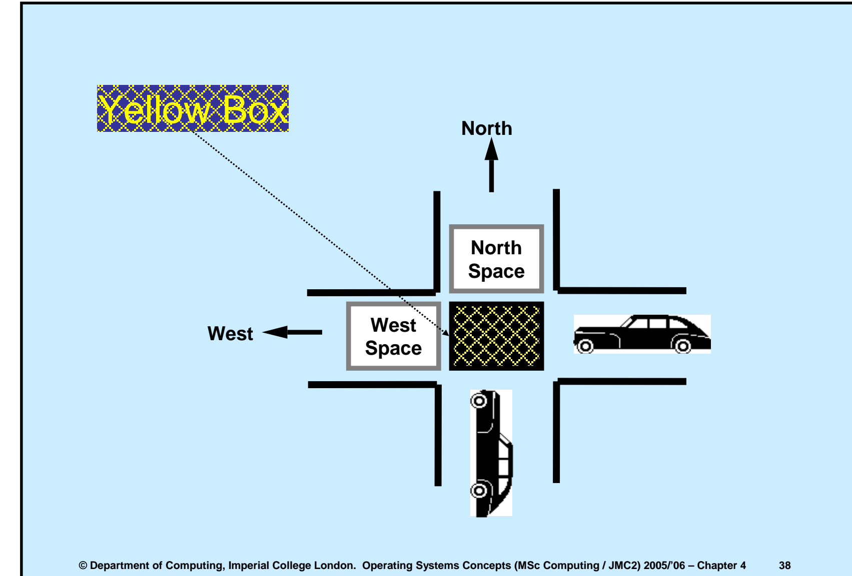
- **Solution still works for multiple Producer and Consumer processes.**
- **When** ***space*** **= 0 Producers cannot deposit items.**
- **When** ***item*** **= 0 Consumers cannot fetch items.**
- **What happens if we reverse the order of P operations in the Consumer?**

Deadlock



- ***Scanner*** **and** ***printer*** **are semaphores controlling access to the scanner and printer resources respectively.**
- **Initially** ***scanner*** **and** ***printer*** **have the value 1, i.e. resources free.**
- **If process A executes P(printer) and process B executes P(scanner), the system can make no further progress since each process will be suspended waiting for a resource (P-operation) held by the other.**
- **This condition is known as** ***DEADLOCK*** **and can occur where processes compete for resources.**





Semaphore Solution

```
var     Ybox : Semaphore    // Initialise to  1
var     Nspace: Semaphore   // Initialise to  1
var     Wspace: Semaphore   // Initialise to  1
```

process GoNorth

```
        P(Nspace)
        P(Ybox)
        Cross Junction
        V(Ybox)
        :
        V(Nspace)
```

end GoNorth

process GoWest

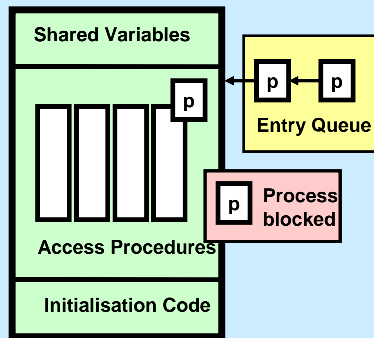
```
        P(Wspace)
        P(Ybox)
        Cross Junction
        V(Ybox)
        :
        V(Wspace)
```

end GoWest

Semaphores - Summary

- A semaphore is a **protected variable**
- A non-negative number – usually **accounts for resource availability**
- Binary semaphore (either 0 or 1) is exactly the **same as a lock** in terms of synchronisation
- Although you could implement a semaphore using a busy wait, the usual definition **requires non-busy waiting**
- Each semaphore has a **queue of processes** waiting for it; the V operation selects a process from the queue to allow access
- Semaphore is a low-level primitive, can be used to implement **mutual exclusion**, **synchronisation**, **communication**
- Like all synchronisation mechanisms, there is a risk of **deadlock** – when waiting processes form a dependence cycle

Process Interaction Mechanism #3: Monitors

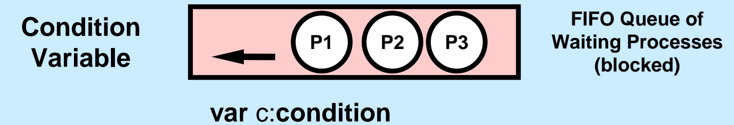


(Hoare C.A.R., Comm. ACM. 17, pp549-57, 1974)

- A monitor is a programming language construct which encapsulates:
  - VARIABLES
  - ACCESS PROCEDURES
  - initialisation code
- Access to the data encapsulated by the monitor is only possible through its *access* procedures
- Monitor =
  Abstract Data Type
  \+
  Only one process can be executing inside the monitor at any one time.
- Access procedures are *critical sections*
- Hence mutual exclusion becomes a *high-level programming primitive*

Monitor Synchronisation Primitives

Condition Variable



FIFO Queue of Waiting Processes (blocked)

var c:condition

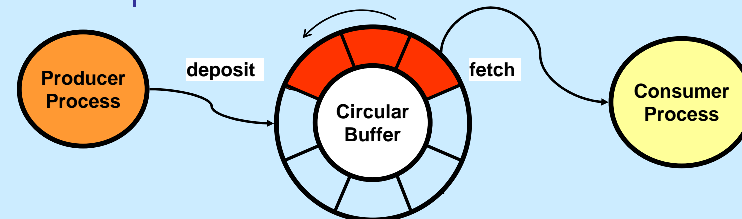
wait(c)	signal(c)
Suspend execution of calling proces Put it on condition queue c.	Resume execution of process at the head of condition queue c.

- Only one process may be executing inside a monitor at a time …
- But a *wait* operation (which blocks a process inside the monitor) will allow another process to enter and execute
- Waiting processes effectively exits the monitor temporarily.

Differences between *conditions* and *semaphores*

- Condition variable has no value associated.
- Wait always causes a process to be suspended
  - P operation sometimes does not (i.e. when s>0)
- Signal has an effect only if there is a process suspended on the signalled condition.

Example: Circular Buffer




```
type  buffer: monitor
  const N: int := 8
  var B: array 0..N-1  of item  // space for N items
  var    nextin, nextout: 0..N-1 // buffer pointers
  var    count: 0..N                // number of items in buffer
  var    nonfull, nonempty: condition
```

Buffer Monitor continued

```
procedure deposit(x:item)
    if count= N then
        wait(nonfull)
    end if
    B(nextin) := x
    nextin := (nextin + 1) mod N
    count := count + 1
    signal(nonempty)   //count > 0
end deposit

procedure fetch(var x:item)
    if count= 0 then
        wait(nonempty)
    end if
    x := B(nextout)
    nextout := (nextout + 1) mod N
    count := count - 1
    signal(nonfull)      //count < N
end fetch

/* initialization */
    nextin := 0
    nextout:=0
    count:=0
end monitor
```

Using the monitor:

```
var charbuff:buffer

{producer call}            {consumer call}
charbuff.deposit('X')      charbuff.fetch(ch)
```


Exercise: Implementing Semaphores using a Monitor

```
type semaphore: monitor
var i: int
var Q: condition

procedure P
    i := i-1
    if i < 0 then
        WAIT(Q)
    end if
end P

procedure V
    i := i+1
    if i ≤ 0 then
        SIGNAL(Q)
    end if
end V

end monitor
```

semaphores	
P(s) ::=	**when** s > 0 **do** s:=s-1
V(s) ::=	s:=s+1

Monitors - Summary

- The need for synchronisation between processes arises when they share a resource or data structure
- A monitor encapsulates the resource or data structure, and enforces mutual exclusion on all the access methods
- A process may block *within* an access method – it may wait on a condition variable:
  - When this happens, another process is allowed to enter the monitor
  - When a process signals a condition variable, the monitor selects the first process on that condition variable's queue to continue
  - The newly-awakened process must still wait for the first process to leave the monitor before it can re-enter

Concurrency: Summary

- Concurrency is a **major source of software unreliability**
- Undisciplined concurrent access to shared data leads to **inconsistency**
- Mutual exclusion is the fundamental technique to ensure that the system behaviour is the **result of some serial interleaving of logical operations** on the shared data
- To control complexity, systems must have **higher-level structure**
- **Semaphores** provide a simple building-block
- **Monitors** combine concurrency control with data encapsulation
- **Deadlock** results from a cycle of processes, each waiting for the next
- Concurrent systems need **careful design** and **validation**, and are extremely difficult to validate by testing

Concurrency: in real life

- There is no magic bullet
- There are automatic tools to help with validation (see Kramer and Magee's book)
- Each operating system offers a different menu of concurrency control primitives
- E.g. in Windows there are
  - Mutex – non-busy waiting lock
  - Semaphore
  - Event
  - Waitable timer
  - Messages
- In distributed systems and databases, great care is needed to ensure consistency – concurrency is an issue, so is **failure.**
  - A key idea is to structure computation as **transactions**, which can either succeed fully, or fail fully, but cannot lead to an externally-visible intermediate state