

RECAP

- Why OS?

- **Virtualises** the Machine
- 2 reasons WHY?
- restricted resources (utilisation, performance)
- abstraction = ease of use/programming
eg keyboard buffer allows typing even when cpu can't cope with input
- Processes and concurrency
- many processes **SHARE** the one CPU

- Later we see how memory is extended!

Chapter 4 – process and concurrency

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

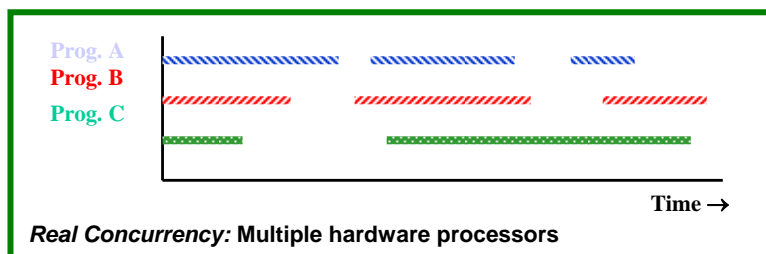
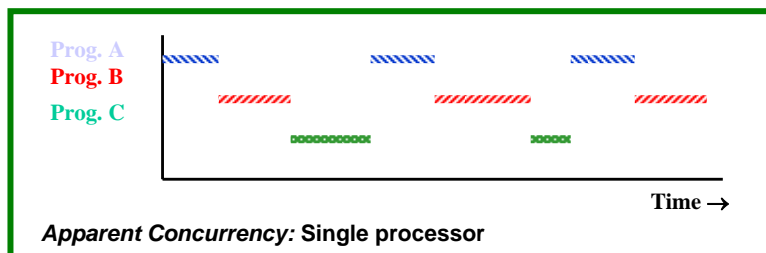
Processes cont

- **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.

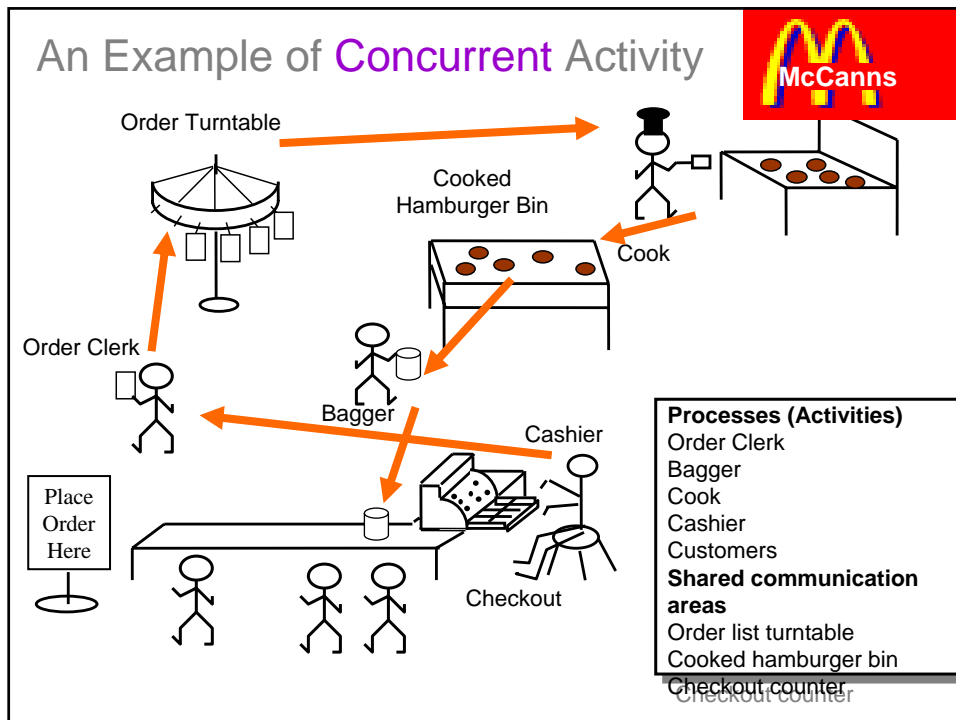
Concurrency and processes

- 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”

Concurrent Processes



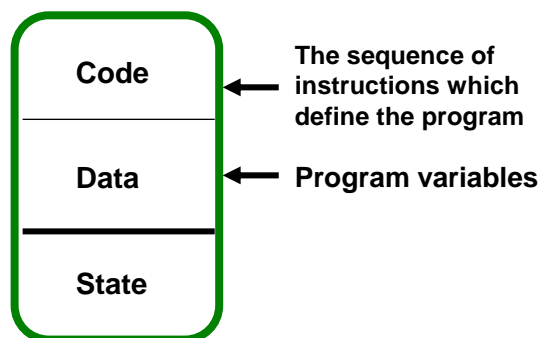
Each activation of a program is termed a **Process**



Program = a sequence of instructions

Process = an activity consisting of the execution of a program

A Process is represented in a computer by:



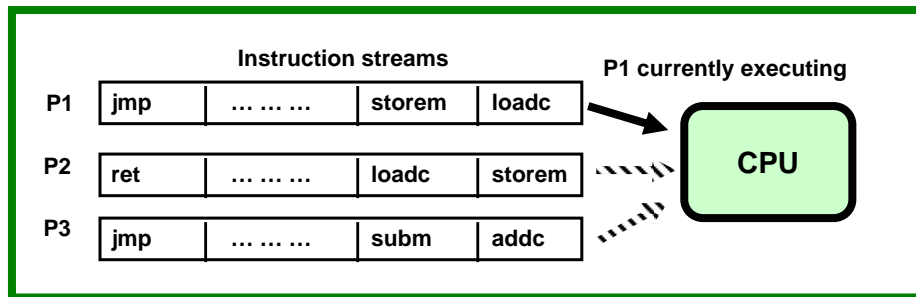
Processor:

- 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 I Interrupt handler

```

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

```

Recall...

```

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

```

```

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

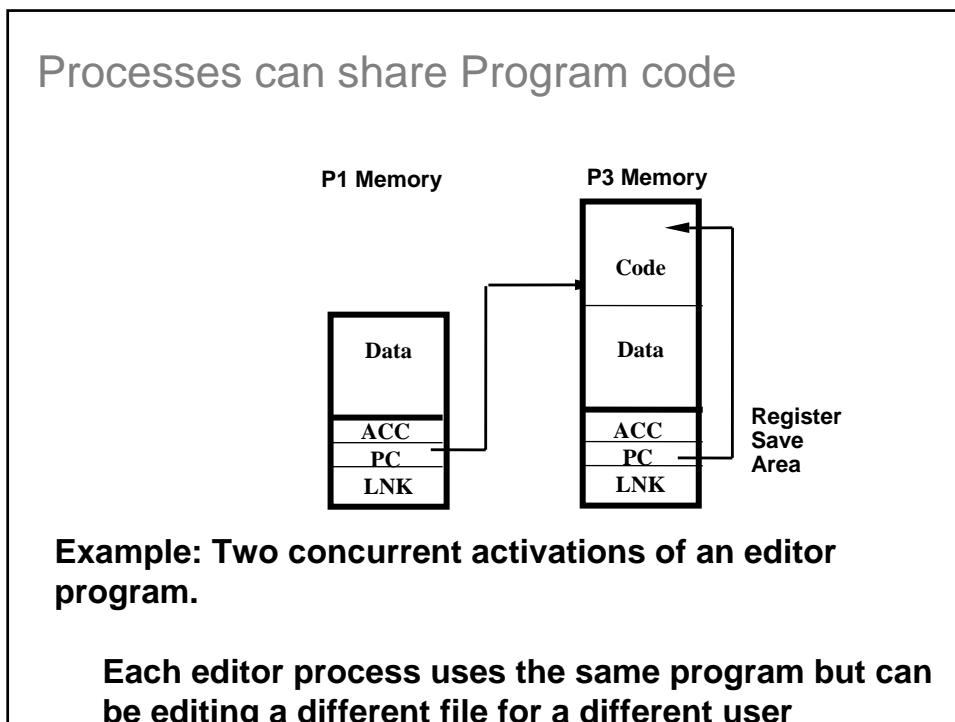
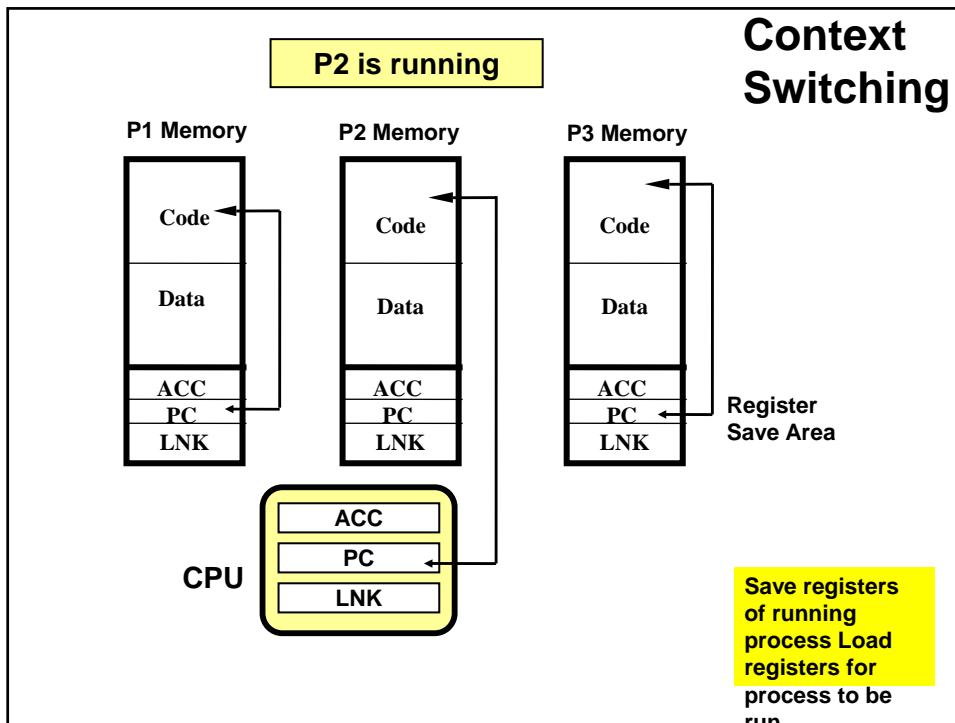
```

Switching between
processes

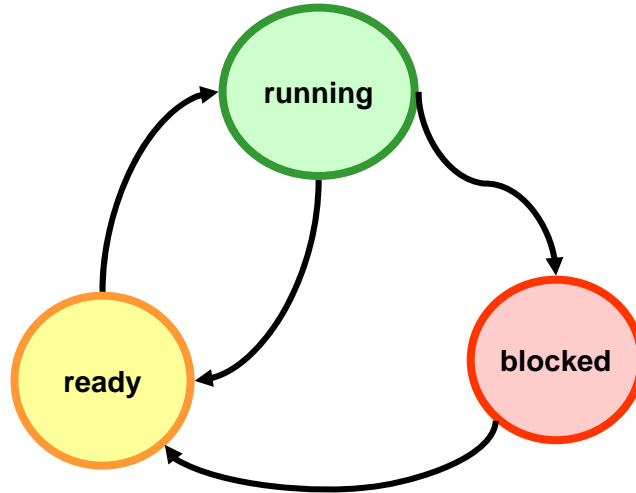
```

void InterruptHandler() {
  saveProcessorState();
  1. Handle the interrupt
  2. Choose which processor state
     to return to
  restoreProcessorState();
  rti; // restore PC from Mem[0] and
      // re-enable interrupts
}

```



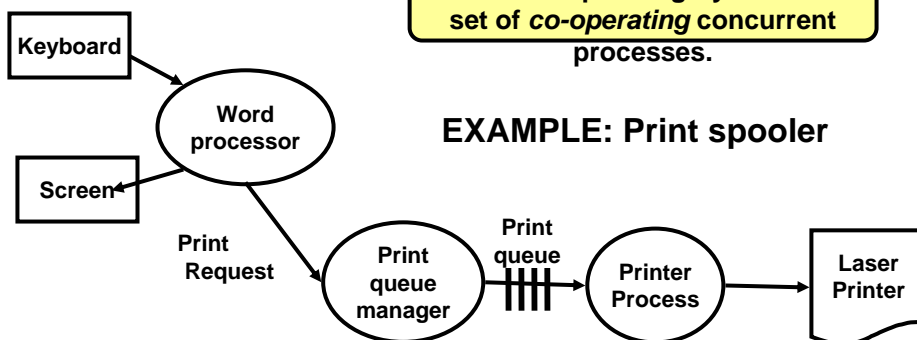
Process State Transitions



Processes are an important concept in OS structuring.

Consider Operating System as a set of *co-operating concurrent* processes.

EXAMPLE: Print spooler



Word processor:

User prepares document, requests printing

Print queue manager:

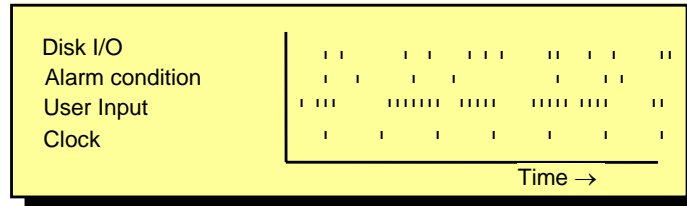
Maintains queue of jobs for printer. If queue was previously empty, starts printer process.

Printer Process:

Translates document to printer commands, and sends them to it. On completion, removes job from queue, and repeats. Terminates queue is empty.

Non- Determinism

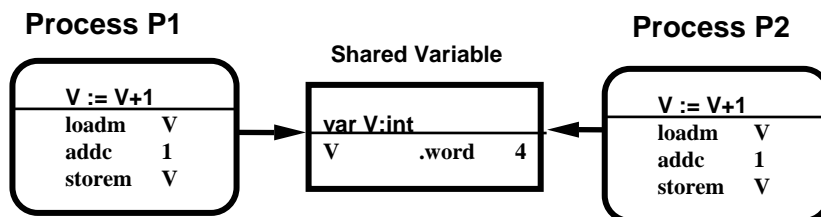
- Operating Systems are *non-deterministic* in that they must respond to events (I/O) which occur in an unpredictable order.
 - Events (or interrupts) **cause process switches**.
 - e.g. an I/O completion interrupt will cause the OS to switch to an I/O process.



- 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)



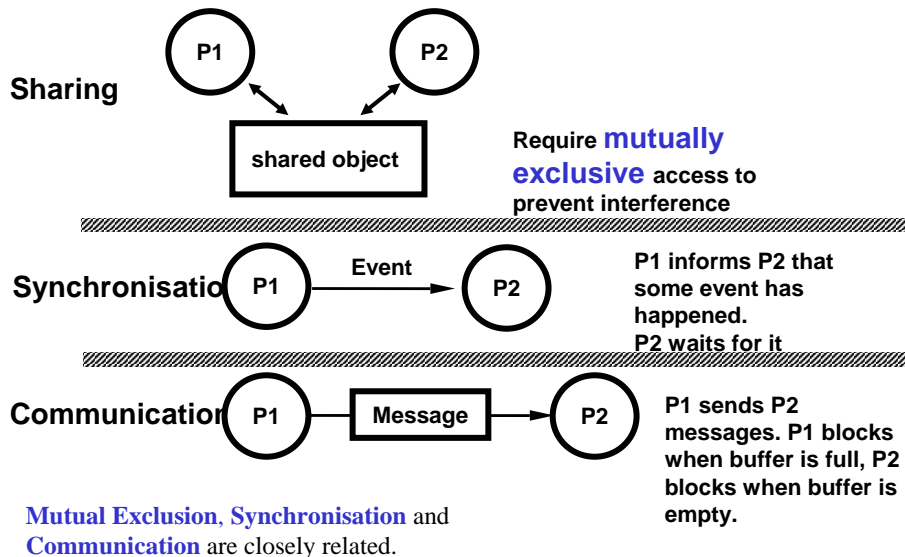
Process	Instruction	Accumulator	Value of V
P1	loadm V	4	4
P2	loadm V	4	4
P1	addc 1	5	4
P2	addc 1	5	4
P1	storem V	5	5
P2	storem V	5	5

Race condition

Remember instructions can be arbitrarily interleaved

PROCESS INTERACTION MUST BE CONTROLLED

Types of Process Interaction



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

- $v = v + 1$



```
loadm V  
addc 1  
storem V
```

- If (lock is free) claim it



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

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

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 } "critical section"
    V = V+1;
    UNLOCK(L)
}
```

- 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

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

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

```
TEST    checks L is set
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	Value of L
P1	TEST L	0
P2	TEST L	0
P2	MOV #1,L	1
P1	MOV#1,L	1

A better implementation of Locks

You might try to implement locks like

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

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

interrupts

- 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

How can we use this instruction?

**LOCK: TS L
BNZ LOCK**

UNLOCK: MOV #0, L

- It only sets the 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)**

Semaphores

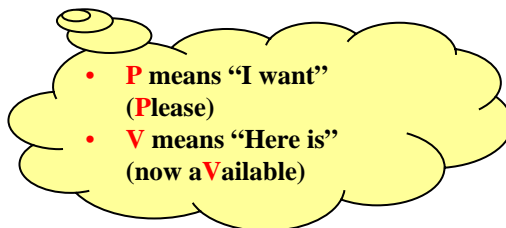
Process Interaction Mechanism #2

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:

$P(s) ::= \text{when } s > 0 \text{ do } s = s - 1$
 $V(s) ::= s = s + 1$



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

- 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 initialisation, $\text{init}(s) = 1$

Using Semaphores: Mutual Exclusion

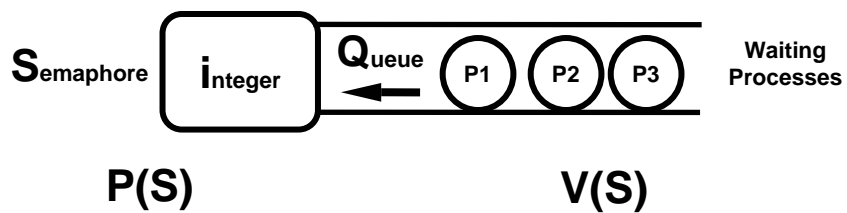
```

var d: int           //shared variable
var s: semaphore
Init Sema(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*.

Non busy wait implementation of semaphores



```

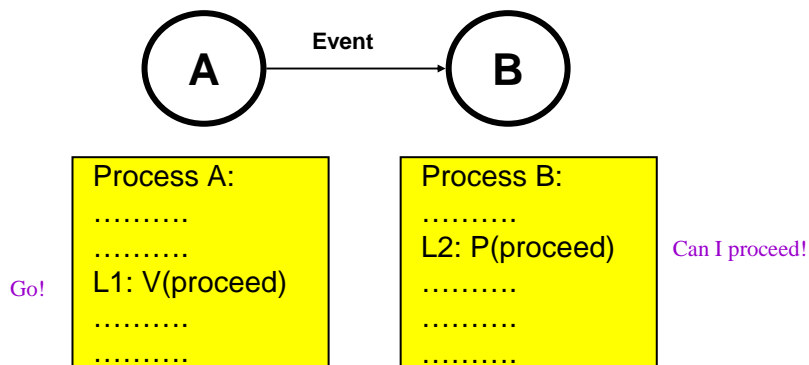
if S.i > 0 then
    S.i := S.i-1
else
    suspend process on S.Q
end if
    
```

```

    S.i := S.i + 1
    if not empty(S.Q) then
        resume a process in S.Q
    end if
    
```

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

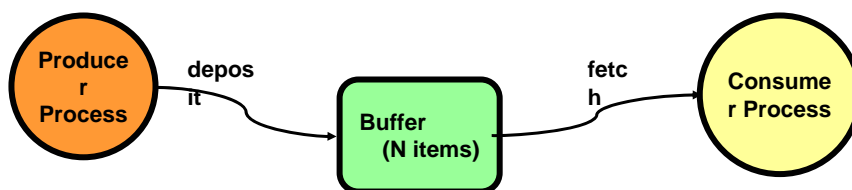
Using Semaphores: Synchronisation



- Process B must wait at L2
- until Process A reaches L1
- and signals that B can proceed by executing `V(proceed)`.

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 **MUTEX**

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

PROCESS A

```

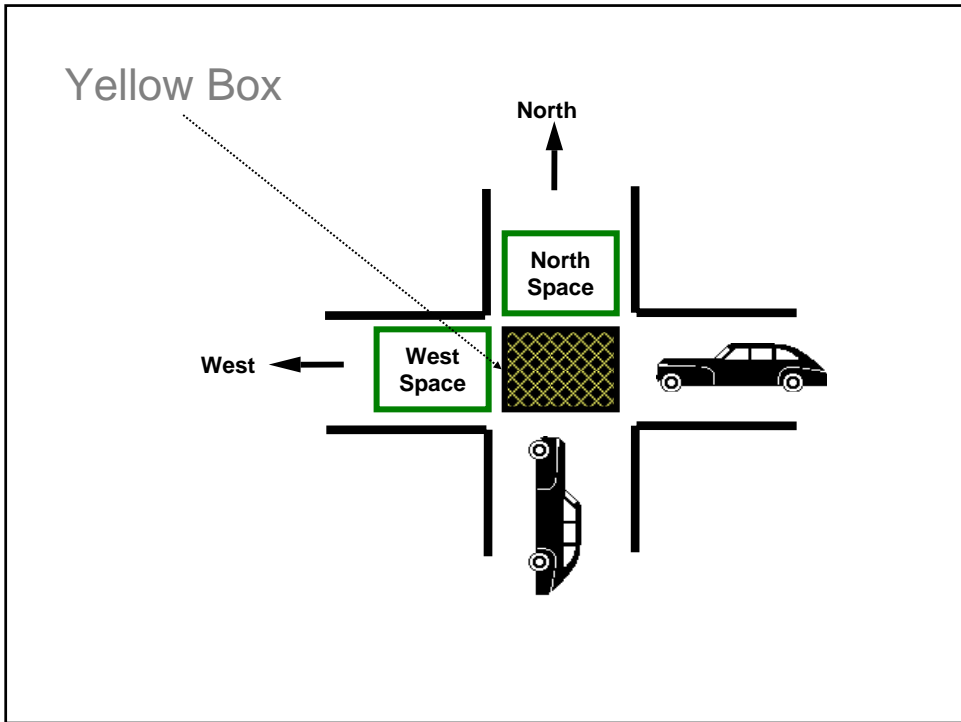
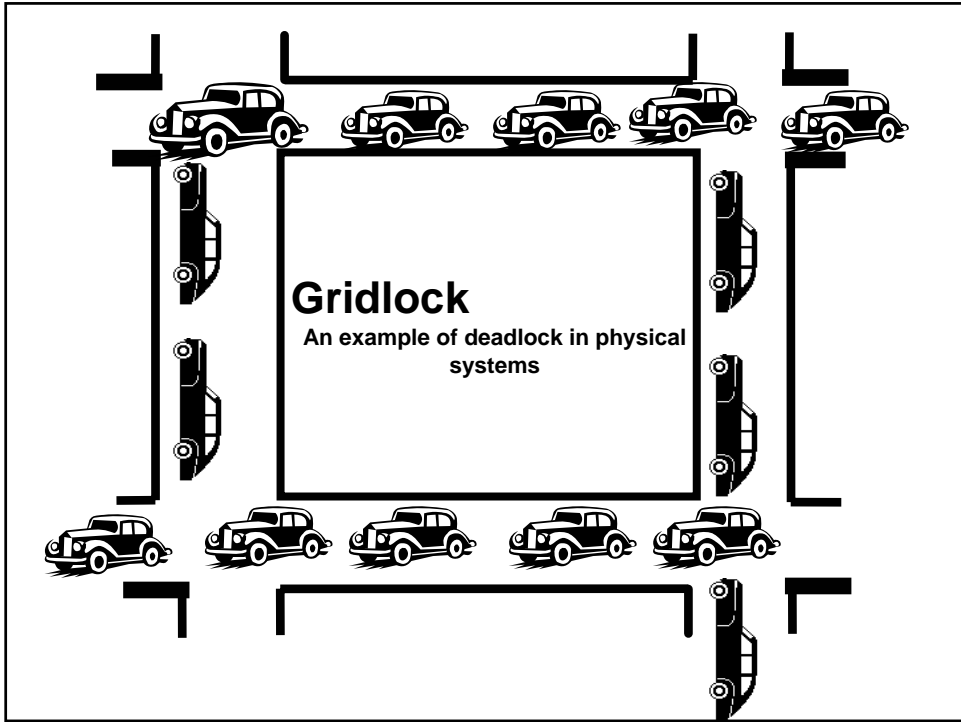
:
P(printer)
:
:
P(scanner)
:
    
```

PROCESS B

```

:
P(scanner)
:
:
P(printer)
:
    
```

- *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
    Cross Junction

    P(Nspace)
    P(Ybox)
    V(Ybox)
    :
    V(Nspace)

end GoNorth

process GoWest
    Cross Junction

    P(Wspace)
    P(Ybox)
    V(Ybox)
    :
    V(Wspace)

end GoWest
```

Semaphores - summary

- A semaphore is a **protected variable**
- A **non-negative** number – usually **accounts for resource available**
- Binary semaphore (either 0 or 1) is exactly the **same as a lock**
- 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

So far

- Mutex, Sync and Comm implemented by:

Lock

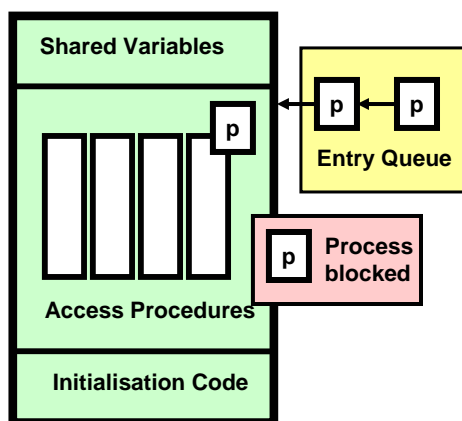
Semaphore

now

monitors

Monitors

Process Interaction Mechanism #3:



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

- A monitor is a programming language construct which encapsulates:
 - VARIABLES
 - ACCESS PROCEDURES
 - initialisation code

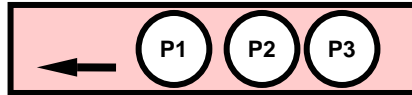
• Access to the data encapsulated by the monitor is only possible through its *access* procedures

• 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 process
Put it on condition queue c.

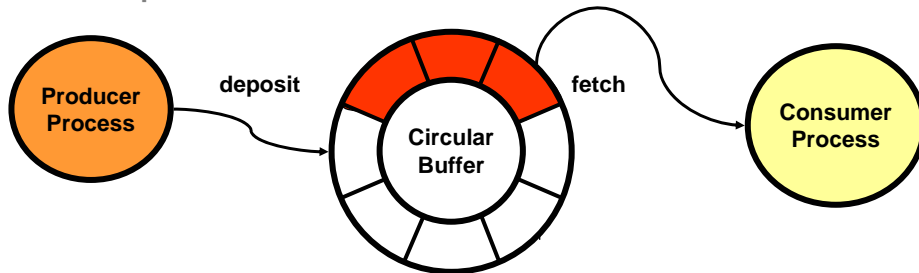
Resume execution of process at the
head of condition queue c.

- 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 exit 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

```
begin monitor
```

```

...
/* initialization */
nextin := 0
nextout := 0
count := 0

```

```

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

```

```
end monitor
```

Using the monitor:

```
var charbuff:buffer
```

```

{producer call}
charbuff.deposit('X')

```

```

{consumer call}
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
end monitor
```

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

```
end V
```

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 are automatic tools to help with validation (see Kramer and Magee's work)
- Each operating system offers a different menu of concurrency control primitives
- E.g. in Windows NT 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

Recap

- Overture – nature of the OS, **Virtualizing** the Machine
- IO – how to build device driver, interrupt handling
- Processes&Concurrency
 - Nature of concurrency and its problems
 - **Stopping Processes running**: locking, semaphores, buffers, monitors
 - **When to run a process: Scheduling** (today)
- Mem Man!

Scheduling Processes

Scheduling

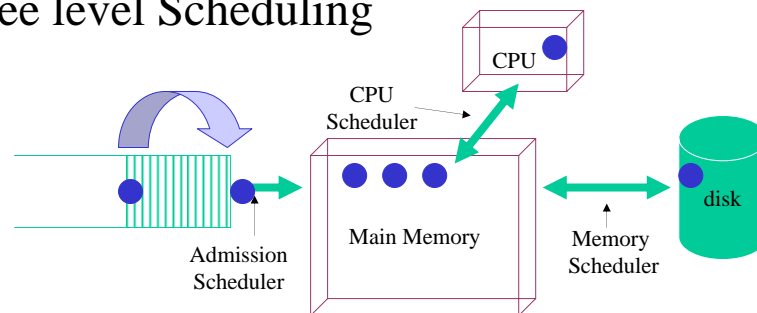
- Only one CPU → scheduler + scheduling algorithm
- PCs were very different from mainframes
 - Modern machines are limited by the rate of input rather than CPU → IO bound
 - High-end *networked workstations* and servers scheduling is v.important
- Must pick **right** process to run
 - Make efficient use of CPU, but bare **process switching** in mind
 - i.e. state saved – registers etc + memory map saved.
 - Load new process details into the MMU

- Scheduling happens when
 - **new** process created,
 - process **exits** or process **blocks** (IO/semaphore etc),
 - or blocked process **resumes** (after IO interrupt)
- **Non pre-emptive** – pick process to run until block or it releases CPU
- **Pre-emptive** –process runs for fixed amount of time (clock interrupt occurs to give control of CPU to scheduler)

Categories

- Goals: fairness, policy enforcement, balance
- Depends on the OS and environment (even system/user pt of view)
- Batch systems → throughput, turnaround time, CPU utilisation
 - Interactive → response time, proportionality (users expectations)
 - Real-time → meeting deadlines, predictability
- Batch algorithms
 - **first-come-first-served** – next on queue is served (simple) (FIFO)
 - **Shortest-Job-first** – well known times, only good if 0 interarrival time
 - **Shortest-remaining-time** – for pre-emptive jobs (new short jobs get good service)

Three level Scheduling



- Jobs arrive at system, placed in input queue stored on disk
- **Admission scheduler** decided which to admit to system
- Job enters system and process created -- competes for CPU
- If memory can't hold processes – swapped to disk
- Memory scheduler determines which kept in memory which on disk
 - Remember swapping to and from disk costs (disk IO bw goes down)
 - Degree of multi-programming = no processes it want in mem
- CPU scheduler decides on which ready process to run

Interactive System Scheduling

- Can be used by the CPU scheduler in batch sys too!

Round-Robin

- Process assigned time interval (**quantum**)
- If process is running at end of quantum then
 - CPU pre-empted and given to another process
- Simple algorithm
 - list of runnable process,
 - end of quantum process moves to end of list.
- Length of quantum affects performance, Why?
 - E.g. context switch = 1msec, quantum = 4 msec → CPU spend 20% on admin
 - Quantum = 100 msec = ? %

Round-Robin cont.

- Performance cont.
 - If 10 users hit <return> same time
 - → 10 processes on queue
 - CPU idle therefore starts first one
 - Next don't get CPU until 100 msec later! Last key = 1sec!!! 1
 - 1 second response time for <return> **NO THANKYOU!**
 - Good to have quantum > mean CPU burst → pre-emption rarely will happen
 - Why is this good?
 - *Recommendation is around 20-50 msec [Tanenbaum]*

Priority Scheduling

*all processes are **equal** but some are **more equal** than others*

- RR= all processes are equal
- Pecking order (deans, profs, SLs, janitors and then students ;-)
- Assigned a **priority** and highest runs
- Even PC (single user) multiple processes e.g. mail daemon vs. video player
- Prevent greedy processes by reducing their priority at clock ticks

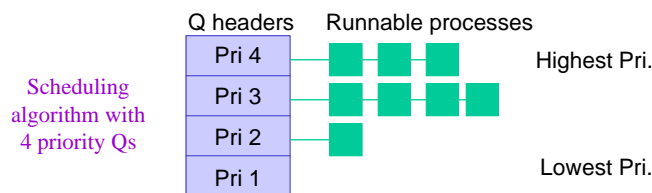
Priority Scheduling cont.

- Can assign **costs** to priority.
- E.g. my fab research job on a supercomputer gets 100 and I pay for it, whereas student compile gets 10 for free! ;-)
 – **Nice** → user reduces priority to be *real nice* to others
 – (ask about the dept, has anyone ever ever used it?)
- **Dynamic allocation**
 - IO bound job gets higher CPU priority, why? →
 - During IO, the Io-Process waits until finished
 - Gets CPU immediately so it can get out of the way
 - Out of the way = out of memory and let CPU bound on for longer in parallel
- Combine RR and Priority (RR within a class of application)
- Beware of **Starvation**

Multiple Queues

- Large quantum = poor response time
- Processes divided into **priority classes**
- Processes in highest class run for 1 quantum
 – Process in next highest run for 2 quanta next 4 etc
 – Whenever process used its quanta it moved down a class
 – Eg.
 - Process requires 100 quanta
 - Gets 1 quanta then swaps out, then 2 quanta, 4, 8 16, 32, 64 (37)
 - 7 Swaps in total, how many for RR?

100



Policy Vs Mechanism

- Large apps like DBMS have many child processes
 - DBMS knows how to schedule better than OS
 - Therefore is OS separates mechanism from policy DBMS can communicate (via parameters) to OS
- **Threads**
 - **User level** - use RR and Pri to schedule them
 - won't have interrupt for long threads hogging time, context switch is lightweight
 - Kernel – context switch **is full state save** (heavy/slow)

Summary

- Scheduling => fairness
 - User/system point of view
 - Different mechanisms
 - Showed: batch/Interactive algorithms -> policy
 - Others: real-time, multimedia etc.