Note: Dispatcher can choose to run each thread to completion, or time-slice in big chunks, or time slice so that each thread executes only one instruction at a time (simulating a multiprocessor, where each CPU operates in lockstep).

If this is possible, programs must work under all cases, for all interleavings!!!. How can you know whether all interleavings will work ? and more importantly, how do you know if your concurrent program works ?

1 The story so far ...

- What operating systems perform
- Why operating systems are needed
- Types of operating systems (Overview)
- The concept and types of concurrency
- The **core** of any OS : **The Kernel**
- Types of kernels - which determine OS model
- Process model
- Threads and Address space
- Uniprogramming, Multiprogramming, Multitasking, Multiprocessing
- Kernel functions

Labs today

Basic introduction to Unix, and identifying processes

A question that occured during the tutorial ?
Is windows 95 a new OS or just an extension to DOS ?

Objective : End of the week Objective :

Get a grip on **processes** and **kernels**

2 Scheduling - A Nightmare!!! on CPU street

- Not that different ... supermarkets, your tutorials and labs, TV programs, aircrafts and trains etc
- Multiple processes running on a machine, competing for resources
- Usually the number of processes are much more than processors
- It is an NP hard problem - i.e. *optimal* allocation *cannot* be guaranteed - however, a set of heuristics possible
- The scheduler (just like your lab and tutorial scheduling) is trying to satisfy a number of different (often conflicting) criteria
- Processes are unique and unpredictable (file I/O, computation etc)

Types :

- Preemptive vs Non-preemptive : Is the execution of a process(es) interruptible
- Static vs Dynamic : Is scheduling done once, or adjusted during execution
- Priority (static or dynamic) assignment ... numerous methods

What criteria is the scheduler balancing ?

- **Fairness** : Share CPU among various users in some equitable way
- **Efficiency** : Keep the CPU(s) as busy as possible - ideally 100
- **Response Time** : Minimise response time for interactive users
- **Turnaround** : Minimise the time to obtain an output from the system
- **Throughput** : Maximise the number of jobs processed per a unit of time
- **Predictable** : Minimise load variation effect on execution
- **Priorities** : In addition to all of the above, also take process priorities into consideration
- ... and others - can you think of any ?

There are three - so called - *scheduling levels* :

- **High-level scheduling** : job scheduling or program scheduling - each made of *groups* of processes
- **Intermediate level scheduling** : Which processes should compete for the CPU. Meeting some overall *systemwide objectives* - over a longer term
- **Low level scheduling** : which *ready* processes will be assigned the CPU, achieves the *dispatching* of processes to CPU. Performed by the *dispatcher*.

Scheduling assumptions : Lots of them !!

Bunch of algorithms for CPU scheduling – big area of research in the 70s. Assume :

- One program per user
- one process (thread) per program
- programs are independent

Unrealistic!!!! - however need to simply to solve.

3 Low Level Scheduling Methods

Natural Completion

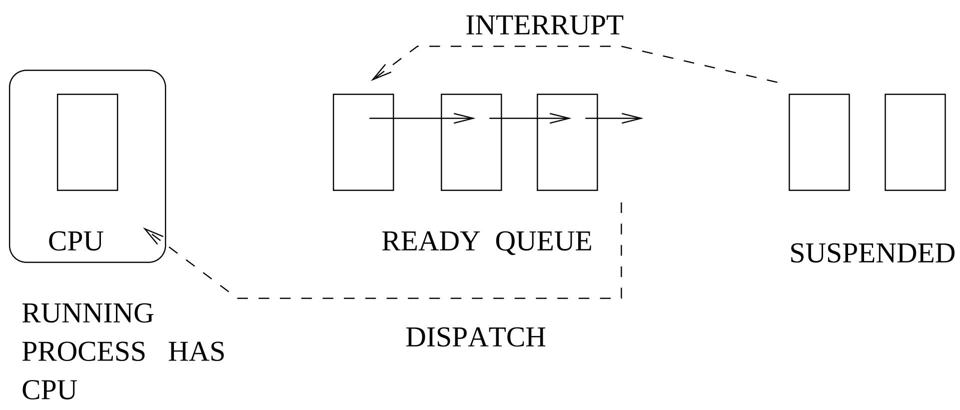

Figure 1:

- Process runs until it either completes or suspends itself by : making a kernel call, cause an error trap or completes and leaves system (unix zombies)
- The runnable process is placed at the end of the *ready* queue.

Preemption

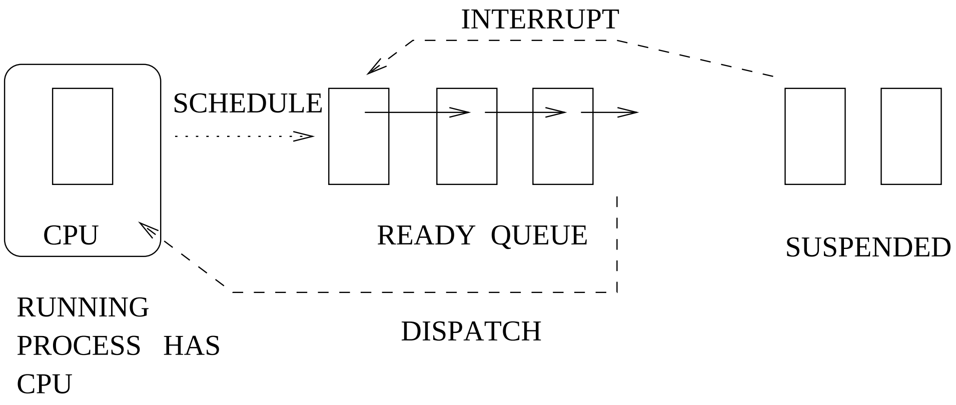

Figure 2:

- Interrupt causes suspended process to become runnable depending on priority
- Insert in ready queue in *priority order* (i.e. at head of queue)
- Dispatcher picks the new high priority process first

Simpler ?

- **FIFO** : First in first out (usually meant one program kept CPU until it completely finished - uniprogramming) - aka FCFS (first come first serve)
  \+ *simple*
  \- *short jobs get stuck behind long ones*
- Solution ? - Add a timer, and preempt CPU from long-running processes (or jobs). Just about every real OS does something of this flavour.

Time Slice with Quantum size

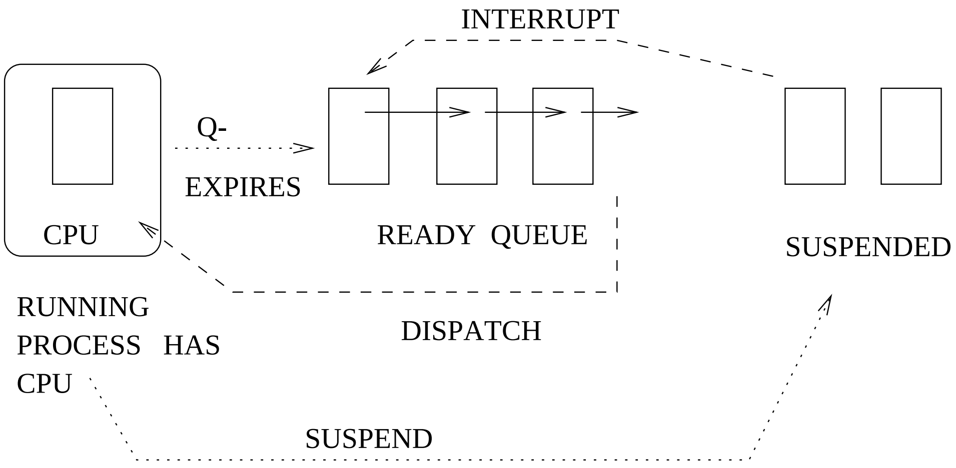

Figure 3:

- Process runs until time slice (quantum) expires
- aka *round robin scheduling*
- What should be the size of the quantum ? (too large ≡ FIFO, too small ≡ too many context switches)
- Optimal is a design parameter
- Effective in time sharing environments (reasonable response time for interactive users)
- *selfish round robin* : processes wait in a *holding queue* until their priorities reach active queue priorities

Typical time slices : 10-100 millisecs; typical o/head : 0.1 - 1 millisec
Overhead = 1%

STCF and SRTCF

- STCF : Shortest time to completion first
- SRTCF : Shortest remaining time to completion first. Preemptive version of STCF, if job with shorter time to completion arrives, give CPU to new job
- Idea : get short jobs out of system (cf answering exam questions?) - Big effort on short jobs, small effect on big ones. *Improved* **average** *performance*

The crystal ball

- STCF/SRTCF : require knowledge of future
- *How do you know how long program will run for ?*
- Ask user ? What if job takes more ? Kill job ? - Bankers algorithm
- *Predicting response of system based on past : very common occurance in computing*
- Program behaviour is regular, most of the time - Exploit this ! (Locality)

Multilevel feedback

- **adaptive** : change scheduling policy (method) based on past behaviour
- Multiple queues, each with different priority
- OS does round robin at each priority level - run highest priority job first - on completion the next priority job etc
- Round robin time slice increases exponentially at lower priorities
- Priority adjustment is as follows :
  Job starts in highest priority queue
  If timeout expires, drop one level
  If timeout does not expire, push up one level (or back to top)
- Result approximates SRTCF : *CPU bound processes drop like a rock, others stay near the top*
- Still unfair : *long running jobs may never get CPU*
- *countermeasure* : put in meaningless I/O to keep job priority high. Cheat!

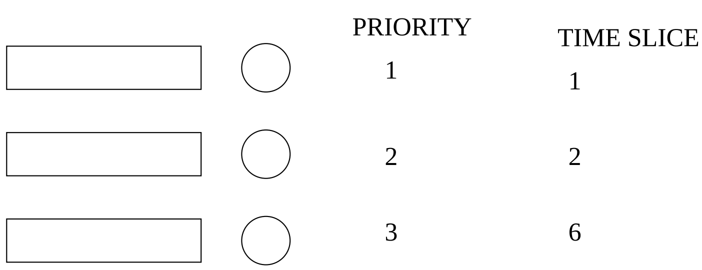

Figure 4:

The Mid-Week draw

- Handling *fairness problems* with above - would it hurt average response time ?
- Each queue is given a fraction of the CPU - what if 100 short running jobs but one long one
- Could adjust priorities : increase *priority* of jobs, as they *don't* get service - as in Unix
- Ad hoc! - as system load goes up - no job gets CPU - so everyone increases priority. Interactive jobs suffer
- *Answer* : **Lottery Scheduling** : every job has a number of lottery tickets, and randomly picks a winning ticket at each time slice. On average, CPU time is proportional to number of tickets given to each job
- Tickets are assigned as SRTCF : long ones get few - short ones get many. To avoid **starvation** : every job gets at least one - hence everyone can make progress
- Major advantage : graceful behaviour with changing loads. Adding or deleting a job affects everyone else proportionately - but independent of how many tickets each job gets

Scheduling on Unix

- Assigns *base priorities* (a high of -20 to a low of +20, with a median of 0)
- Priorities are *adjusted* according to changing system conditions
- *Current priorities* are calculated by adding to base values - process with highest priority is dispatched first
- Priority adjustment is *biased* in favour of processes that have recently used relatively little CPU

Scheduling on VAX/VMS

- *VAX/VMS* : process priorities range from 0-31, with 31 assigned to critical real-time processes. *Normal processes* priorities range from 0-15, and real time from 16 to 31
- Constant priorities for real-time processes (run to completion or preemption by higher priority process)
- Normal process priorities do vary - their *base* priorities remain fixed, but dynamic priority adjustment to give preference to I/O-bound processes over CPU-bound processes
- Normal process retains the CPU until preempted, quantum expiry or a special state event wait, with priority adjustment from 0 to 6 over their base level
- Processes with the same current priorities - scheduled as round robin