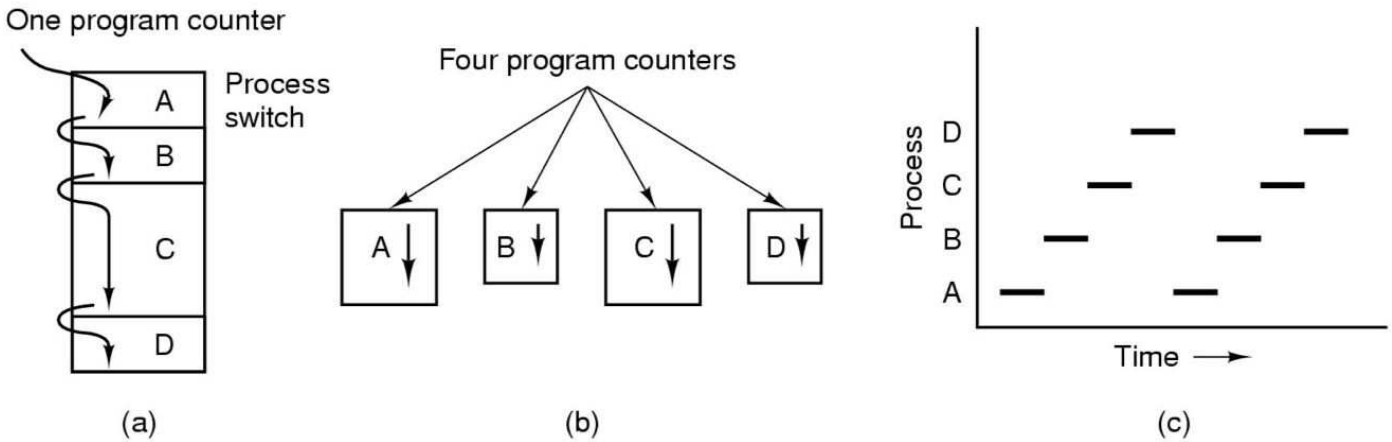
The University of Melbourne
School of Computing and Information Systems

COMP30023
Computer Systems

Semester 1, 2017

Process Scheduling

Processes



A conceptual model of 4 independent, sequential processes. Only one process is active at any instant.

Process states

A process can be in one of three states.

Running Actually using the CPU.

Ready Runnable; temporarily stopped to let another process run.

Blocked Unable to run until some external event happens.

Usually there are separate states for each reason for blocking, e.g. waiting for terminal I/O, waiting for disk I/O etc.

In a machine with N CPUs, up to N processes can be Running at once.

State transitions

Running to Blocked	Process requests service which cannot be provided at this time.
Running to Ready	Scheduler gives the CPU to another process.
Ready to Running	Scheduler gives a CPU to this process.
Ready to Blocked	This transition cannot happen.
Blocked to Ready	The requested service is now available.
Blocked to Running	The requested service is now available, and the scheduler assigns a CPU to this process.

Process table / Process control block

The saved states of processes are stored in the kernel, in the *process table*. This table has one slot for every process. Some typical fields in process table entries (there are more):

- process id (usually a number)
- process state (running, ready or blocked)
- user id and other privilege information
- start addresses and sizes of memory areas (code, data, stack)
- CPU time used and other accounting information
- priority (used by the scheduler)
- working directory
- list of open files, and info about them
- space for saved copies of registers (including PC, PSW and SP)

Process table / Process control block

The process table – sometimes referred to as a process control block (PCB) – provides a "snapshot" of the execution and protection environment.

```
enum state_type {new, ready, running, waiting, halted};

typedef struct_control_block
{
    struct control_block *next_pcb;
    enum state_type state;
    int pid;
    address PC;
    int reg_file[NumRegs];
    int priority;
    address page_table;
} control_block;
```

Creating a process

When creating a process there are many "issues" to consider:

- Allocate memory to the process
- Initialise the process control block
- Insert the process into a "ready queue" – mark as READY
- Estimate how long it will take to run (?)
- Analyse the memory requirements
- Load the first part of the process into memory

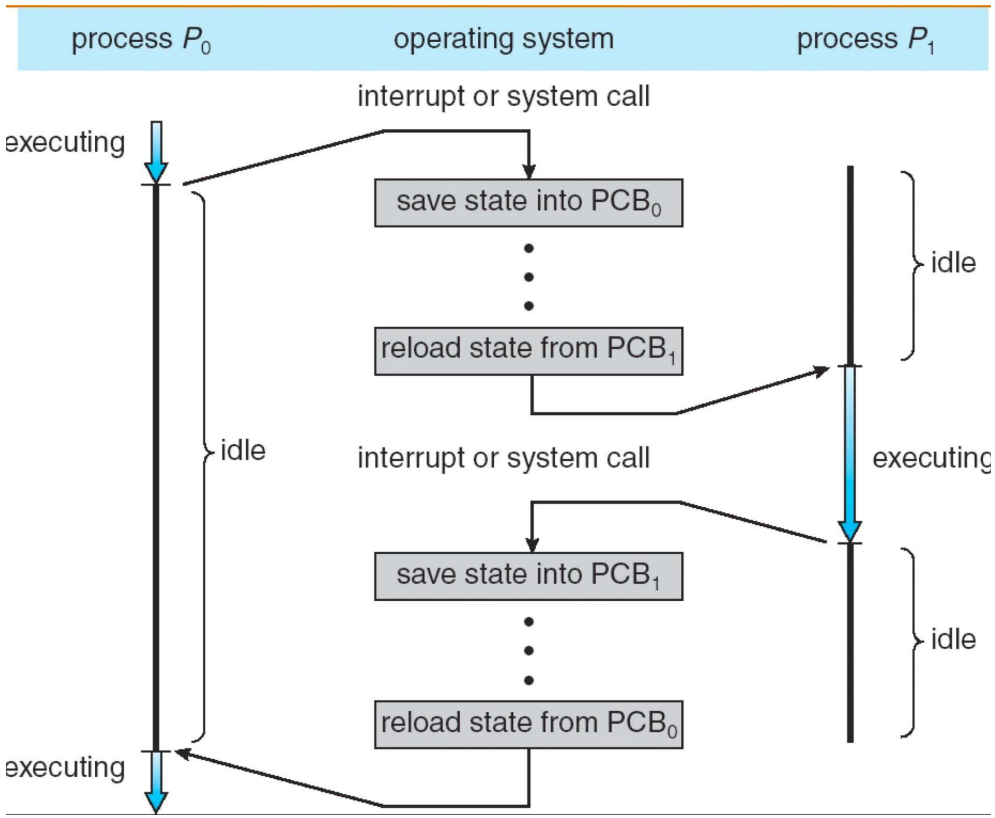
Scheduling at interrupts

The interrupt handler changes the state of the current thread to ready before servicing the interrupt. Afterwards, it finds the highest priority ready **thread**, and restores its saved state.

This thread may or may not be the one that was running before the interrupt. If it isn't, the event is a context switch, and we say the previously running thread was *preempted*.

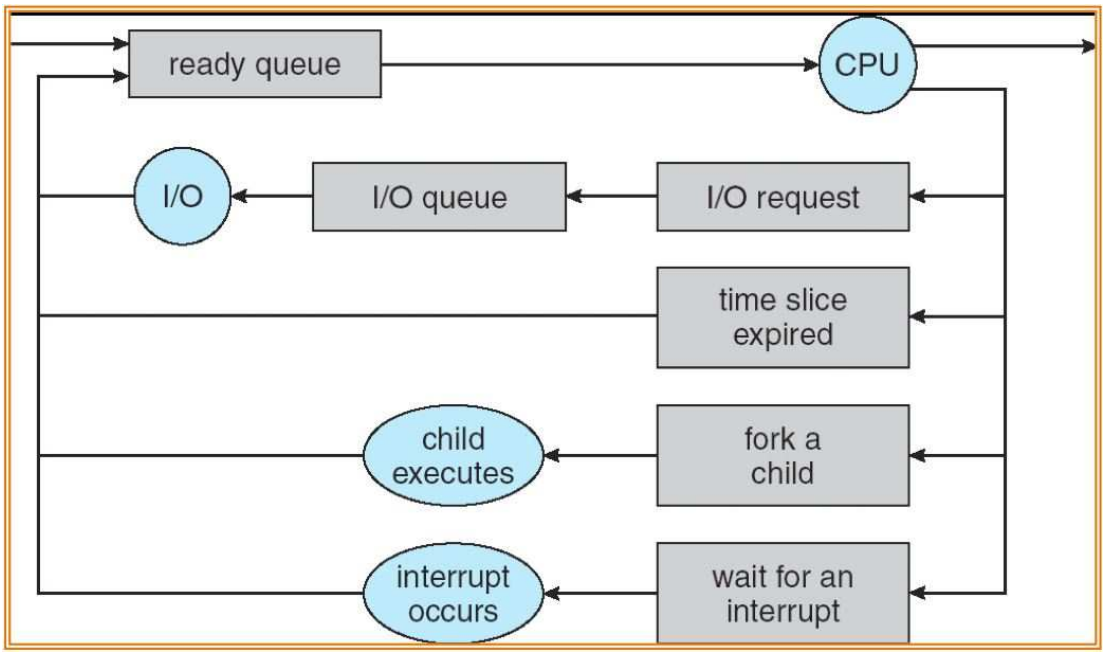
Processes and threads are usually not aware of interrupts or context switches.

Context switch: act of switching processes



Processing queues

Processes move from queue to queue as they change state. Scheduling decision have to made about which order to remove processes from queues.



Aims of scheduling

- **Fairness**: every process and/or thread should get its fair share of the CPU.
- **Throughput**: should maximize the number of jobs processed per unit time.
- **Response time**: should minimize waiting for interactive users.
- **Turnaround time**: should minimize waiting for batch jobs.

Throughput can only be maximized by reducing overhead, since each user job requires a certain minimum amount of computation.

Response time can only be minimized in multi-user systems by running each user's job whenever that user requires service.

However, the latter requires lots of context switches, which are a major source of overhead, and may lock out other threads unfairly.

First-come first-served

The simplest algorithm is first-come first-served, or FCFS, which is used only in the simplest of *batch* environments.

It performs reasonably well if all jobs require approximately the same time.

If not, it discriminates against short jobs by delaying their start until all long jobs before them in the queue have finished; the result is poor turnaround time.

Shortest job first

Suppose you have four jobs, A, B, C and D, and the time required by each job is:

A	B	C	D
16	8	4	2

Order ABCD: jobs finish at times 16, 24, 28 and 30; average = 24.5.

Order DCBA: jobs finish at times 2, 6, 14, 30; average = 13.

SJF has good turnaround time, but it may starve long jobs if the machine is heavily loaded.

"Eight item or less" queues in supermarkets are as close as supermarkets can come to SJF (since it is obviously not practical to predict the time needed for each "job" with any precision).

Round robin

The RR algorithm keeps a list of *ready* threads, and allocates to each thread a time interval or time slice, called a *quantum*, that it is allowed to run.

If a thread is still running at the end of its quantum, the CPU is given to another thread, and the old thread is put on the end of the ready queue.

Short quantum:	good response time, poor throughput; good for interactive systems.
Long quantum:	poor response time, good throughput; good for batch systems.

Selfish round robin

No scheduling algorithm can give good response time if there is too much demand for the CPU. It may be better to give good service to some jobs and none to others than to give uniformly bad service.

The SRR algorithm works like RR except that it does not put new jobs in the ready queue until the demand for the CPU is "acceptably low".

Depending on the exact meaning of that phrase, SRR can approximate either FCFS or RR.

Priority scheduling

This algorithm assumes that each thread has a priority. It always runs the highest priority ready thread.

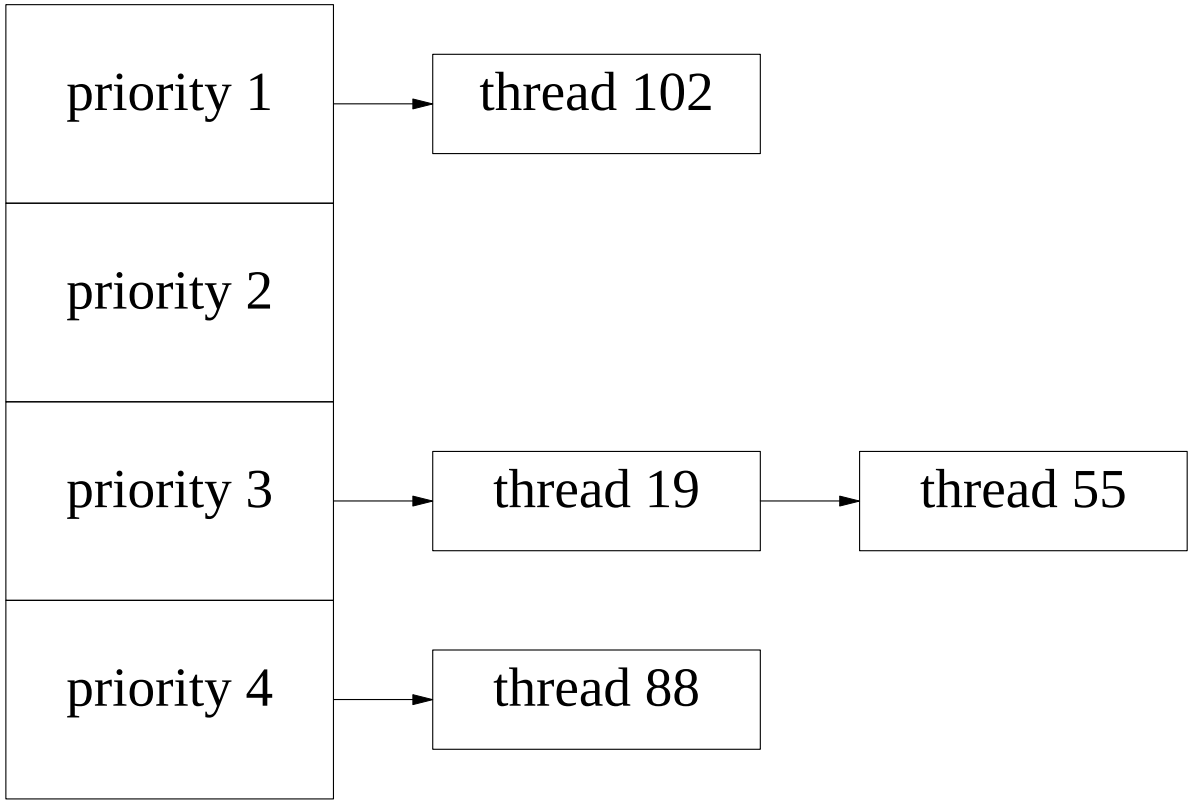
In pure priority systems, a running thread is never preempted; this is useful in some real-time applications.

In other systems, the threads of each priority level are scheduled in a round-robin fashion among themselves.

This is usually used in time-sharing systems, with the intention of keeping interactive jobs among the top priority levels and keeping batch jobs among the bottom priority levels.

Note that what matters is not a process's absolute priority, but its priority relative to the other processes it is competing with.

Priority queues



Priorities

One possible priority system for general-purpose time-sharing:

1. threads awakened after keyboard input
2. threads awakened after disk input/output
3. threads preempted after a full quantum at priority 1 or 2
4. threads preempted after a full quantum at priority 3 or 4

This algorithm, like SJF, suffers from the possibility of starving some threads if the demand for the CPU is too high.

Multi-level feedback queue

Have N queues of threads. New threads start at priority 1 with quantum Q_1.

After exhausting their quantum at priority level i with quantum Q_i, they are moved down to priority level $i+1$. One possibility for quantum Q_{i+1} is $2Q_i$.

The top levels should be close to pure RR for good response time; the bottom levels should be close to pure FCFS for good throughput. Both can be achieved if all levels are RR (or SRR) with varying parameters.

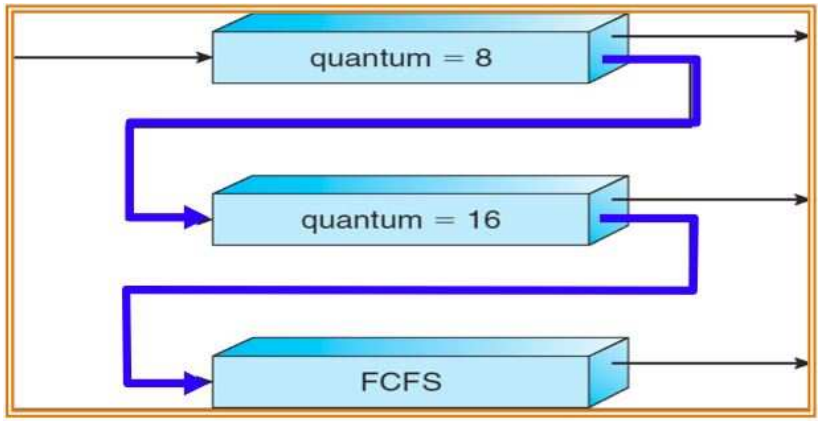
If a process is preempted before it finishes its quantum, it stays in its current position in the multilevel queue; when it starts running again, it will be given a chance to finish the remainder of its quantum.

Multi-level feedback queue

Something to think about:

If a process is preempted before it finishes its quantum, then giving it a full quantum at its current level when it starts running again would be unfair to other processes, while moving it down to the next priority level would be unfair to the process itself.

Multi-level feedback queue



Long-running "compute tasks" demoted to low Priority.

If a process is preempted before it finishes its quantum, then giving it a full quantum at its current level when it starts running again would be unfair to other processes, while moving it down to the next priority level would be unfair to the process itself.

Subsystem scheduling

A single machine may run different classes of applications, which may require different scheduling algorithms: e.g. batch, time-sharing and real-time.

The first two classes are reasonably well handled by the previous scheme. The third class has only recently become important in general purpose operating systems (due to demand for multimedia).

A typical recent scheduler will give all real-time threads permanently higher priority than all non-real-time threads.

It will also use something like multilevel feedback queues for non-real-time threads, but a different algorithm when real-time threads are runnable.

Earliest deadline first

Real-time scheduling algorithms assume that the workload consists of a set of jobs, with each job having periodic deadlines.

For example, playing an MPEG movie requires showing a new frame 25 times a second, or one frame every 40 milliseconds.

In every period of 40 ms, the movie player will spend some of its time computing the next frame and giving it to the screen controller device with instructions to display it at the end of the period (the deadline). During the rest of the period, it will be idle.

The earliest deadline first (EDF) algorithm says that the next job to run should be the one with the earliest deadline.

Estimating task duration

It is actually quite hard to predict how long a task will take.

Different inputs will require different amounts of processing.

Even the time taken by something as basic as a memory access can vary by a factor of 200 (depending on whether you get a cache hit or a cache miss).

In many cases, the only practical answer is to

- measure the time taken on lots of invocations,
- take the maximum,
- multiply it by a factor of safety (at least 2, preferably more), and
- hope it is enough.

Implementation

SJF is feasible only if the operating system requires users to supply time estimates or bounds for every job. No popular general purpose OS requires this.

FCFS and RR are fairly simple to implement.

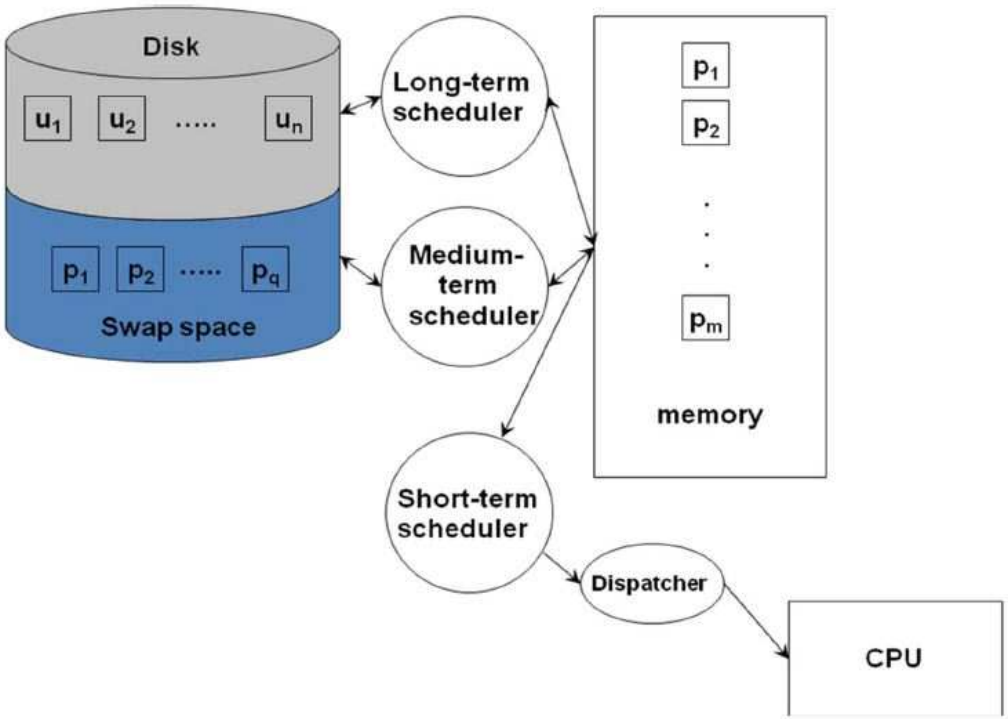
Systems with many queues should have a mechanism to find the highest priority queue with a ready process that does not require searching all priority levels.

To avoid having to handle the case of no ready processes, some systems include a null process which is always ready. Its code can be very simple: "while (1);" will do. The important point is that the null process must never suspend; this means it can't do I/O.

Implementation concerns for servers

- Time- and state-dependent algorithms should be coded in such a way as to look at as few threads as possible on each invocation.
- The efficiency gained by a better schedule can be lost through increased overhead.
- The scheduling algorithm should not be susceptible to tricks that raise the priority of processes, e.g. doing unnecessary terminal I/O.

Scheduling environment



Two-level scheduling

When insufficient main memory is available, some ready processes have to be kept on disk. Hence scheduling consists of two parts:

- deciding which processes should be in main memory, and
- deciding which of these should get the CPU.

The high-level scheduler (called the *swapper* in Unix) needs to run only rarely (it is ok to have a second or more between invocations). The low-level scheduler must run many times per second.

Choosing a scheduling algorithm

The effectiveness of schedulers is hard to compare because scheduling policies influence the arrival rate of new jobs (a sort of observer or Heisenberg effect).

Schedulers are reasonably easy to simulate but it is hard to obtain representative test data:

- Instrumentation of e.g. a dispatcher will get so much data that recording that data has substantial cost, interfering with the behavior you want to observe.
- It is hard to figure out what dependencies exist between successive events.

A fair amount of theory is available (queueing theory, Markov chains), but not many operating system designers use it.

Scheduling on multiprocessors

One useful result from queueing theory applies to multiprocessors. If you have more than one CPU, this theory says it is better to have a single queue, and have each CPU select from it, than have separate queues for each CPU. (You can see a similar effect at supermarket checkouts; it is too easy to pick the wrong queue to stand in.)

Unfortunately, accessing a single central list of processes requires too much overhead for synchronization (which we will look at later).

Most multiprocessor OSs compromise by having one queue of jobs for each CPU, but redistributing all the jobs periodically (like every 100 ms) to even out the load.

Scheduling / Multi-threaded CPUs

Some recent CPUs are *multithreaded*. In CPU design, this means that the CPU has room for the state of more than one thread.

This way, if one thread doesn't have anything to do (e.g. because it is waiting for a read from main memory, which can take 200 cycles), the CPU can switch to executing another thread.

You can combine multiprocessors and multithreading: for example, Sun's Niagara 1 has eight cores (CPUs), each of which has room for four threads.

The scheduler must know that scheduling two processes to run in different threads of the same CPU will be slower than scheduling them to run on different CPUs, since two threads of the same CPU share the use of many hardware resources, and thus usually cannot run at the same time.