CIS2380 (second term) - Week 2

Processes and Scheduling
All the runnable software, sometimes even including the operating system is organised into a number of sequential processes. A process is an executing program, including current values of the program counter, registers and variables.

Major Requirements of a Modern Operating System are (i) to allocate resources to processes (ii) to support interprocess communication and user creation of processes (iii) to interleave the execution of a number of processes to maximise processor utilisation while providing reasonable response time.

This week we look at
(i) Processes and their possible state(s) (running, waiting ..)
(ii) Schedulers - which control which processes are running, waiting etc. and - very briefly:
(iii) Interrupts ;
(iii) Threads - which allow for greater efficiency for (among other things) in processing.

(i) Processes and their possible state(s): As we saw last week, Figure 1(a) shows Multiprogramming where memory is partitioned and each program has its own piece of memory. While one job was waiting for I/O to complete, another job could run. In Fig 1(b) each program has its own flow of control and its own program counter (which is saved when it is not running). Fig 1(c) shows how only one program is active at a given time. How long an individual program takes to run, and how it actually runs on the CPU depends on other programs which are run at the same time.
Difference between a Program and a Process
A program is an algorithm expressed in some suitable notation, while a process is an activity consisting of a program run - where data has possibly to be fetched and/or output. While awaiting data from the keyboard (say) the CPU might switch to another process. A program is analogous to a recipe - while a process is analogous to the activity of preparing the dish in question. The preparation might be interrupted by a higher-priority process (like a domestic emergency) or temporarily interrupted because it is taking too long or because the cook is short of an ingredient.

The simplest model of processing is a Two State Process Model (Figure 2) which concerns a process running for a time, then giving way to a different process and queueing, then running to completion. A dispatcher or context switcher is a program which moves the processor from one process to another in order to prevent a single process from monopolising processor time.

Process Creation
There are FOUR events which cause processes to be created:
(1) System Initialisation: an example is when an operating system is booted and processes can be foreground or background. Background processes - or daemons such as email, web page requests commonly sleep most of the day but spring to life when (e.g.) email arrives. In Unix `ps` is used to list running processes\(^1\). (All processes can be listed and not just initialising processes.) In Windows 2000 the task manager is used.

(2) Execution of a process creation system call by a running process: A process creates another process (its child in Unix) - in both Unix and Windows, after a process has been created both the parent and child

\(^1\)If you use Gnome there is also a special facility.
have their own distinct address spaces. However in Unix the child address space is a copy of the parent address space - which allows for easier communication. In Windows NT the parent address space MAY be duplicated OR the parent may specify the name of a program for the operating system to load into the address space of the new process.

(3) **A user request to create a new process:** the process is initiated by a command or by clicking an icon;

(4) **Initialisation of a batch job:** this occurs when a batch job in a queue takes its turn to run.

**Process Termination**

The following conditions result in process termination

(1) **Normal Exit:** This occurs when the process has completed - and is voluntary. In this case the compiler executes a system call to tell the operating system it has finished.

(2) **Error exit:** - this occurs when the process discovers an error such as attempting to compile a program from a non-existent file. (Occasionally they ask the user to try again!). This is also a voluntary exit.

(3) **Fatal error:** this type of error is caused by the process - a program bug for example;

(4) **Killed by another process:** a process executes a system call telling the operating system to kill some other process. In Unix this call is `kill`, and in Windows it is **TerminateProcess**. In both cases the process issuing the kill command must have necessary authorisation. For example the general Unix user can only kill his/her own processes. Note that in Unix, processes can form a hierarchy (see Figure 3) - where process A created processes B and C and process B created processes D, E, F. Processes A-F are in the same group - the ‘kill’ command can be used to kill off all processes in the same group. The old version of Windows, in contrast does not have any concept of process hierarchy. The nearest is the **handle** - a token that the parent is given to control the child. However it can pass on this handle. (In contrast - the newer versions of Windows does have a concept of process hierarchy see later.)
1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

![Three State Process Model](image)

**Process States**
We have seen simplest version of processing in the 2 state model (Figure 2). A more realistic model includes the fact that a process may be blocked - which means it cannot run - for example because it is waiting for input. Thus a process can be in THREE states: blocked, ready or running. A blocked process is quite different from a process which is ready but not running. A ready process has been previously stopped for some reason. Figure 4 shows the three states and transitions between them. Transition 1 occurs when a process discovers it cannot continue - it is initiated by the process. Transitions 2 and 3 are caused by the process scheduler not the process. Transitions 2 occurs when the scheduler decides that the running process has run long enough and a different process is assigned to the CPU, Transition 3 occurs when all the other processes have had their fair share of CPU time and the first process is allowed to run. Transition 4 occurs when the external event for which the process is waiting happens. This could be for example input from a keyboard. The process is now ready to run. The three state model becomes the five state model if we add ‘start’ and ‘finish’ states. (See handout.)

(ii) **Scheduling Processes:** the objective of multiprogramming is to have some process running at all times - in order to maximise CPU utilisation. The idea is to give each process a certain amount of CPU time and when it has either run out of time or is waiting an event (e.g. input) it is put in either the ‘ready’ queue or the ‘blocked queue’. As each process in the latter is unblocked, it is put in the ready queue. (See Figure 5.) CPU scheduling takes place under the following four sets of circumstances:
(1) when a process switches from running to waiting;
(2) when a process switches from running to ready;
(3) when a process switches from waiting to ready;
(4) when a process terminates.
(Compare with the transitions of Figure 4.)
When scheduling takes place only under 1 and 4, we say the scheduling scheme is non-preemptive - otherwise it is preemptive. Once the CPU
has the process it keeps it until the process terminates or switches to the
waiting state. CPU scheduling deals with the problem of deciding which
of the processes in the ready queue is to be allocated the CPU. Different
scheduling algorithms have different properties and may favour one class of
process over another.

**First-Come-First-Served (FCFS)** - this is the simplest method to im-
plement. However it turns out that the average waiting time is large (see
tutorial 3) because if one process is heavy on CPU time it can block many
others. This is an example of a non-preemptive because there is no inter-
vention once the process has the CPU.

**Shortest-Job-First** - this is done by estimating the time each process will
need the CPU before requiring I/O. (This is called the **CPU burst**.) How-
ever it is quicker to say 'shortest job'! The scheduling algorithm is provably
optimal - as it gives minimum average waiting time. However it is difficult
to estimate which process is the candidate 'shortest'. Also - a long job may
have a very long wait. A non-preemptive version of this algorithm allows
a job to finish - whatever job arrives in the queue next. A preemptive ver-
sion of the algorithm is called **Shortest Remaining Time First** - which
estimates the remaining time to run of the current job and the jobs in the
queue. If there is a job in the queue which has a shorter time to run - it will
preempt the current job. However it is still the case that a long job may
wait.

**Round-Robin Scheduling (RR)** is a preemptive version of FCFS. A
small unit of time, a **time quantum (slice)** is defined (10 - 100 milliseconds)
and the ready queue is treated as a circular queue. The CPU scheduler
goes around the ready queue allocating each process a time interval of up
to 1 time quantum. However the average time under RR is quite long. Also
there may be a long **switching time** between one process ceasing on the
CPU and another starting. The object is to make the time quantum long
enough so that it is worth the time spent in switching processes. On the
other hand, if the quantum is too large then RR degenerates to FCFS.

**Multilevel processing** A multilevel queue-scheduling algorithm partitions
the ready queue into several separate queues according to some priority policy. (E.g. foreground jobs take precedence over background jobs.) Additionally, jobs can move between queues - so if a process uses too much CPU time it will be moved to a lower priority queue. This algorithm is the most general - however there are problems in determining (for example) the number of queues and the algorithm for each queue. Figure 6 shows 4 queues - of which the first 3 are FIFO and last is RR.

**Switching Processes**

A process is switched because (i) of an interrupt (for example the process has run out of time) or ‘hardware’ (for example a memory address is in virtual memory and needs to be brought into main memory).

**Interrupts** are a suspension of a process caused by an event external to that process and performed in such a way that the process can be resumed.

Interrupts (i) are an interruption of normal processing; (ii) improve processing efficiency by (e.g.) allowing the processor to execute other instructions while an I/O operation is in progress.

An interrupt could be due to a program bug (arithmetic overflow, division by zero, a reference outside a user’s memory space ..), the timer (previously discussed), I/O, hardware failure. Figure 7 shows a new version of the Execute/Fetch cycle - which includes interrupts.

**Context Switching:** When an interrupt the “context” of the process is saved. This includes program counter, and other registers. This is done by using the ‘process control block (PCB)’ (see Figure 8) - this is so that the process can be resumed. The PCB is updated and moved to an appropriate queue (according to the reason for the interrupt). Another process is then selected from the ready queue. Figure 9 provides a brief summary of what
Fetch/Execute/Interrupt Cycle

Interrupt Disabled

Enabled

Check for Interrupt: Process Interrupt

Halt

Figure 7: Cycle with Interrupts

<table>
<thead>
<tr>
<th>Process management</th>
<th>Memory management</th>
<th>File management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Pointer to text segment</td>
<td>Root directory</td>
</tr>
<tr>
<td>Program counter</td>
<td>Pointer to data segment</td>
<td>Working directory</td>
</tr>
<tr>
<td>Program status word</td>
<td>Pointer to stack segment</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Stack pointer</td>
<td></td>
<td>User ID</td>
</tr>
<tr>
<td>Process state</td>
<td></td>
<td>Group ID</td>
</tr>
<tr>
<td>Priority</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time when process started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Children’s CPU time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of next alarm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 8: Some of the fields in a Process Control Block/Table

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Figure 9: Skeleton of OS tasks when interrupt occurs
Figure 10: First column: items shared by all threads in a process; Second column: Some items private to each thread

happens.

Threads - these are sometimes known as ‘Light Weight Processes’ and they exist because in many applications, multiple activities are going on at once. As one thread blocks another can run. Processes are used to group resources together - while threads are the entities scheduled for execution on the CPU. Threads share an address space, open files and other resources. The CPU switches back and forth between the threads, giving the illusion that the threads are running in parallel. An example is provided by a server for a web site. Requests for pages come in and the requested page is sent back to the client. Web servers improve performance by maintaining a cache, a collection of heavily used pages in main memory. One thread, the dispatcher, reads incoming requests for work from the network. After a request it chooses a ‘worker’ thread and hands it the request. The ‘worker’ thread looks in the cache for the page - and eventually the disc if the page is not in the cache. While possibly waiting for input from the disc, the dispatcher thread can continue, or possibly another ‘worker’ thread can run. By this means, the server works more efficiently - for otherwise the server would be idle while waiting for the disc. Figure 10 shows items shared by all threads in a process (first column) and some items private to each thread (second column).

Threads in Either User or Kernel Space?
Threads can be implemented in either user or kernel space. (They can also be hybrid.) When threads are implemented entirely in user space, the kernel is unaware of them. This means a user-level threads package can be implemented on an operating system that does not support threads. Each process needs its own private thread table to keep track of the threads in that process. (See Figure 11(a).) An advantage of user-level threads is that if it is blocked by another thread in its process, it is much quicker to switch to another thread. A disadvantage is that if a thread is blocked by a page fault (when some of the program is not in main memory) the entire process is blocked by the kernel, even though other threads are runnable.
If threads are implemented in kernel space, the kernel has the thread table (see figure 11(b).) A disadvantage of threads being implemented this way is that if a thread blocks there must be a system call, at considerably greater cost than if the run-time system procedure handled the block. An advantage is (for example) that if there is a page fault, the kernel can easily check to see if another thread is runnable.

**Tutorial 3**

In next week's tutorial there will be exercises on scheduling etc.