Processes

1 Processes

1.1 What is a Process?

- The primary function of an operating system is to provide an environment where user programs can run. The operating system must provide a framework for program execution, a set of services (file management etc.) and an interface to these services. On a multiprogramming system, the operating system must also ensure that the many programs loaded into memory do not interfere with each other.
- This restricted form of program execution, with access to the services of the operating system, is known as a *process*.
- Specifically, a process is a sequence of computer instructions executing within a restricted environment. This includes:
  - An **address space**, which is a set of memory locations which stores the machine–code instructions required by process, a data area for the process' global variables, and a stack area for the stack frames which contain the process' local variables. The address space is protected, so that other processes cannot interfere.
  - A set of **CPU registers**, so that the process can perform some operations quickly (e.g additions, subtractions) without having to access memory, which can be slow.
  - The ability to invoke system calls, so that the process can obtain services from the operating system.
- This environment is often known as a process' **context**.

1.2 The Process Environment

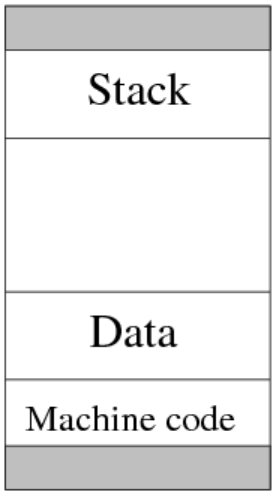
- As noted before, the operating system ensures that a process lives in a **protected** environment: it is protected against any interference from other processes currently in main memory. The operating system also prevents a process from seeing or altering other processes, and of course the operating system itself.
- This is achieved by running all of the machine code instructions of all processes in user mode. In this mode, certain privileged machine instructions are `disabled', areas of memory not owned by a process are made invalid, and access to all hardware is prevented by the CPU.

1.3 System Calls

- If the process can execute its instructions and read/write its data, but nothing else, how does it actually do anything useful, like read from a file, get mouse movements etc.? There needs to be a mechanism to allow a process to ask for specific services from the operating system, without compromising the protected environment that constrains the process.
- This is achieved by a special instruction built into the CPU: the *TRAP* instruction. A process accesses the services of the operating system by using the TRAP instruction, which passes execution directly from the process to the operating system. Information placed in the CPU's registers by the process tells the operating system what service the process wants. This mechanism of obtaining services via a TRAP instruction is known as a **system call**.
- Each operating system has its own set of available system calls. The set of available system calls is known as that operating system's **Application Program Interface** or API.

1.4 Layout of a Process

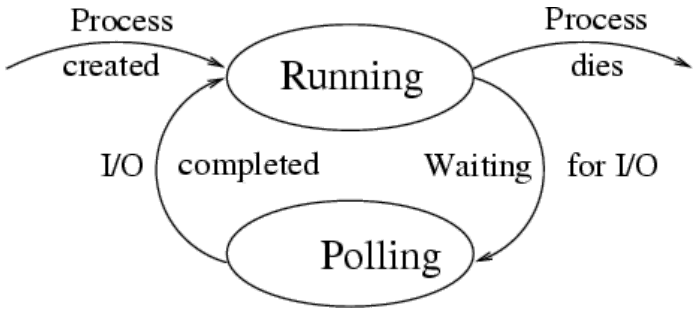
- A typical Unix process memory map looks like the following:



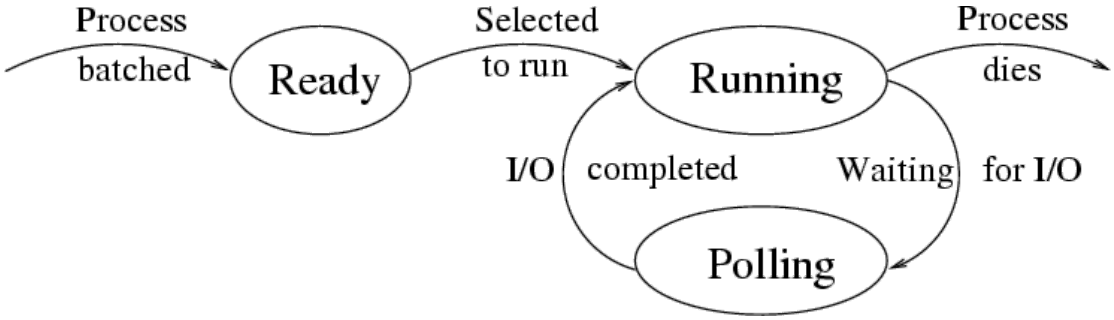
- Memory map is another term for `address space'. The memory map allows the stack to grow down and the data area to grow up: in other words, the process is able to use more main memory that the operating system initially allocates. Areas above or below the process' memory are invalid because of protections set by the operating system; they usually contain other processes. On some machine architectures, such as segmented architectures, the three sections are separated.

1.5 Process Models

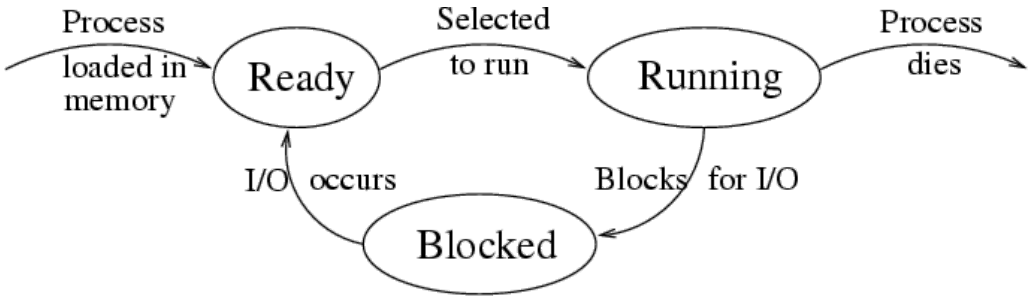
- Although I've said a process is in memory and doing something, often a process can't do anything, either because another process is running or it is waiting for I/O. A **state diagram** shows what states a process can be in, and how it moves to new states.
- On an **embedded** system, there is only one process in memory. Therefore the process in memory is either running, or it is polling for I/O. For example, a DVD player's process may be waiting for the user to hit the 'Play' button.



- Remember that polling means that either the operating system or the process itself is repetitively querying a device to see if the I/O has completed. As you would imagine, polling is very wasteful of the CPU resources.
- A **batch** system may have many processes (known as jobs in batch–speak) loaded into main memory. Although there are many jobs ready to execute, only one is selected at any time. Once that job is started it *runs to completion* before the next job can begin running. Polling is still used by the operating system to detect the completion of I/O.

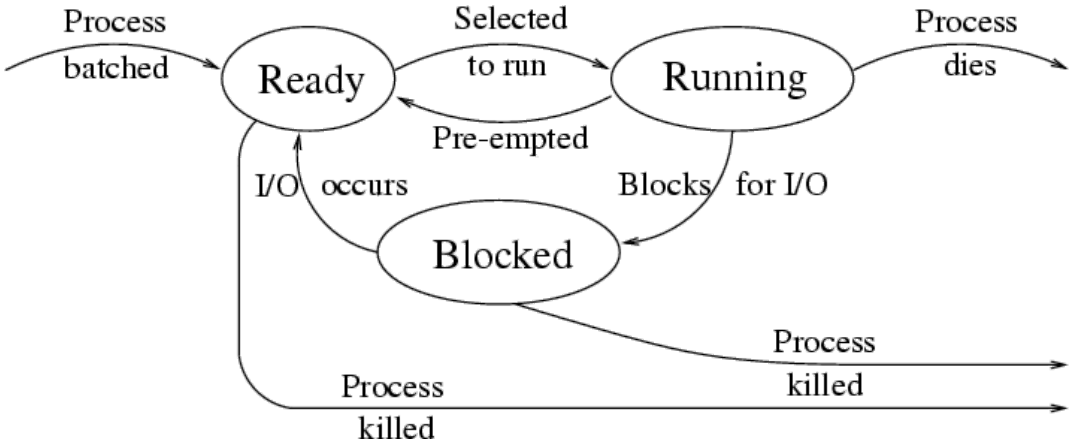


- Now, remember that I/O devices like disks, tape drives, modems, network cards, keyboards etc. are at least 1,000 times slower than main memory, and 1,000,000 times slower than the CPU. So, while the system is polling a device for the completion of input or output, the CPU is doing no useful work.
- To reduce the waste of the CPU resource by polling, on some batch systems the processes *run to block*. Instead of waiting for I/O, the operating system **blocks** the process (puts it to sleep), and selects another ready–to–run process to run.



- When the I/O is finished, the operating system finds which sleeping process requested the I/O, delivers any incoming data (if required), and moves the process from the **blocked** state to the **ready** state, as it can now execute more instructions.
- As we saw in a previous lecture, in this situation an *interrupt* from the hardware would have told the operating system that the I/O operation had finished.
- There can be more than one process in the ready state, and more than one process in the blocked state, but as there is only one CPU, there can only be **one** process in the running state. On multiprocessor systems, there can be up to one running process per CPU.
- Question: Why is this type of system unsuitable for interactive use? Think about this for a minute or two before going on to the next paragraph.

- A *run to block* system is unsuitable for interactive use for the following reason. Imagine a word processor process which was blocked waiting for some keyboard input from the user. Once the user types a key, an interrupt informs the operating system that some input is available. The operating system delivers this to the word processor, and moves it to the ready state. At this point, there is **no** guarantee that this process will ever run. If a CPU–bound process (i.e one that never does any I/O) is currently running, then it will never relinquish the CPU. Therefore the word processor will never get access to the CPU.
- The solution here is simple. On an interactive **multiprogramming** system, the system may **pre–empt** a running process to allow fast and fair use of the computer. Periodically, the operating system takes the running process off the CPU, puts it back on the ready queue, and selects another ready process to move to the running state. Thus, although the CPU is running only one process at any time, the quick switching between processes makes it appear as if many processes are running simultaneously.



- The characteristic of an interactive operating system is the transition from `Running' to `Ready'.

- Note that all processes eventually terminate. Most of the time, this is a normal termination, or the process itself detects an internal error, and terminates itself with an error code.
- Sometimes, processes get killed either because they are misbehaving, or because a human (via another process) kills them on purpose. For example, if a process tries to read a memory location for which it doesn't have permission, an exception will be sent into the kernel. The kernel may choose to kill the process, rather than let it continue. On Unix and Linux, this is known as a segmentation fault. On Windows, this is a General Protection fault.

1.6 How the Operating System Deals with System Calls

- In a pre–emptive multitasking operating system, processes don't see the pre–emption, and system calls appear to happen instantaneously. A process cannot `see' the period of time when it is not running, because it doesn't execute any instructions during this period.
- The **top–half** of the operating system deals with system call requests from the running process. This part of the operating system knows that system calls, especially those dealing with I/O, may take some time. It makes sense not to do any polling or *busy–waiting*, and to move the CPU to another process that is ready to run. Therefore, the top–half blocks the process until the I/O operation completes. The state of the running process is changed to blocked, and the operating systems chooses another running process from the ready queue.
- The **bottom–half** of the operating system reacts to all of the interrupts sent by the various devices. These tell the operating system that I/O is done. The bottom half finds the blocked process that asked for the I/O operation, changes its state from blocked to ready, and uses the RTI instruction to return back to the currently running process.
- To summarise, the *process*:
  - lives in a transparently pre–emptive scheduling environment,
  - cannot block itself, and
  - can be scheduled (stopped, restarted) by the operating system.
- The *top–half* of the operating system:
  - is only used when a system call is performed by a process,
  - is never scheduled (it isn't a process), and
  - runs until the system call is done, or blocks the calling process.
- The *bottom–half* of the operating system:
  - reacts to hardware interrupts,
  - is never scheduled (it isn't a process),
  - can never be blocked or stopped, and
  - wakes up a blocked process, making it ready to run.

1.7 Process Control Blocks

- When the operating system moves a process from the running state, it must ensure that the process' bits and pieces are kept so it can be restarted. This means that the operating system must:
  - Preserve the process' areas of memory,
  - Protect these from other processes,
  - Save copies of the CPU registers used by the process,
  - Save information about any other resources used,
  - Mark the process' new state, and
  - If blocked, record which I/O operation the process is blocked on, so when I/O completes, the process can be moved back to `Ready'.
- All of this information is stored in the **Process Control Block** in the operating system. There is one Process Control Block per process, and it usually contains several sections:

- **Machine dependent section:**
  - Register copies
  - Information about the process' memory areas
- **Machine independent section:**
  - The process' state
  - The process' priority, other scheduling information
  - Open files and their state
  - The process id
  - The user id of the user running the process
  - On what the process is blocked
- **Statistics section:**
  - Total time the process has been running, etc.
- The above is an example only of the possible types of information that must be recorded. Each operating system stores different things in its PCBs. The statistics section is used to aid the operating system in deciding how/when to allocate resources to the process, as will be seen in future lectures.
- An example PCB (from Minix) is:

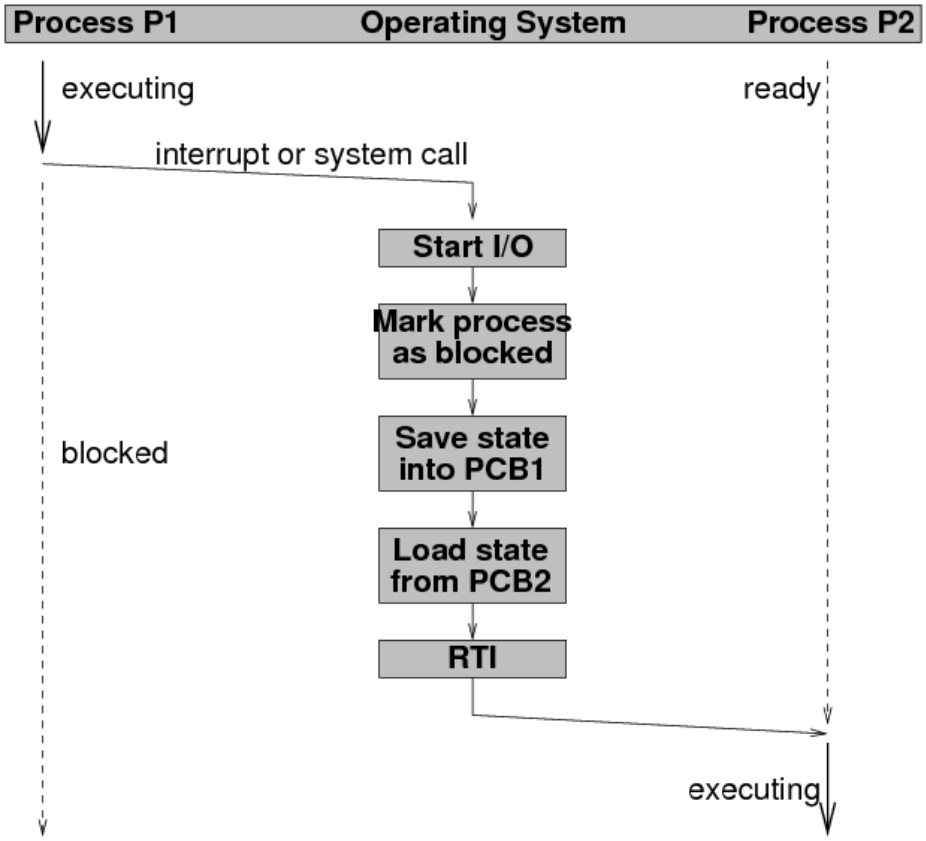
Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

(from Tanenbaum)

1.8 Context Switching

- Once a process has moved out of the ready state, the operating system must choose a ready process to run. It must perform the reverse operation, i.e:
  - Enable the process' areas of memory,
  - Reload the registers used by the process,
  - Mark the process' new state as running,
  - Start the process from where it left off, and
  - Return the result of any system call to the process.

- The operation of changing running processes is known as a **context switch**, and often has a significant overhead (typically hundreds of machine instructions). This goes against the philosophy of operating system design which tries to minimise the number of instructions not used by user processes.



2 Process Scheduling

2.1 Introduction

- On a multiprogramming system, there is usually a queue of processes blocked waiting for I/O, and a queue of processes waiting to run. There is only ever 0 or 1 processes actually running. When a process vacates the running position, the operating system needs to choose a replacement from the ready queue. This is known as *process scheduling*.
- On interactive systems, a process leaves the running position because it dies, it blocks on I/O, or it is *pre-empted*. In the latter case, the running process is placed back on the ready queue. As we just saw, context switching between processes on the ready queue and the running position takes the operating system some time, thus wasting resources that could be used by user processes.
- However, if pre-emption is not done often enough, when a user's keypress unblocks a process and moves it back to the ready queue, it may take seconds before the process runs and responds to the keypress. On most interactive systems, a running process gets pre-empted after a period known as the **quantum** or **timeslice**. This is typically 10 milliseconds.

2.2 Process Behaviour

- Most processes exhibit a behaviour where they do some CPU work, and then block as they perform I/O. However, there is a spectrum of behaviour. Some processes (e.g. simulations, scientific calculations) spend nearly all their time executing instructions: these are *CPU-bound processes*. Some processes (e.g word processors, web servers) spend nearly all their time waiting for I/O to occur: these are *I/O-bound processes*. Obviously, I/O-bound processes need less CPU time to perform their operation as compared to CPU-bound processes.
- Process behaviour will vary over time. For example, a spreadsheet may be I/O-bound while the user enters data, but become CPU-bound when it is asked to perform some heavy calculations. Then it may go back to being I/O-bound.

- Also note that, as CPU performance increases, the time required to execute instructions decreases; therefore, overall process behaviour is tending towards the I/O–bound end of the spectrum.

2.3 Scheduling Goals

- Operating systems **schedule** processes in order to try and achieve certain goals:

  a)
  Fairness: each process gets a fair share of the CPU.

  b)
  Efficiency: keep the CPU as busy as possible.

  c)
  Response time: minimise response time for interactive users.

  d)
  Throughput: maximise the number of processes run per hour.

  e)
  Policy: ensure that the stated scheduling policy is enforced.

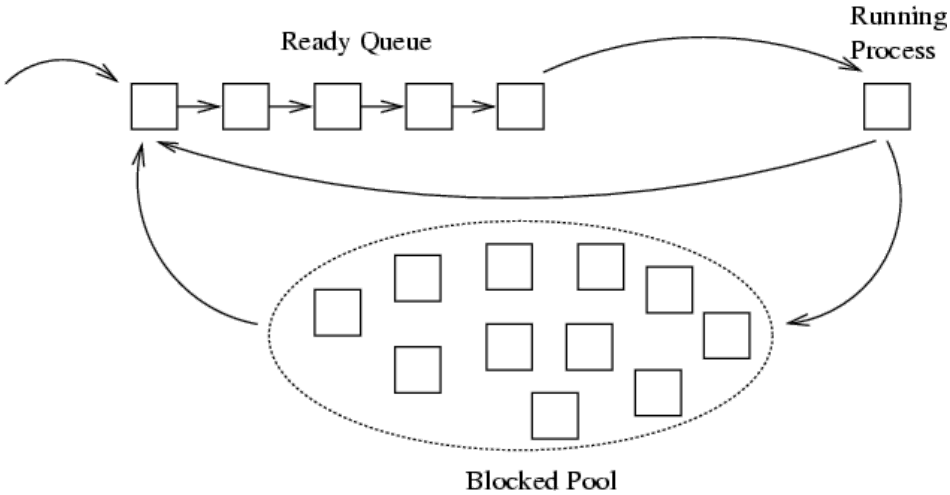
- However, not all of these can be satisfied at the same time, because the CPU is a finite resource. For example, to satisfy goal d), you would process only batch jobs and never do any pre–emptions, to minimise context switching time; unfortunately, this would violate goal c). For any system, a policy has to be set which tries to meet the stated scheduling goals for the system. The process scheduler then implements the policy.
- On a mainframe system, throughput is probably the main goal. On these systems, schedulers may implement **run to block/completion** scheduling: wait until a process blocks or dies before rescheduling. This policy would not be suitable for server or desktop systems, as the running process may not block for days if it is CPU–bound.
- On server systems, the goal is probably to minimise response time. This is because there are external clients which require a response to their questions. On these systems, schedulers may implement **pre–emptive** scheduling, and suspend a CPU–bound running process after a period of time known as the process' **quantum** or **timeslice**. This allows other processes to run, even if the original process hadn't blocked, and allows processes that have had their I/O completed a chance to run and respond. Pre–emption is needed for interactive systems.
- On desktop systems, response time is a goal. The scheduling policy might also try to favour the process which the user is currently interacting with. For example, if a process has an open window, and that window has focus (i.e mouse events and keyboard events are being sent to that window), then it is most likely the process that the user is working with. This process may be given a higher running priority than other processes.

2.4 Scheduling Algorithms – Interactive (Pre–emption)

- We will concentrate on scheduling algorithms which are suitable for server and desktop systems.
- There are several algorithms available to choose a process from the list of ready–to–run processes when the running state has been vacated. Each one reflects a certain system policy for process scheduling. Each algorithm has its advantages and disadvantages. Wee will look at:
  - First Come First Served/Round Robin
  - Timeslice Priority
  - Multiple Priority Queues
  - Long Term Schedulers

2.5 First Come First Served/ Round Robin

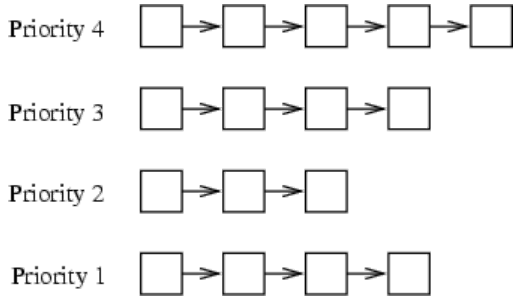
- This scheduling algorithm works as follows:
  - Created processes are put on the tail of the **ready queue**.
  - When the running process blocks, move it to the **blocked pool**.
  - As each process unblocks, place it at the tail of the ready queue.
  - Alternatively, when the running process is pre–empted, place it at the tail of the ready queue.
  - When the running state is vacant, move the process at the head of the ready queue to the running state.



- This scheduling algorithm is easy to implement, but has one big disadvantage: one **CPU–bound** process will hog the CPU, thus reducing the overall **throughput**.
- How can this happen? It should be obvious that all processes will get equal opportunity to enter the running state. However, once there, CPU–bound processes will use up all of their *timeslice*, whereas processes that do lots of I/O will block, giving up the CPU to another processes. They must then cycle through the blocked state before they can get back to the ready state. Small **I/O bound** processes therefore tend to wait in the ready state for the CPU–bound process to give up the CPU.

2.6 Priority Scheduling

- One solution to this is to introduce some concept of scheduling priority to the system: when the running position is vacant, the ready process with the highest priority is chosen to run. How the priority is established is part of the scheduling priority. On multi–user systems, perhaps certain groups of users (e.g. staff) may be given higher priority than others (e.g. students). On a desktop system, the processes with open windows may be given higher priority than background processes (e.g. mail delivery), and multimedia processes may be given even higher priority.
- Assuming that process priorities have been set, the single ready queue can be split into multiple ready queues, ordered by priority.



- When choosing the next process to move to the running state, the operating systems schedules the process from the highest priority non–empty queue. This can lead to *starvation* of lower priorities if one queue has many CPU bound processes, as the lower queues only get CPU time when the upper queues are empty (no ready processes at that priority, or all the processes at that priority are blocked).

2.7 Dynamic Priority Scheduling

- In some priority systems, a process' priority remains fixed. This often does not reflect the changing importance of a process. For example, a user is probably not interested in a mail client while it is fetching new mail, but is interested in the client when they want to read the mail. Also remember, a process' behaviour over time may change, and so the priority that it has been given might also need to change.
- A dynamic priority scheduler is designed to enforce a more sophisticated system policy with regards to process scheduling, by changing processes' priorities over time. For example, a scheduler may lower a process down through the priority queues if the system is an interactive one and the process is CPU–bound. The aim here is to lessen the effect of the CPU–bound process on the interactive ones.

2.8 The Unix Dynamic Priority Scheduler

- Unix uses multiple priority queues, and has a dynamic priority scheduler which moves processes up or down in priority. The policy is to give priority to low–CPU usage processes, as they are usually interactive processes which are I/O–bound on user input.
- There are 32 ready queues, with priorities ranging in groups of 4 from [0–3] (highest priority) up to [124–127] (lowest priority). Every process starts at the highest priority of 0.
- Once a process is given the running position, it can leave this position by blocking for I/O, or by being pre–empted. If the process gets blocked, then its priority remains unchanged. If the process is pre–empted, then its priority value goes up by 1 (i.e. its priority decreases by 1). If the process is CPU–bound, if there are no other processes competing for the CPU, and if the timeslice is 10 milliseconds, then the process' priority could decrease by 100 every second. The aim here is to penalise processes which use too much CPU time and keep exceeding their timeslice.
- Every second, *all* processes' priority values are halved. Thus, a process which uses little CPU has its priority value repetitively halved, and the priority value tends towards 0. A process which uses a lot of CPU keeps increasing its priority value, and hence maintains a priority value well above 0, even in the face of the repetitive halving.
- Here is a snapshot of a set of processes running on a Unix machine:

```
load averages:   0.29,  0.11,  0.11
31 processes:  2 running, 29 sleeping
Cpu states: 97.7% user,  0.0% nice,  1.9% system,  0.4% interrupt,  0.0% idle

PID USER PRI   SIZE   RES STATE   TIME   WCPU    CPU COMMAND
3071 wkt  57    608K  228K run     0:03 88.08% 15.98% gunzip
 715 wkt  18    636K  480K sleep   0:14  0.00%  0.00% tcsh
 714 wkt  18    584K  432K sleep   0:02  0.00%  0.00% tcsh
  95 wkt  18    416K  120K sleep   0:09  0.00%  0.00% sendmail
  86 wkt  18    280K  104K sleep   0:06  0.00%  0.00% cron
  25 wkt  18    216K    0K sleep   0:00  0.00%  0.00% <adjkerntz>
 685 wkt  10    488K    0K sleep   0:00  0.00%  0.00% <sh>
   1 wkt  10    364K    0K sleep   0:00  0.00%  0.00% <init>
 701 wkt  10    200K    0K sleep   0:00  0.00%  0.00% <xinit>
 718 wkt   3    200K   44K sleep   0:08  0.00%  0.00% sshd
1077 wkt   3    200K    0K sleep   0:07  0.00%  0.00% <sshd>
 137 wkt   3    156K    0K sleep   0:00  0.00%  0.00% <getty>
 136 wkt   3    156K    0K sleep   0:00  0.00%  0.00% <getty>
```

- The `gunzip` process is running and has a low–ish priority of 57. Processes like `tcsh`, `sendmail` and `cron` have run recently (which is why they have a priority of 18), but are currently blocked on I/O. Processes like `sshd` and `getty` have not run recently (which is why they have a priority of 3), and are also currently blocked on I/O.
- The load averages give the number of jobs in the Ready queues averaged over 1, 5 and 15 minutes.

2.9 Fair–Share Scheduling

- On a multi–user system like Unix, if a user can solve a problem using many processes, then each process will get scheduled independently, and as a group, the processes will get more CPU time than any single process. This gives the user an unfair advantage over other users.
- The same problem may occur on server systems. For example, an Apache web server may start a separate process for each incoming client connection. These processes may compete unfairly with the database, if it is only running as a single process.
- One way to combat this is with fair–share scheduling. Each user is allocated some percentage of the CPU time, and this is divided amongst their processes.
- There are many ways of doing this. None are entirely satisfactory. There is also the issue of what to do with spare CPU time. For example, if three users are allocated 33% of the CPU each, but the processes of one user never consume more than 25%, what should the scheduler do with the spare CPU time?

2.10 Which is the Right Scheduling Algorithm to Use?

- There is no simple answer to this question. The right choice of algorithm depends heavily on the type of system, and the types of processes running on the system. A good algorithm can be selected by gathering data on the current scheduling algorithm and the types of processes that are run by the users, simulating new algorithms

using the data collected, developing analytic models of the CPU usage patterns, and testing the new algorithms out on a real system.

- It is also important to note that no matter how good a scheduling algorithm is, the available CPU resources are finite. The number of processes completed by an operating system doesn't increase as more processes are loaded into memory; in fact, the throughput decreases due to the overhead of context switching between the processes. Some operating systems (such as batch systems) have algorithms that limit the burden on the CPU by preventing new processes from being loaded. Interactive systems generally don't do this, as users would get frustrated if they could not launch a new application. This does mean that users on an interactive system can accidentally or intentionally cause a CPU *denial of service*.

2.11 The Idle Process

- What does the CPU do when there are no processes in the ready queue, and a process must be chosen to move to the vacant running position? Here the operating system schedules an **idle process**, which is a CPU–bound process built into the operating system. The idle process is given the lowest priority possible, so it is not executed if there are any ready processes in the system.

File translated from TEX by TTH, version 3.85.
On 8 Dec 2010, 08:43.