MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

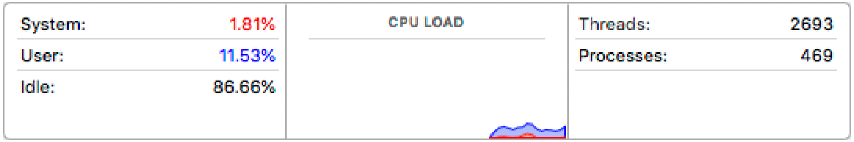
159.341 Programming Languages, Algorithms & Concurrency

Processes (Part 1)

Daniel Playne
d.p.playne@massey.ac.nz

Processes

Essentially every modern computer is actually performing many tasks at once. Operating systems run many different processes (many unknown to the user).



System:	1.81%
User:	11.53%
Idle:	86.66%

Threads:	2693
Processes:	469

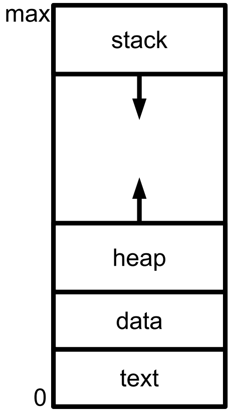
Processes on my computer while making these slides

These processes run a range of system tasks such as checking for incoming email or updating antivirus definitions as well as the user applications such as web browsers, music programs, text editors etc.

Processes

Processes are a key concept in concurrency - a process is essentially a program that is currently being executed.

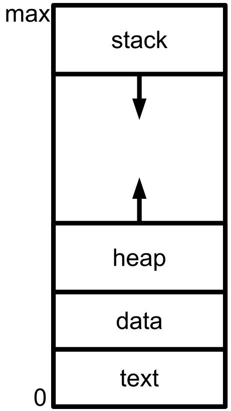
Each process has the **program** it is running along with an **address space** or the part of memory the process can access and stores the data of the program, its stack and the program executable itself.



Processes

Structure of a process address space:

- stack - local variables, return address
- heap - dynamically allocated memory
- data - global variables
- text - program, program counter, registers



Processes

A program is a ***passive*** entity, it is just a file containing a list of instructions.

A process is an ***active*** entity that contains everything needed to actually run a program (including but not limited to the program).

A program becomes a process when the operating system loads it into memory and set up the necessary resources for a process.

Processes

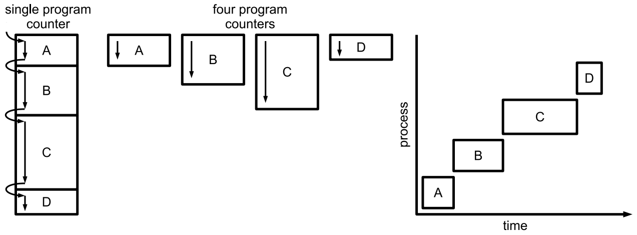
In a multiprocessing system, the CPU will switch back and forth from one process to another very quickly running each process for tens or hundreds of milliseconds.

On single-core machines this is sometimes called **pseudoparallelism** because it gives the illusion that many process are all executing at once (in reality the CPU only executes one at a time).

On a multi-core/processor machine this is true parallelism as the multiple cores will actually be executing multiple processes at the same time.

The Process Model

Each process conceptually has it's own virtual CPU that it executes on, in reality the CPU switches between different processes executing.



Different ways of conceptualising processes executing.

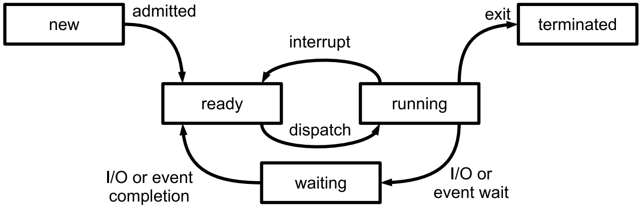
Process State

As a process executes, it transitions through several different states:

- **New** - the process is being created
- **Running** - instructions are being executed on the CPU
- **Waiting** - the process is waiting for an event (I/O or signal)
- **Ready** - the process is ready to execute and is waiting to be scheduled
- **Terminated** - the process has finished execution

Process State

The transitions between the different process states are shown below:



Process state diagram.

Process Control Block

In order to switch between processes, the operating system stores information on each process's state in a ***Process Control Block***.

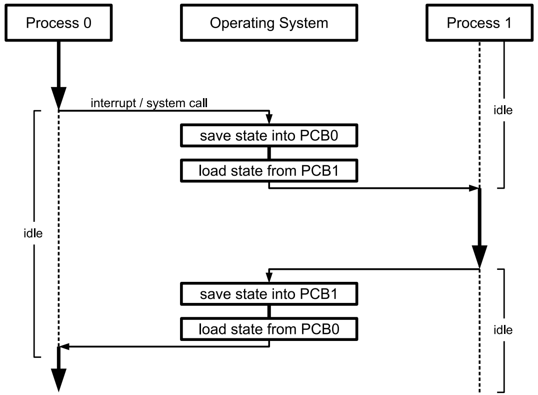
- **Process state** - new, ready, running, waiting, terminted
- **Program counter** - the address of the next instruction for this process.
- **Registers** - the value of the general and special* registers.

* special registers include stack pointers, index registers, accumulators.

Process Control Block

- **Scheduling Information** - process priority, scheduling queues etc
- **Memory Management** - base & limit registers, page tables etc.
- **Accounting** - CPU and real-time used, time limits, etc
- **I/O Status** - list of I/O devices, open files etc.

Process Switching



Process switching diagram.

Process Scheduling

The goal of multiprogramming is to maximise CPU utilisation by ensuring that some process is running at all times.

The goal of time sharing is to switch the CPU between processes frequently so that users can interact with each program.

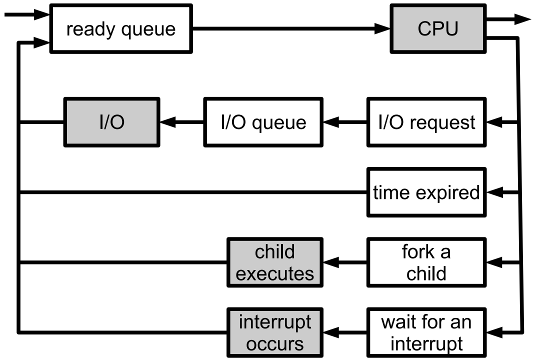
To meet these objectives, the **process scheduler** must select which process to execute (and for how long).

Scheduling Queues

When a process is created it is put into a **job queue** which contains of all the processes in the system. Processes held in main memory that are ready and waiting to execute are put in the **ready queue**.

There are other queues in the system - processes that access shared devices (such as a disk) must wait for the device to respond to their I/O request. Each device has its own **device queue**.

Scheduling Queues



Queueing diagram.

Process Scheduling

In a batch system, there are normally more jobs submitted than can be executed. The **long-term** or **job scheduler** selects a process from the pool of jobs and loads it into memory for executing.

The **short-term** or **CPU scheduler** selects processes from the ready queue and allocates them to run on the CPU.

Process Scheduling

The main difference between these schedulers is their frequency - the **short-term** scheduler must select new processes very quickly. A process may only run for a few milliseconds before having to wait for I/O etc.

Short-term schedulers execute at least once every 100 milliseconds - if the scheduler took 10 milliseconds to select a process it would be wasting $\approx 9\%$ of the CPU time.

Process Scheduling

On the other hand, the **long-term** scheduler may be executed far less frequently, minutes can separate the creation of two processes.

The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory) and must make careful decision about which process to add to the job queue.

Process Scheduling

In general processes can be described as being either **I/O-bound** or **CPU-bound**.

- **I/O Bound** - spends more time doing I/O than computation, many short CPU bursts.
- **CPU Bound** - spends more time doing computation, fewer long CPU bursts.

The long-term scheduler will aim to have a good mix of both I/O- and CPU-bound processes to keep the resources of the computer busy.

Process Scheduling

Some operating systems such as time-sharing systems like UNIX and Windows do not have a long-term scheduler. All new processes are put on the job queue and are scheduled by the short-term scheduler for CPU time.

These systems often rely on self-limiting behaviour of the users. If the computer starts to run too slowly then the users will close applications.

Process Operations

Process Operations

Process Creation

Processes can create other processes during their execution - these are usually referred to as child processes. These child processes can in turn can create other processes and so on (forming a tree of processes).

Most operating systems will identify each process with a unique **pid** identifier.

Process Creation

When a process creates another process there are usually two possibilities for execution:

- Parent and child execute concurrently
- Parent waits until some or all of its children terminate

There are also two options for address-space

- The child process is a duplicate of the parent.
- The child loads an entirely new program.

Process Creation - UNIX

For an example for creating and using processes in UNIX there are three system calls needed:

`fork()`
Create and launch a new process with a copy of the address space of the original process. The function will return a **pid** which differs between the two processes. The original process will get the **pid** of the child process while the child process receives a 0.

Process Creation - UNIX

`exec()`
Can be used to replace the process's memory space with a new program. This will load a binary file into memory and start execution.

`wait()`
A parent process can wait until a specific child process or all children have terminated.

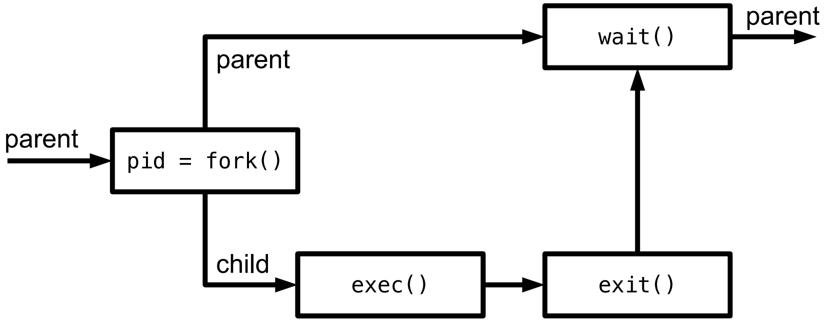
Process Creation - UNIX

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main() {
   pid_t pid;
   pid = fork();

   if(pid < 0) { // error
      return 1;
   } else if(pid == 0) { // child process
      execlp("/bin/ls", "ls", NULL);
   } else { // parent process
      wait(NULL);
      printf("Child process complete");
   }
   return 0;
}
```

Process Creation - UNIX



Process creation diagram.

Process Creation - Windows

With the Windows API, processes are created using the `CreateProcess()` function which allows a process to create a new child, but rather than loading the address space of the parent it will always load a specific program into the child.

This approach can be somewhat unwieldy with `CreateProcess()` requiring ten parameters to be specified when creating a new process.

Process Creation - Windows

```c
#include <stdio.h>
#include <windows.h>

int main() {
   // Startup and process information
   STARTUPINFO si;
   PROCESS_INFORMATION pi;

   // Clear Startup and Process information
   ZeroMemory(&si, sizeof(si));
   si.cb = sizeof(si);
   ZeroMemory(&pi, sizeof(pi));

   // Create new process
   if (!CreateProcess(NULL,      // use command line
          "C:\\WINDOWS\\system32\\mspaint.exe",   // command
          NULL,       // don't inherit process handle
          NULL,       // don't inherit thread handle
          FALSE,      // disable handle inheritance
          0,          // no creation flags
          NULL,       // use parent's environment block
          NULL,       // use parent's existing directory
          &si, &pi)) {
```

Process Creation - Windows

```
int main() {
    ...
    // Create new process
    if (!CreateProcess(...)) {
            // Error
            fprintf(stderr, "Create Process Failed");
            return -1;
    }

    // Wait for child process to complete
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    // Close Handles
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Process Creation - Fork Bomb

A **fork bomb** or a **rabbit**/**wabbit** is a program which is designed to slow down or crash a computer system by infinitely creating additional processes.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main() {
    while(1) {
        fork()
    }
}
```

Process Termination

A process terminates when it completes executing the final statement and asks the operating system to terminate it by calling `exit()` which usually returns a status value indicating the nature of the termination.

When a process terminates, all resources (I/O buffers, open files, physical/virtual memory etc), allocated to the process are deallocated by the operating system.

Process Termination

There are some other ways that a process can be terminated, a parent process can terminate a child process. This may be because:

- The child has exceeded its usage of resources
- The task of the child is no longer necessary
- The parent is exiting and the operating system does not allow a child to continue without the parent.

Some systems do not allow a child process to continue if the parent has been terminated. When a parent process terminates, all the child process are also terminated in something called **cascading termination**.

Process Termination

A process in UNIX terminates and returns a status value when `exit(1)` is called (can also return a termination status with a `return` in the `main()` function). A parent process can wait for a child process to terminate using `wait()`.

The parent can obtain the **pid** of the terminating process and well as the status.

```
pid_t pid;
int status;

pid = wait(&status);
```

Process Termination

When a process terminates, the resources allocated to the process are deallocated by the operating system but the entry in the process table remains until the parent calls `wait()`.

A process that has terminated but whose parent has not yet called `wait()` is called a **zombie** process (not quite dead but not alive).

All processes become zombies on termination but usually only exist in this state very briefly.

Process Termination

If a parent doesn't call `wait()` and instead terminates, the child process could be left in the process table as an **orphan**.

UNIX and Linux address this by assigning orphan processes a new parent - `init` which is the root of the process hierarchy in UNIX & Linux. This process will periodically call `wait()` to allow orphan process to be released from the process table.

Summary

- Process States
- Process Lifecycle
- Scheduling Queues
- Process Operations