Process Life Cycle (part 1)

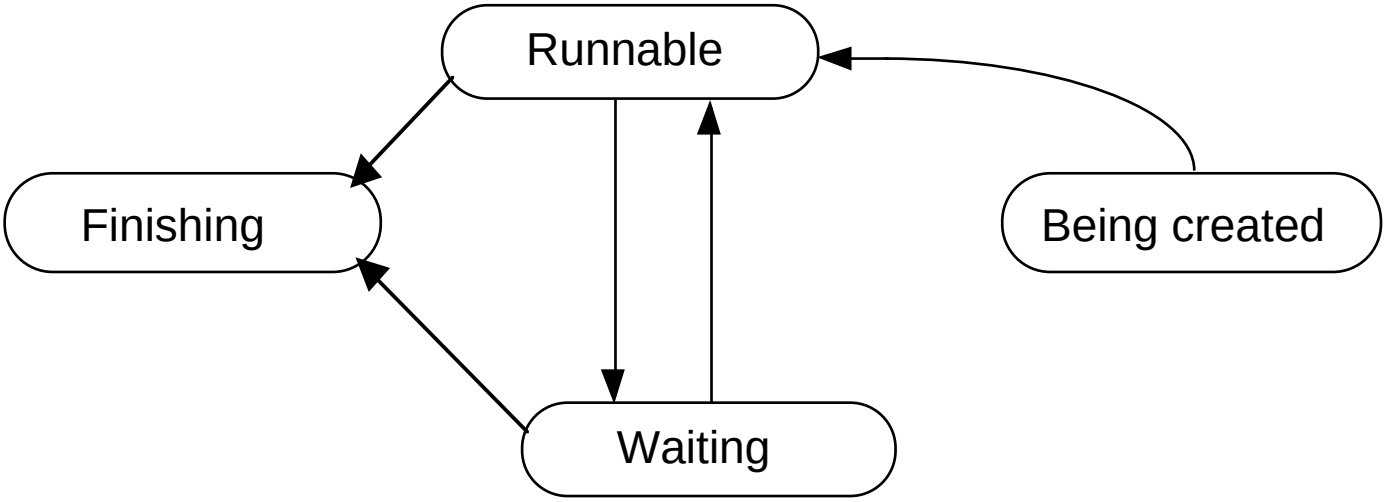
Birth

Life

Waiting/Sleeping

including the way it works in UNIX

Process States (Status in the notes)



Different methods of creating processes

create process system call - takes a program name

copy process system call - a strange way of doing it but is now very widespread thanks to UNIX

create a new terminal session

Whichever way

- find a spare (or create a new) PCB

  what if there isn't one?

- mark it "being created"
- generate a unique identifier

  why isn't a pointer good enough?

- get some memory or

  at least fill in the page table entries

- set up PCB fields with initial values

  priority, resource limits

- when all set up change the state to "runnable"

  this could be done by inserting into a queue of runnable processes

  what about other resources?

  some OSs carefully allocate resources before a process runs

  others leave this to the process to collect as it runs

Creating a thread

much easier

- find a spare (or create a new) thread data structure
- allocate some stack space

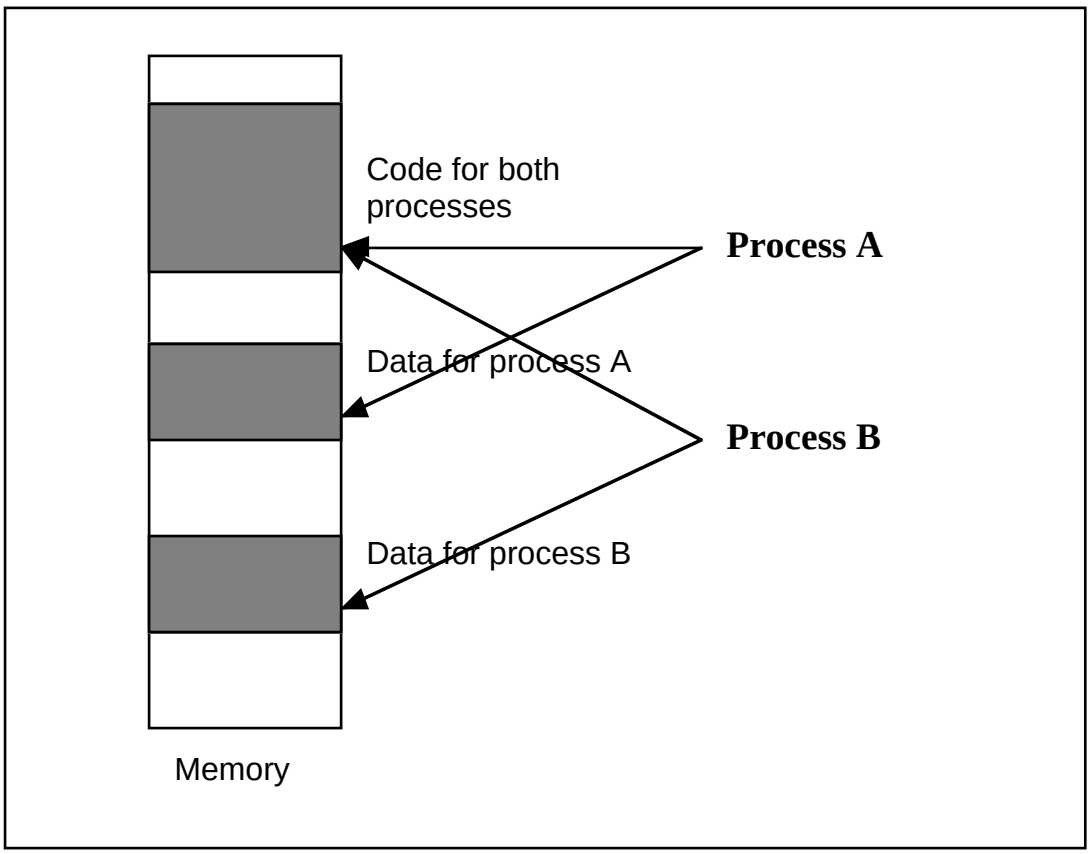
fork

The UNIX (originally) fork call duplicates the currently running process.

parent process - the one which made the call

child process - the new one

Traditionally memory was duplicated - the code was shared even from earliest days.



Since they are exact (almost) copies they share the same resources, in fact if they both write to a file their output will be interleaved.

Open file information blocks will have the count of processes using them increased by one.

Same thing with shared memory regions.

Fork returns 0 in the child process and the child's pid in the parent.

How many processes are started up by the following code?

```
main()
{
 int i;
 i = 0;
 while (i < 2) {
  fork();
  i++;
 }
}
```

Usually calls to fork are followed by calls to exec in the child process.

```
if (fork() == 0)
 execl("nextprog", "nextprog", 0);
```

which keeps the process the same but changes the program

exec

checks to see if the file is executable

saves any parameters in some system memory

releases currently held memory

loads the program

moves the saved parameters into the stack space of the new program

ready to run again

Fork used to always copy the data memory of the process.

If the child is going to do an exec this is a waste of effort.

Particularly bad with virtual memory.

So most systems now use a copy on write

vfork() - another solution which doesn't make a copy

programmers use it when an exec is going to follow

How many processes are created by the earlier code if we use vfork?

Back to our process/thread which is now runnable

The "dispatcher" part of the scheduler discussed in detail later in the course chooses processes/threads to run from the runnable queue (or queues).

The change from one process/thread running to another one running on the same processor is usually referred to as a "context switch". Sometimes we also talk about a context switch when a process performs a system call. More about that later too.

Being runnable

On a single processor only one process/thread can run at a time.

Many may however be runnable - either running or ready to run.

The process/thread itself cannot directly detect when it is not running

it sees no difference between ready and running.

The OS sees things differently.

What causes the change from running to ready (or ready to running)?

Cooperative multitasking

Either the process/thread tells the system that another process can run now,

or the system uses a sneaky method hidden inside a system call which all processes are supposed to call regularly to start another process.

This does not mean a task will necessarily work to completion

in fact it is very unlikely. *Why?*

Preemptive multitasking

A clock interrupt causes the OS to check to see if the current process/thread should continue

Each process/thread has a time slice

More about time slices later

Usually 10 - 100 milliseconds

What advantages/disadvantages does preemptive multitasking have over cooperative multitasking?

control

predictability

critical sections

efficiency

A mixture

Some systems combine these methods

e.g. UNIX traditionally has not allowed preemptive multitasking when a process has made a system call

Regardless, the condition (state) of the process/thread has to be stored so that when it recommences nothing has changed - in particular process registers

If changing processes the page table needs altering

rest of environment restored

If changing threads within the same process simply

restore registers including pc and the threads stack

Some systems have multiple sets of registers

eventually even these overflow and memory must be used to store them

Waiting or Sleeping

Not to be confused with Java's wait() and sleep()

Processes seldom have all the resources they need when they start

- memory
- data from files or devices e.g. keyboard input

and these things require time to get

- process/thread waits
- multiprogramming - no point in keeping the whole system waiting

so

- state is changed to waiting
- may be more than one waiting state
  - short wait e.g. for memory vs
  - long wait e.g. for an archived file
  - this is important for scheduling decisions
- removed from the ready queue
- possibly entered on a queue for whatever it is waiting for

when the resource becomes available

- state is changed to runnable
- removed from the waiting queue
- put back on the runnable queue

Suspended

another type of waiting

ctrl-z in some UNIX shells

operators or OS temporarily stopping a process

- allows others to run to completion more rapidly
- or to preserve the work done is there is a system problem

UNIX waiting

- the queue is associated with the hash value of a kernel address
- (waiting or suspended processes may be swapped out)

when the resource becomes available

- originally used to scan whole process table
- all things waiting for that resource are woken up
- (may need to swap the process back in)
- first one to run gets it
- if not available when a process runs the process goes back to waiting

a little like in Java

```
while (notAvailable)
   wait();
```