Processes

We now look at the programs we want to run

Processes

We now look at the programs we want to run

The word *process* is used to describe

- the executable code *and*
- its data *and*
- the associated information the OS needs to run it

Processes

We now look at the programs we want to run

The word *process* is used to describe

- the executable code *and*
- its data *and*
- the associated information the OS needs to run it

Note this is different from *processor* and *program*

Processes

We now look at the programs we want to run

The word *process* is used to describe

- the executable code *and*
- its data *and*
- the associated information the OS needs to run it

Note this is different from *processor* and *program*

Other words: task, job

Processes

A single program might possibly use more than one process

Processes

A single program might possibly use more than one process

For example, one process to compute a picture and another to display it: this is called *structure by process*

Processes

A single program might possibly use more than one process

For example, one process to compute a picture and another to display it: this is called *structure by process*

Though this is perhaps the exception, these days. Quite often a program uses just one process

Processes

A single program might possibly use more than one process

For example, one process to compute a picture and another to display it: this is called *structure by process*

Though this is perhaps the exception, these days. Quite often a program uses just one process

And structuring can be done by using multiple *threads of execution*, often running in parallel

Processes

A single program might possibly use more than one process

For example, one process to compute a picture and another to display it: this is called *structure by process*

Though this is perhaps the exception, these days. Quite often a program uses just one process

And structuring can be done by using multiple *threads of execution*, often running in parallel

But it is coming back in Web browsers using one process per tab to provide security isolation between tabs

Processes

A single program might possibly use more than one process

For example, one process to compute a picture and another to display it: this is called *structure by process*

Though this is perhaps the exception, these days. Quite often a program uses just one process

And structuring can be done by using multiple *threads of execution*, often running in parallel

But it is coming back in Web browsers using one process per tab to provide security isolation between tabs

Note to think about later: Web browsers use OS process protection and isolation mechanisms to provide tab protection and isolation

Processes

An OS needs to keep a lots of information about a process, including

Processes

An OS needs to keep a lots of information about a process, including

- where in memory its code is

Processes

An OS needs to keep a lots of information about a process, including

- where in memory its code is
- where in memory its data is

Processes

An OS needs to keep a lots of information about a process, including

- where in memory its code is
- where in memory its data is
- what permissions it has on those parts of memory (MMU flags)

Processes

An OS needs to keep a lots of information about a process, including

- where in memory its code is
- where in memory its data is
- what permissions it has on those parts of memory (MMU flags)
- how much time it is allocated

Processes

An OS needs to keep a lots of information about a process, including

- where in memory its code is
- where in memory its data is
- what permissions it has on those parts of memory (MMU flags)
- how much time it is allocated
- how much time it has used

Processes

An OS needs to keep a lots of information about a process, including

- where in memory its code is
- where in memory its data is
- what permissions it has on those parts of memory (MMU flags)
- how much time it is allocated
- how much time it has used
- similarly for other shared resources, e.g., the amount of I/O or networking done

Processes

An OS needs to keep a lots of information about a process, including

- where in memory its code is
- where in memory its data is
- what permissions it has on those parts of memory (MMU flags)
- how much time it is allocated
- how much time it has used
- similarly for other shared resources, e.g., the amount of I/O or networking done
- the cpu's PC and registers

Processes

An OS needs to keep a lots of information about a process, including

- where in memory its code is
- where in memory its data is
- what permissions it has on those parts of memory (MMU flags)
- how much time it is allocated
- how much time it has used
- similarly for other shared resources, e.g., the amount of I/O or networking done
- the cpu's PC and registers
- and lots more as we shall see later

Processes

An OS needs to keep a lots of information about a process, including

- where in memory its code is
- where in memory its data is
- what permissions it has on those parts of memory (MMU flags)
- how much time it is allocated
- how much time it has used
- similarly for other shared resources, e.g., the amount of I/O or networking done
- the cpu's PC and registers
- and lots more as we shall see later

It uses this information to schedule and protect the process

Processes

A process can be in one of several *states*. In a simplified model, the five main states are

Processes

A process can be in one of several *states*. In a simplified model, the five main states are

1. New. A process that has just been created

Processes

A process can be in one of several *states*. In a simplified model, the five main states are

1. New. A process that has just been created
2. Running. It is currently executing on the CPU

Processes

A process can be in one of several *states*. In a simplified model, the five main states are

1. New. A process that has just been created
2. Running. It is currently executing on the CPU
3. Ready. It is ready to run, but not currently running as some other process is currently using the CPU

Processes

A process can be in one of several *states*. In a simplified model, the five main states are

1. New. A process that has just been created
2. Running. It is currently executing on the CPU
3. Ready. It is ready to run, but not currently running as some other process is currently using the CPU
4. Blocked. Can't run right now as it is waiting for some event or resource to become available. E.g., waiting for a block of data to arrive from the disk

Processes

A process can be in one of several *states*. In a simplified model, the five main states are

1. New. A process that has just been created
2. Running. It is currently executing on the CPU
3. Ready. It is ready to run, but not currently running as some other process is currently using the CPU
4. Blocked. Can't run right now as it is waiting for some event or resource to become available. E.g., waiting for a block of data to arrive from the disk
5. Exit. A process that has finished

Processes

A process can be in one of several *states*. In a simplified model, the five main states are

1. New. A process that has just been created
2. Running. It is currently executing on the CPU
3. Ready. It is ready to run, but not currently running as some other process is currently using the CPU
4. Blocked. Can't run right now as it is waiting for some event or resource to become available. E.g., waiting for a block of data to arrive from the disk
5. Exit. A process that has finished

Real OSs will have more states than this, but these are the important ones

Processes

We shall assume, for simplicity, that we have just one processor

Processes

We shall assume, for simplicity, that we have just one processor

The OS will have sets of processes in each state, so the scheduling decision is making the choice of which process to move between which states

Processes

We shall assume, for simplicity, that we have just one processor

The OS will have sets of processes in each state, so the scheduling decision is making the choice of which process to move between which states

In real OSs, these sets will will not be simple lists. They might be arranged in priority order, or might be some more sophisticated datastructure: e.g., a pair of lists, one for real-time processes and the other for non-real-time; or a tree

Processes

Example: in Unixes, processes are arranged in *trees*

```
systemd-+-ModemManager---2*[{ModemManager}]
        |-NetworkManager---2*[{NetworkManager}]
        |-Thunar---3*[{Thunar}]
        |-accounts-daemon---2*[{accounts-daemon}]
        |-agetty
        |-atd
        |-auditd---{auditd}
        |-avahi-daemon
        |-chrome-+-2*[cat]
        |        |-chrome-+-chrome-+-chrome---12*[{chrome}]
        |        |        |        |-chrome---19*[{chrome}]
        |        |        |        |-3*[chrome---11*[{chrome}]]
        |        |        |        |-chrome---15*[{chrome}]
        |        |        |        |-chrome---17*[{chrome}]
        |        |        |        |-chrome---16*[{chrome}]
        |        |        |        |-chrome---10*[{chrome}]
        |        |        |        `-chrome---23*[{chrome}]
        |        |        `-nacl_helper
        |        |-chrome-+-chrome
        |        |        `-7*[{chrome}]
```

Processes

Trees allow easy manipulation of whole bunches of (usually related) processes in a simple way

Processes

So we have these five main states: New, Ready, Running, Blocked and Exit, and a process will be moved by the OS between them

Processes

So we have these five main states: New, Ready, Running, Blocked and Exit, and a process will be moved by the OS between them

A new process will begin in the state New

Processes

So we have these five main states: New, Ready, Running, Blocked and Exit, and a process will be moved by the OS between them

A new process will begin in the state New

A process just finished will be in the state Exit

Processes

So we have these five main states: New, Ready, Running, Blocked and Exit, and a process will be moved by the OS between them

A new process will begin in the state New

A process just finished will be in the state Exit

In between the OS must decide, as part of its scheduling, where to place each process

Processes

So we have these five main states: New, Ready, Running, Blocked and Exit, and a process will be moved by the OS between them

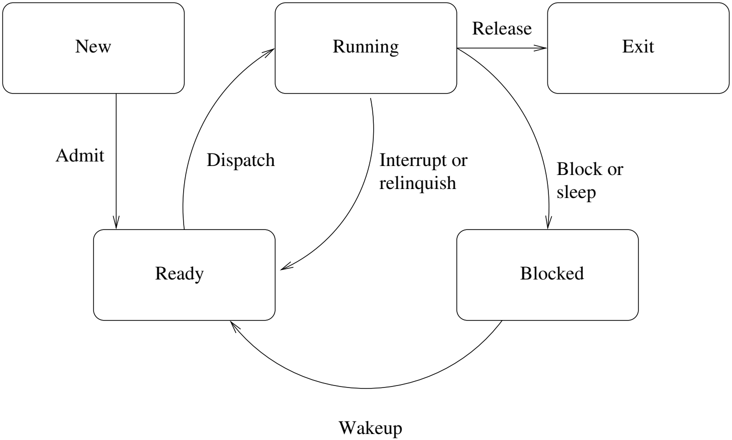
A new process will begin in the state New

A process just finished will be in the state Exit

In between the OS must decide, as part of its scheduling, where to place each process

There is a standard *finite state machine* that describes the allowed transitions between states

Processes



Process State Transitions

Processes

A typical transition is

Processes

A typical transition is

1. The OS decides to schedule a process on the ready list

Processes

A typical transition is

1. The OS decides to schedule a process on the ready list
2. The process is *dispatched*, i.e., the OS marks its state as running and starts executing it (jump and drop privilege)

Processes

A typical transition is

1. The OS decides to schedule a process on the ready list
2. The process is *dispatched*, i.e., the OS marks its state as running and starts executing it (jump and drop privilege)
3. The process may choose to voluntarily suspend itself: *relinquish* (e.g., a clock program displaying the time might suspend itself for a minute)

Processes

A typical transition is

1. The OS decides to schedule a process on the ready list
2. The process is *dispatched*, i.e., the OS marks its state as running and starts executing it (jump and drop privilege)
3. The process may choose to voluntarily suspend itself: *relinquish* (e.g., a clock program displaying the time might suspend itself for a minute)
4. Or an interrupt may arise, e.g., from a packet arriving on the network card, or a key being hit on the keyboard

Processes

A typical transition is

1. The OS decides to schedule a process on the ready list
2. The process is *dispatched*, i.e., the OS marks its state as running and starts executing it (jump and drop privilege)
3. The process may choose to voluntarily suspend itself: *relinquish* (e.g., a clock program displaying the time might suspend itself for a minute)
4. Or an interrupt may arise, e.g., from a packet arriving on the network card, or a key being hit on the keyboard
5. Or a timer interrupt may arise. In any of these three cases the OS moves the process to the Ready state

Processes

6. Or the running process may need some resource the OS must supply (e.g., for disk access) so it does a syscall and must wait until the resource is ready (e.g., the disk returns some data); the OS moves it to Blocked

Processes

6. Or the running process may need some resource the OS must supply (e.g., for disk access) so it does a syscall and must wait until the resource is ready (e.g., the disk returns some data); the OS moves it to Blocked
7. In the case of a blocked process, perhaps data has returned from the disk and the process can *wake up* and become Ready again. Note that the process won't necessarily start running immediately, it is just ready to run when it gets its chance

Processes

6. Or the running process may need some resource the OS must supply (e.g., for disk access) so it does a syscall and must wait until the resource is ready (e.g., the disk returns some data); the OS moves it to Blocked
7. In the case of a blocked process, perhaps data has returned from the disk and the process can *wake up* and become Ready again. Note that the process won't necessarily start running immediately, it is just ready to run when it gets its chance

And to make it clear: it's not the processes moving themselves between the states, it's the OS moving them between the sets of processes in each state

Processes

Remember, early OSs without timer interrupts had to rely on processes relinquishing control every once in a while: cooperative multitasking

Processes

Remember, early OSs without timer interrupts had to rely on processes relinquishing control every once in a while: cooperative multitasking

Even though we can now have preemptive multitasking, processes may wish to relinquish voluntarily

Processes

Remember, early OSs without timer interrupts had to rely on processes relinquishing control every once in a while: cooperative multitasking

Even though we can now have preemptive multitasking, processes may wish to relinquish voluntarily

Exercise Write a program that voluntarily relinquishes occasionally

Processes

New and Exit states happen just once per process

Processes

New and Exit states happen just once per process

- New. For a process just created, perhaps code and data are not yet loaded into memory. The OS datastructures needed to manage the process must be created and filled in

Processes

New and Exit states happen just once per process

- New. For a process just created, perhaps code and data are not yet loaded into memory. The OS datastructures needed to manage the process must be created and filled in
- Exit. For a process that has just finished. Some tidying up is usually needed after a process ends, such as closing files or reclaiming memory or other resources it used

Processes

A real example:

```
USER        PID  PPID PRI %CPU %MEM STAT      TIME COMMAND
rjb        3974  4831  22  0.0  0.1 R+    00:00:00 ps
rjb        4495  4831  24  0.0  2.0 S     00:01:11 emacs
rjb        4538  4530  23  0.0  0.2 Ss+   00:00:00 bash
rjb        4540  4534  24  0.0  0.2 Ss    00:00:00 bash
rjb        4664  4556  21  0.0  0.6 S+    00:00:08 pine
rjb        4831  4829  24  0.0  0.2 Ss+   00:00:00 bash
rjb        7839  4831  15  0.0  0.1 Ss    00:00:00 firefox
rjb        7851  7839  14  0.0  0.1 S     00:00:00 run-mozilla.sh
rjb        7856  7851  24  0.2 16.6 Sl    00:31:47 firefox-bin
rjb       14880     1  16  0.0  3.1 Dsl   00:06:43 recollindex
```

Example processes under Linux

Processes

- S. Sleeping: like blocked (interruptible sleep; waiting for an event like a timer or other interrupt)
- D. Disk wait (uninterruptible sleep; waiting for requested I/O)
- R. Running or ready to run
- It is hard to catch new and exiting processes

(s: session leader; +: foreground process group; l: multithreaded)

Processes

Other columns of interest

Processes

Other columns of interest

- User. The user who owns the process

Processes

Other columns of interest

- User. The user who owns the process
- PRI. Priority. In Linux, priorities are integers, larger indicates less important

Processes

Other columns of interest

- User. The user who owns the process
- PRI. Priority. In Linux, priorities are integers, larger indicates less important
- PID. Process identifier. An integer that uniquely identifies this process

Processes

Other columns of interest

- User. The user who owns the process
- PRI. Priority. In Linux, priorities are integers, larger indicates less important
- PID. Process identifier. An integer that uniquely identifies this process
- PPID. Parent PID. The PID of the process that started this process. This allows processes to be grouped in trees. Process number 1 is the parent of all processes

Processes

Other columns of interest

- User. The user who owns the process
- PRI. Priority. In Linux, priorities are integers, larger indicates less important
- PID. Process identifier. An integer that uniquely identifies this process
- PPID. Parent PID. The PID of the process that started this process. This allows processes to be grouped in trees. Process number 1 is the parent of all processes
- CPU, MEM, TIME. How much of these resources this process is using

Processes

So we can see some more of the information that a process needs to collect and maintain:

Processes

So we can see some more of the information that a process needs to collect and maintain:

- User identifiers (userids)

Processes

So we can see some more of the information that a process needs to collect and maintain:

- User identifiers (userids)
- A priority

Processes

So we can see some more of the information that a process needs to collect and maintain:

- User identifiers (userids)
- A priority
- Statistics like memory and CPU used

Processes

So we can see some more of the information that a process needs to collect and maintain:

- User identifiers (userids)
- A priority
- Statistics like memory and CPU used
- The scheduling state

Processes

So we can see some more of the information that a process needs to collect and maintain:

- User identifiers (userids)
- A priority
- Statistics like memory and CPU used
- The scheduling state

But there are still more that will become clearer as we go along

Processes

So we can see some more of the information that a process needs to collect and maintain:

- User identifiers (userids)
- A priority
- Statistics like memory and CPU used
- The scheduling state

But there are still more that will become clearer as we go along

This collection of data a process needs is called the *process control block*, or PCB

Processes

To pause and restart a process (e.g., on an interrupt) requires the saving and restoring of all the running process' state: CPU registers, stack pointers, MMU flags, etc.

Processes

To pause and restart a process (e.g., on an interrupt) requires the saving and restoring of all the running process' state: CPU registers, stack pointers, MMU flags, etc.

This will also be stored in the PCB

Processes

To pause and restart a process (e.g., on an interrupt) requires the saving and restoring of all the running process' state: CPU registers, stack pointers, MMU flags, etc.

This will also be stored in the PCB

And will be retrieved from the PCB when the process next gets scheduled to run

Processes

Process creation is quite involved

Processes

Process creation is quite involved

- Allocate and create PCB structure

Processes

Process creation is quite involved

- Allocate and create PCB structure
- Find a free PID

Processes

Process creation is quite involved

- Allocate and create PCB structure
- Find a free PID
- Determine and allocate the necessary resources (in particular memory)

Processes

Process creation is quite involved

- Allocate and create PCB structure
- Find a free PID
- Determine and allocate the necessary resources (in particular memory)
- Determine the initial priority of the process

Processes

Process creation is quite involved

- Allocate and create PCB structure
- Find a free PID
- Determine and allocate the necessary resources (in particular memory)
- Determine the initial priority of the process
- Insert PCB into the relevant kernel list of PCBs

Processes

Process creation is quite involved

- Allocate and create PCB structure
- Find a free PID
- Determine and allocate the necessary resources (in particular memory)
- Determine the initial priority of the process
- Insert PCB into the relevant kernel list of PCBs

This is what happens in the New state; it can now be moved to Ready

Processes

Process creation is quite involved

- Allocate and create PCB structure
- Find a free PID
- Determine and allocate the necessary resources (in particular memory)
- Determine the initial priority of the process
- Insert PCB into the relevant kernel list of PCBs

This is what happens in the New state; it can now be moved to Ready

Again, the process might not start running immediately, as there could be some higher priority process that must run first

Processes

Most processes are created (*forked/spawned*) by other processes: of course, only the OS can actually create processes

Processes

Most processes are created (*forked/spawned*) by other processes: of course, only the OS can actually create processes

A user process that wants a new process will ask the OS to create one (using a syscall)

Processes

Processes use resources like memory and CPU, so the OS must be involved

Processes

Processes use resources like memory and CPU, so the OS must be involved

- A process decides it wants to start another process. E.g., a GUI process as a response to a user clicking on an icon

Processes

Processes use resources like memory and CPU, so the OS must be involved

- A process decides it wants to start another process. E.g., a GUI process as a response to a user clicking on an icon
- It calls the OS kernel (syscall), telling it what process it want to start (e.g., "start the browser program")

Processes

Processes use resources like memory and CPU, so the OS must be involved

- A process decides it wants to start another process. E.g., a GUI process as a response to a user clicking on an icon
- It calls the OS kernel (syscall), telling it what process it want to start (e.g., “start the browser program”)
- The OS can now create a new process according to the specifications given

Processes

Processes use resources like memory and CPU, so the OS must be involved

- A process decides it wants to start another process. E.g., a GUI process as a response to a user clicking on an icon
- It calls the OS kernel (syscall), telling it what process it want to start (e.g., “start the browser program”)
- The OS can now create a new process according to the specifications given
- The new process can now be scheduled

Processes

Of course, the OS can choose not to create the new process if some policy says not to, or there is not enough memory, or some other reason

Processes

Of course, the OS can choose not to create the new process if some policy says not to, or there is not enough memory, or some other reason

In that case, the originating process usually gets a message back from the OS (via the value returned from the syscall) explaining the problem

Processes

Of course, the OS can choose not to create the new process if some policy says not to, or there is not enough memory, or some other reason

In that case, the originating process usually gets a message back from the OS (via the value returned from the syscall) explaining the problem

For example, “no permission to exec that program”

Processes

Exercise The question arises: if processes are created by other processes, how do we get started? Read about the *bootstrapping problem*