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

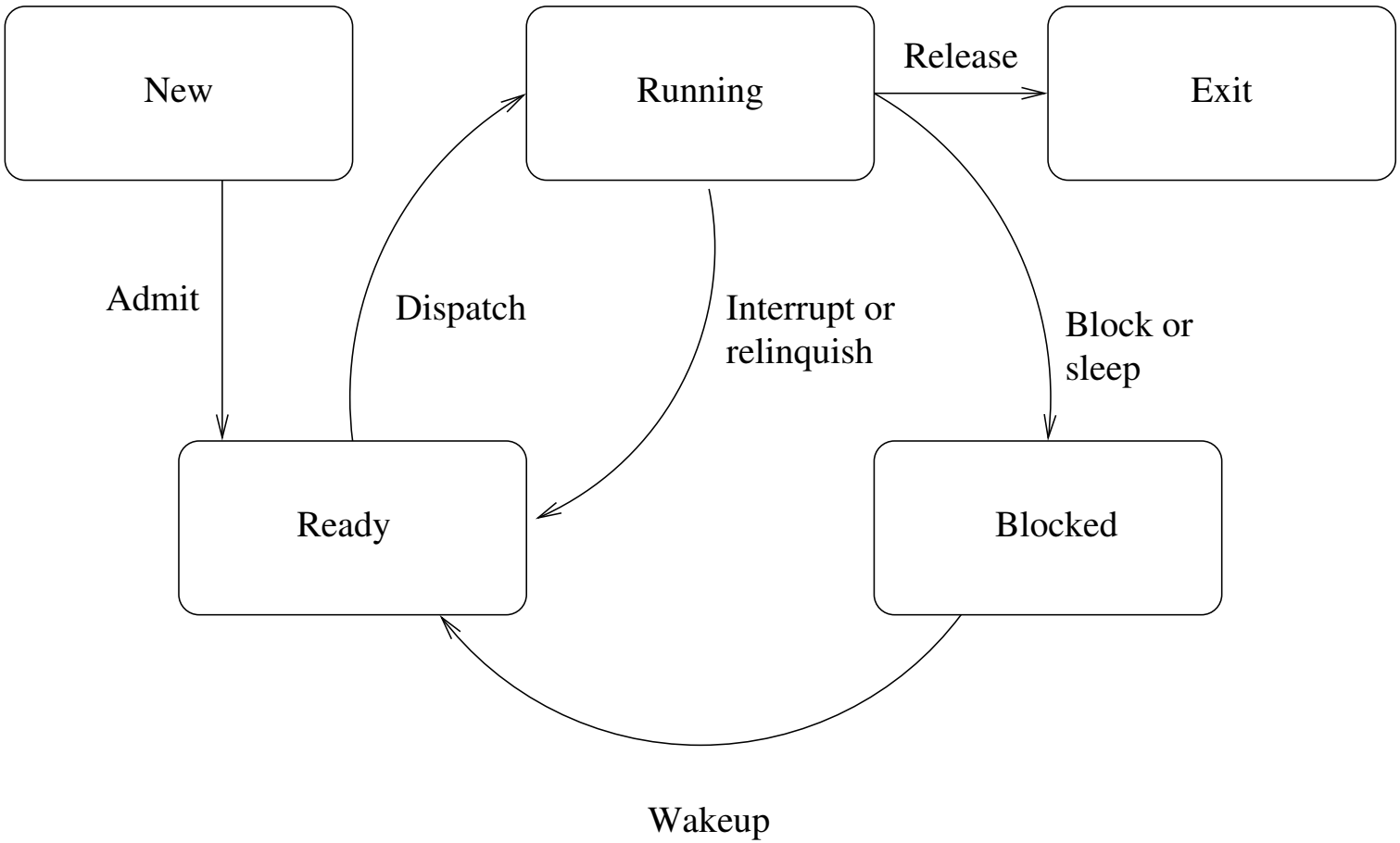
A new process will begin in the state New

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 happen when the process has used its *time slice*. 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 lists 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

User programs running on such OSs had to be explictly written to be cooperative

Processes

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

User programs running on such OSs had to be explictly written to be cooperative

And so were often not

Processes

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

User programs running on such OSs had to be explictly written to be cooperative

And so were often not

For example, Windows 3.1, MacOS 9

Processes

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

User programs running on such OSs had to be explictly written to be cooperative

And so were often not

For example, Windows 3.1, MacOS 9

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 have been 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 have been 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

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 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 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 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 the 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 the 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 the process state: CPU registers, stack pointers, MMU flags, etc.

This will also be stored in the PCB

So process handling is very similar to the way interrupts are handled