Processes

Operating System Lab
Spring 2015

Roya Razmnoush
CE & IT Department, Amirkabir University of Technology

Process Concept

- Programs are loaded into memory and run as individual **processes**. The operating system arranges to time-share the CPU(s) among all the processes so that each process appears to run independently and concurrently. Some programs can themselves split into multiple **threads**, each of which gets its own CPU resources.

What is a Process?

- Program
  - An executable file
- Process
  - An instance of a program that is being executed by the OS.
  - Every process has a unique ID (PID).

Process-Job-Task

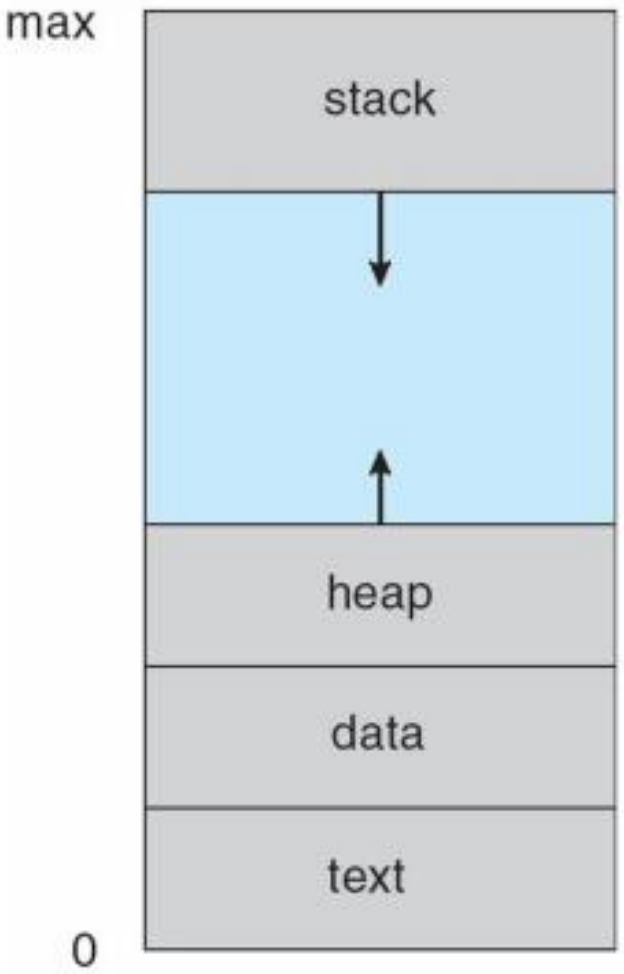
- An OS executes a variety of programs:
  - Batch systems – jobs
  - Time-shared systems – user programs or tasks

- Process – a program in (sequential) execution
- A process includes:
  - program counter (PC), stack, data section, …

Job vs. Process

- A process is any running program with its own address space.
- A job is a concept used by the shell - any program you interactively start that doesn't detach (ie, not a daemon) is a job. If you're running an interactive program, you can press CtrlZ to suspend it. Then you can start it back in the foreground (using fg) or in the background (using bg).
- While the program is suspended or running in the background, you can start another program - you would then have two jobs running. You can also start a program running in the background by appending an "&" like this: program &. That program would become a background job. To list all the jobs you are running, you can use jobs.
- It is called Job Control.

A process in MM



Process Control Block (PCB)

Information associated with each process:

- State
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

Process Control Block (PCB)

PID

- Every process has a unique ID (PID).
- PIDs are 16-bit numbers that are assigned sequentially by Linux as new processes are created.

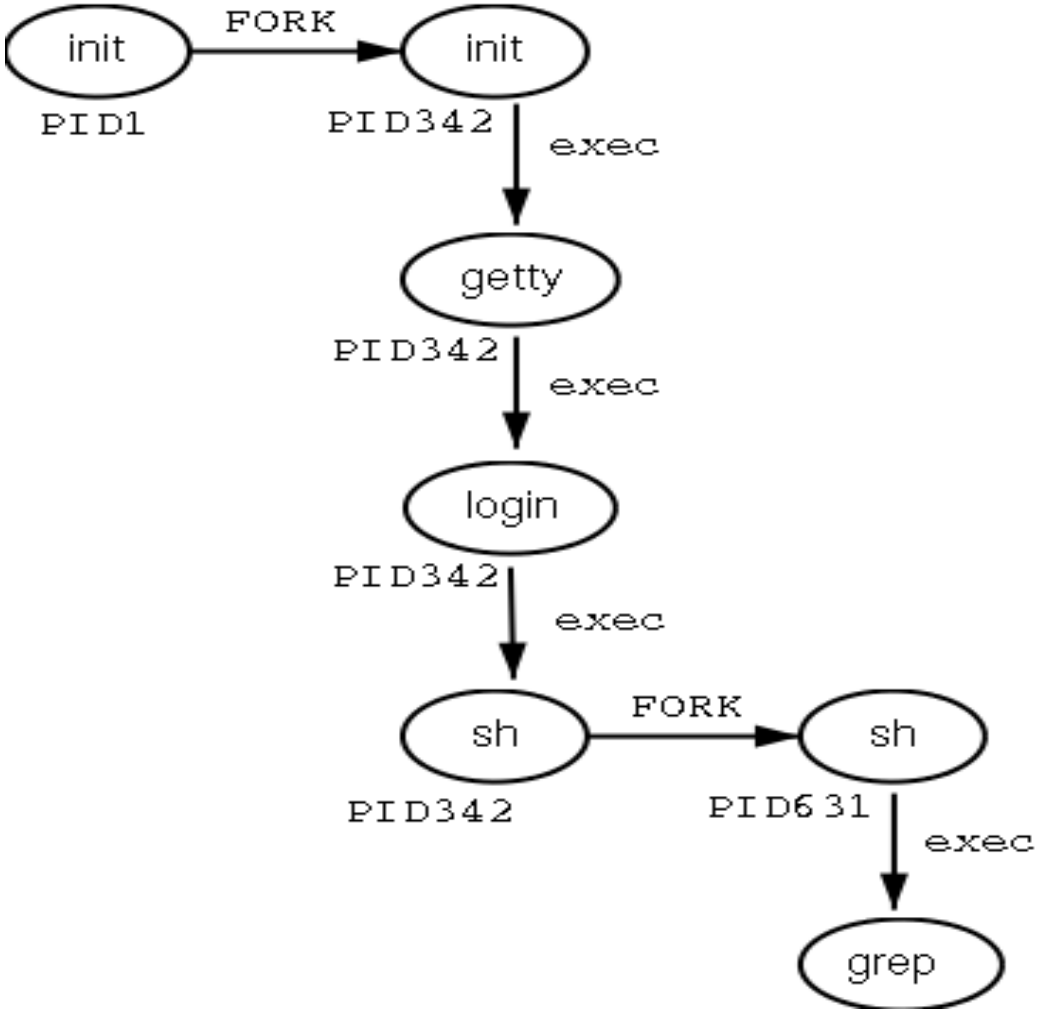
Process Owener

- A process usually runs with ("is owned by") the single userid and multiple group Permissions of the person who started it. (Recall that when you log in, you have one user permission and multiple group permissions.) Only the user who owns the process (and, of course, the root super-user) can terminate or adjust the priority of a process.

Process Parent

- Every process except the very first Unix process starts by **forking** itself from some **parent** process. A parent process may fork off multiple child processes, which may themselves fork off more child processes. This gives processes a tree-like hierarchical structure of multiple generations of parents and children.
- Each process has a numeric *process ID* or **PID** and a *parent process ID* or **PPID**.

Process Parent



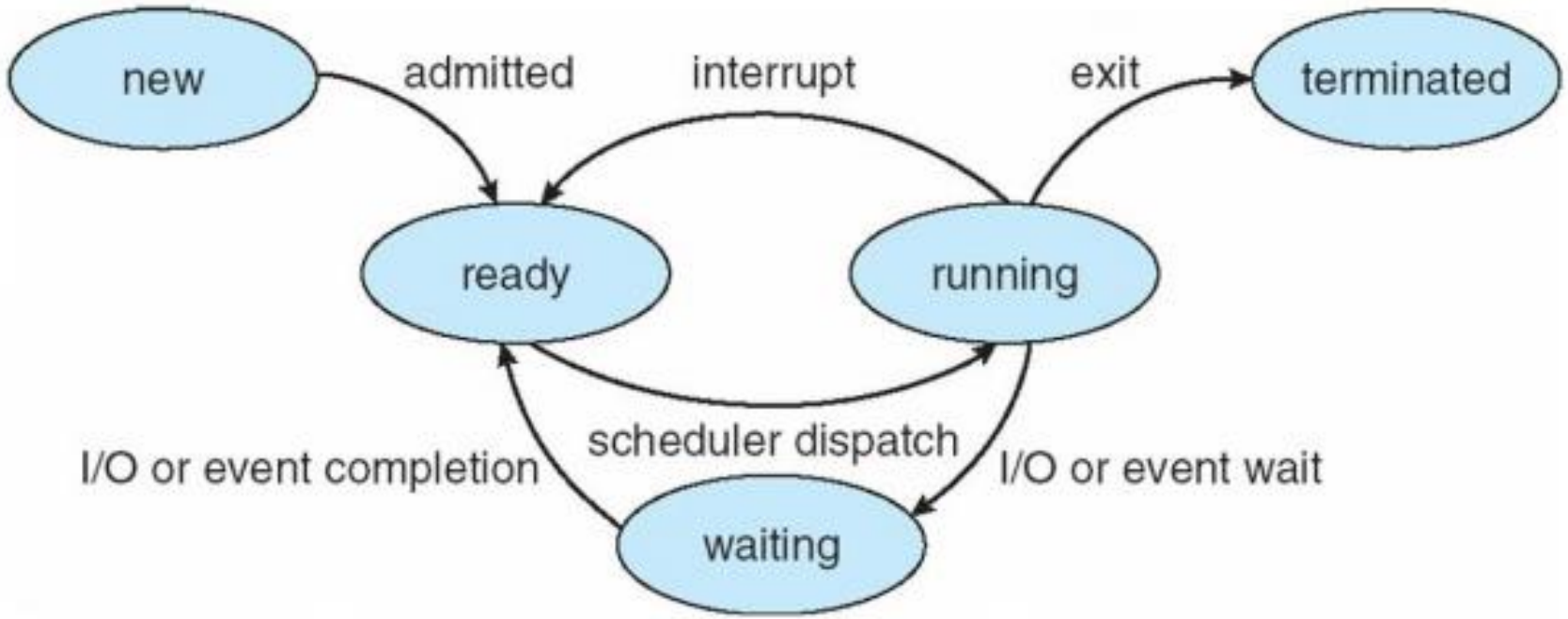
Process Management

- The Unix OS is a time-sharing system.
- Each process is represented by a task_struct data structure.
- The task_vector is an array of pointers to every task_struct data structure in the system.

Process State

- As a process executes, it changes *state*
  - new - the process is being created
  - running - instructions are being executed
  - waiting - the process is waiting for some event to occur
  - ready - waiting to be assigned to a processor
  - terminated - the process has finished execution

Diagram of process state



Process Statuses

- Running
  - The process is either running or it is ready to run.
- Waiting
  - The process is waiting for an event or for a resource.
- Stopped
  - The process has been stopped, usually by receiving a signal.
- Zombie
  - This is a halted process which, for some reason, still has a task_struct data structure in the task vector.
  - "A child process that terminates but is never waited on by its parent becomes a zombie process.
  - When a process dies, its child processes all become children of process number 1, which is the init process. Init is ``always'' waiting for children to die, so that they don't remain as zombies.

Type of Processes

- Interactive Process (jobs)
  - Initiated from (and controlled by) a shell
- Daemon Process
  - Run in the background until required

Viewing active processes

- ps displays the processes controlled by the terminal in which ps is invoked.

```
% ps
  PID TTY          TIME CMD
21693 pts/8    00:00:00 bash
21694 pts/8    00:00:00 ps
```

```
% ps -e -o pid,ppid,command
```

- -e : display all processes running on the system
- -o pidd.ppidd,command : what information to show about each process

ps command

- ps
  - Report process status.

  $ ps l 18347

  F UID PID PPID PRI NI VSZ RSS WCHAN STAT TTY TIME COMMAND

  4 0 18347 18345 20 0 22120 2192 wait S pts/2 0:00 /bin/bash

  - When a parent process exits (terminates) before its children, the children are orphaned and are passed to be children of Unix process Number One (1), traditionally named init.

  -The Linux ps command options are a crazy mix of the BSD ps command options (no dashes), the incompatible SystemV UNIX ps command options (with dashes), plus some GNU extension options (with double-dashes). From man ps on Linux.

ps command

```
$ ps         # BSD: some of your processes (in current terminal window)
$ ps x       # BSD: all of your processes in all windows, everywhere
$ ps xl      # BSD: all your processes, long format
$ ps xlww    # BSD: all your processes, long format, full wide listing
$ ps ax      # BSD: all processes for all users, numerid UID
$ ps auxww   # BSD: all processes for all users, text userid, full wide listing
$ ps laxww   # BSD: all processes, all users, long format, full wide listing
$ ps f       # BSD: ascii art hierarchical display (forest)

$ ps -e      # UNIX: all processes
$ ps -elww   # UNIX: all processes, long format, wide listing, numeric UID
$ ps -efww   # UNIX: same but with text userid instead of numeric

$ pstree     # summary process tree command; similar to "ps axf" but less detail
```

Job Control

- pstree
  - Display a tree of processes.
- top
  - Display top CPU processes.
- jobs
  - List the active jobs.

Process related commands

Related Commands

- nice
  - Run a program with modified scheduling priority.
- renice
  - Alter priority of running process.
- kill
  - Send signal to a process.

Terminate or Pause Process - kill

Processes can be paused (stopped) or terminated by their owners (and `root`) by sending a **signal** to the process id using the poorly-named `kill` command:

```
kill [ -SIG ] pid ...              # SIG is the signal name to send
```

Most systems have an actual executable `kill` program in one of the `bin` directories, but `kill` is usually also built-in command to each shell so that it can use some shell-specific syntax.

For a list of signals that you can send to processes, type `kill -l`. Most modern shells allow signals by name, e.g. `kill -KILL` instead of the old numeric way `kill -9` where the numbers didn't always mean the same thing on different systems.

Terminate or Pause Process - kill

The default signal sent by `kill` is `TERM` (**Terminate**), and it allows a process to clean up before terminating. (For example, `vim` will save its buffers in a `*.swp` recovery file before exiting.) If `TERM` doesn't kill a process, other signals to try are `HUP` and finally `KILL`:

```
$ kill 123          # default is TERM signal - same as kill -TERM 123
$ kill -HUP 123     # stronger signal (HUP = Hangup)
$ kill -KILL 123    # or kill -9 123 - strongest signal - avoid using it
```

The safest way to signal a specific process is by using `ps` to find the process **PID** and then using `kill` to send a signal to that **PID**. Some systems come with a `killall` command that can also kill processes by name (see the man page before you try this):

```
$ killall [ -SIG ] name ...      # SIG is the signal to send to name
```

Note that the `killall` command is dangerous, since by killing processes based on name it might match unintended processes with the same name, or (if `root`) processes with the same name run by other users. (If an attacker decides to name an attacking program `bash`, then `killall -9 bash` would not be a good thing to type on your system.)

Signals

Signal Name	Signal Number	Description
SIGHUP	1	Hang up detected on controlling terminal or death of controlling process
SIGINT	2	Issued if the user sends an interrupt signal (Ctrl + C).
SIGQUIT	3	Issued if the user sends a quit signal (Ctrl + D).
SIGFPE	8	Issued if an illegal mathematical operation is attempted
SIGKILL	9	If a process gets this signal it must quit immediately and will not perform any clean-up operations
SIGALRM	14	Alarm Clock signal (used for timers)
SIGTERM	15	Software termination signal (sent by kill by default).

List of signals:
$ kill -l

nohup command

- Runs a command, making it immune to any **HUP** (hangup) signals while it is running
- If standard input is a terminal, **nohup** redirects it from **/dev/null**. If standard output is a terminal, append output to "**nohup.out**" if possible, "**$HOME/nohup.out**" otherwise. If standard error is a terminal, redirect it to standard output. To save output to file *FILE*, use "**nohup** *COMMAND* **>** *FILE*".
- nohup find -size +100k > log.txt
- Run the **find** command, instructing it to search for any file bigger then 100k. **find** will continue to search regardless of the user's connection to the terminal, and log results to the file **log.txt**.

at command

- at - execute commands at a later time.
- Scheduled task with at command.
- Use for set the reboot/shutdown time.
- sudo reboot | at 12:30 pm
- sudo At 12:30 pm

>at: reboot

> EOT

- You can also use “sudo shutdown +30 ” and “sudo shutdown 17:30”.

at command

- Examples: (it's very flexible!!!)

At 08:15 am Jan 24

At 8:15amjan24

At now " +1day"

At 5 pm FRIday

At now + 1 minute

At 2pm + 1 week

At 2 pm next week

Process Scheduling

- Linux schedules the parent and child processes independently
- Linux promises that each process will run eventually
- Alter execution priority:
  - By default, every process has a niceness of zero.
  - A higher niceness value means that the process is given a lesser execution priority;
  - conversely, a process with a lower (that is, negative) niceness gets more execution time.
- $ nice –n value command
- renice command to change the niceness of a running process

nice & renice commands

- Every running process in Unix has a priority assigned to it.
- You can change the process priority using nice and renice utility.
- Nice command will launch a process with an user defined scheduling priority.
- Renice command will modify the scheduling priority of a running process.
- Linux Kernel schedules the process and allocates CPU time accordingly for each of them. But, when one of your process requires higher priority to get more CPU time, you can use nice and renice command.
- The process scheduling priority range is from -20 to 19. We call this as nice value.

A nice value of -20 represents highest priority, and a nice value of 19 represent least priority for a process.

- By default when a process starts, it gets the default priority of 0.
- The current priority of a process can be displayed using ps command.
- The “NI” column in the ps command output indicates the current nice value (i.e priority) of a process.

nice & renice commands

- **Launch a program with high priority:**

$ nice -10 perl test.pl

$ ps -fl -C "perl test.pl"
F S UID PID PPID C PRI NI ADDR SZ WCHAN STIME TTY TIME CMD 0 R
Bala 7044 6424 99 90 10 - 1556 - 13:58 pts/3 00:00:03 perl test.pl

- **Launch a program with less priority**

nice --10 perl test.pl
ps -fl -C "perl test.pl"
F S UID PID PPID C PRI NI ADDR SZ WCHAN STIME TTY TIME CMD 4 R
root 3534 3234 99 70 -10 - 1557 ? 19:06 pts/1 00:00:56 perl test.pl

Note: Regular users are not allowed to launch a program with a higher priority. Only root user is allowed to launch a program with high priority.

nice & renice commands

- **Change the priority with option –n:**

- # nice -n -5 perl test.pl

- # nice -n 5 perl test.pl

- **Change the priority of a running process:**

- # ps -fl -C "perl test.pl“

F S UID PID PPID C PRI NI ADDR SZ WCHAN STIME TTY TIME CMD
4 R root 3534 3234 99 70 -10 - 1557 ? 19:06 pts/1 00:00:56 perl test.pl

- # renice -n -19 -p 3534

- # ps -fl -C "perl test.pl"

F S UID PID PPID C PRI NI ADDR SZ WCHAN STIME TTY TIME CMD 4 R
root 3534 3234 99 70 -19 - 1557 ? 19:06 pts/1 00:00:56 perl test.pl

nice & renice commands

- **Change the priority of all processes that belongs to a group:**

renice -n 5 -g geekstuff

- **Change the priority of all processes owned by user:**

renice -n 5 -u bala

ps -fl -C "perl“
F S UID PID PPID C PRI NI ADDR SZ WCHAN STIME TTY TIME CMD
0 R bala 2720 2607 99 85 5 - 1556 - 14:34 pts/2 00:05:07 perl test.pl
0 R bala 2795 2661 99 85 5 - 1556 - 14:39 pts/3 00:00:09 perl 2.pl

Linux shell job control & background processes

Normally when you run a command, the shell waits until the command is finished before it prompts you to enter the next command. This waiting for the process to finish is called a shell "foreground" process or job. The foreground job is attached to your keyboard and screen (unless you have redirected input or output):

```
$ sleep 10        # shell waits until sleep finishes before next prompt
$
```

You can interrupt most foreground jobs using the `^C` (**Ctrl-C**) key that sends a `SIGINT` (Interrupt) signal to the foreground process being run:

```
$ sleep 10
^C
$
```

If you want to run a command "in the background", without having the shell wait for it to finish, end the command with an ampersand `&`, e.g.

```
$ sleep 99 &
[1] 18920
$
```

The `&` starts the job without requiring the shell to wait for it. The shell gives you a prompt for your next command immediately. The background command runs independently; it can't be interrupted by the `^C` interrupt signal from the keyboard.

Modern Unix/Linux shells often provide some added syntax that lets you manage background processes that are forked from your current shell. Processes forked from the current shell are called the `jobs` of the shell. You can see these jobs using the shell built-in command of the same name:

```
$ jobs            # show child processes *only of the current shell*
[1]+  Stopped  sudo -s  (wd: ~)

$ jobs -l         # use -l to show the process IDs as well
[1]+ 12345 Stopped  sudo -s  (wd: ~)
```

Jobs of the current shell can be specified by short job numbers instead of process IDs to shell built-in commands such as `kill`:

```
$ kill %1         # send SIGTERM to job number 1 of this shell
```

Moving jobs backgroung/foreground

If you have already typed a command and forgot to use the `&`, you can put a foreground job into the background by typing `^Z` (**Ctrl-Z**) to send the job a signal to pause (stop or suspend) the job, followed by `bg` to put it into the background:

```
$ sleep 99
^Z
[1]+  Stopped                 sleep 99
$ bg
[1]+ sleep 99 &
```

You can bring a background job into the foreground, so that the shell waits for it again (and so `^C` can interrupt it) using `fg`:

```
$ jobs
[1]+  Running                 sleep 99
$ fg
sleep 99
```

Output from / input to background jobs

Any output produced by a background job will appear on your terminal screen, mixing on your screen with whatever else you are reading or typing at the time. In the example below, the user started to type an `echo` command line right after starting the background job, and you can see the output from the background job appearing on the screen in the middle of typing the command:

```
$ ( sleep 3 ; date ) &        # sleep a bit then run date, in background
[1] 17619
$ echo my mother is aSat Nov  8 04:07:30 EST 2014
 nice person
my mother is a nice person
[1]+  Done                    ( sleep 3; date )
$
```

Output from / input to background jobs

You can type `^L` at the shell to clear your screen of output from background jobs, and `^L` also works to clear your screen in `vim` command mode as well as in `less` and `more`.

If a background job tries to read from your keyboard, the system will pause (stop or suspend) that job until you make it a foreground job again:

```
$ wc &
[1] 1538
$ jobs -l
[1]+  1538 Stopped (tty input)     wc
$ fg
wc
The wc command is now foreground and reading my keyboard.
^D
1  10  59
$
```

The shell ensures that only one job is "foreground" and reading your keyboard. All other jobs that try to read the keyboard will be paused (`stopped`). When the shell isn't running any commands, the shell has your keyboard.

Sending signals to jobs and processes

Your shell can send signals, including stop and termination signals, to jobs of the *current* shell using job numbers instead of process numbers:

```
$ kill %1
[1]+  Terminated              sleep 99
```

To send signals to processes or jobs that are not started from the current shell, you first need to use `ps` to find their process numbers.

Keyboard signals. ^C , ^Z, ^\

Your terminal is configured to turn some characters you type into signals sent to foreground processes, e.g. `^c` sends an **Interrupt** signal to the current foreground process, just as if `kill -INT` had been used:

```
$ sleep 10
^C
$
```

Other control characters send other signals to foreground jobs:

```
Ctrl-C - ^C - send Interrupt signal (SIGINT)
Ctrl-Z - ^Z - send Stop signal (SIGSTOP)
Ctrl-/ - ^\ - send Quit signal (SIGQUIT)
```

You can change which characters send these signals using the `stty` command.

Related Commands

- bg
  - Place a job in background (similar to &);
- fg
  - Place a job in foreground.
- Ctrl+z
  - Stopped a process.
- Ctrl+c
  - Terminate a process.

References