COMP 311 Linux Operating System Lab

Lab 7: *Job and Process Management*

Ahmed Tamrawi

 atamrawi  atamrawi.github.io  ahmedtamrawi@gmail.com

جامعة بيرزيت
BIRZEIT UNIVERSITY

Computer Science Department

Linux OS Laboratory Manual (V 1.2)

COMP311

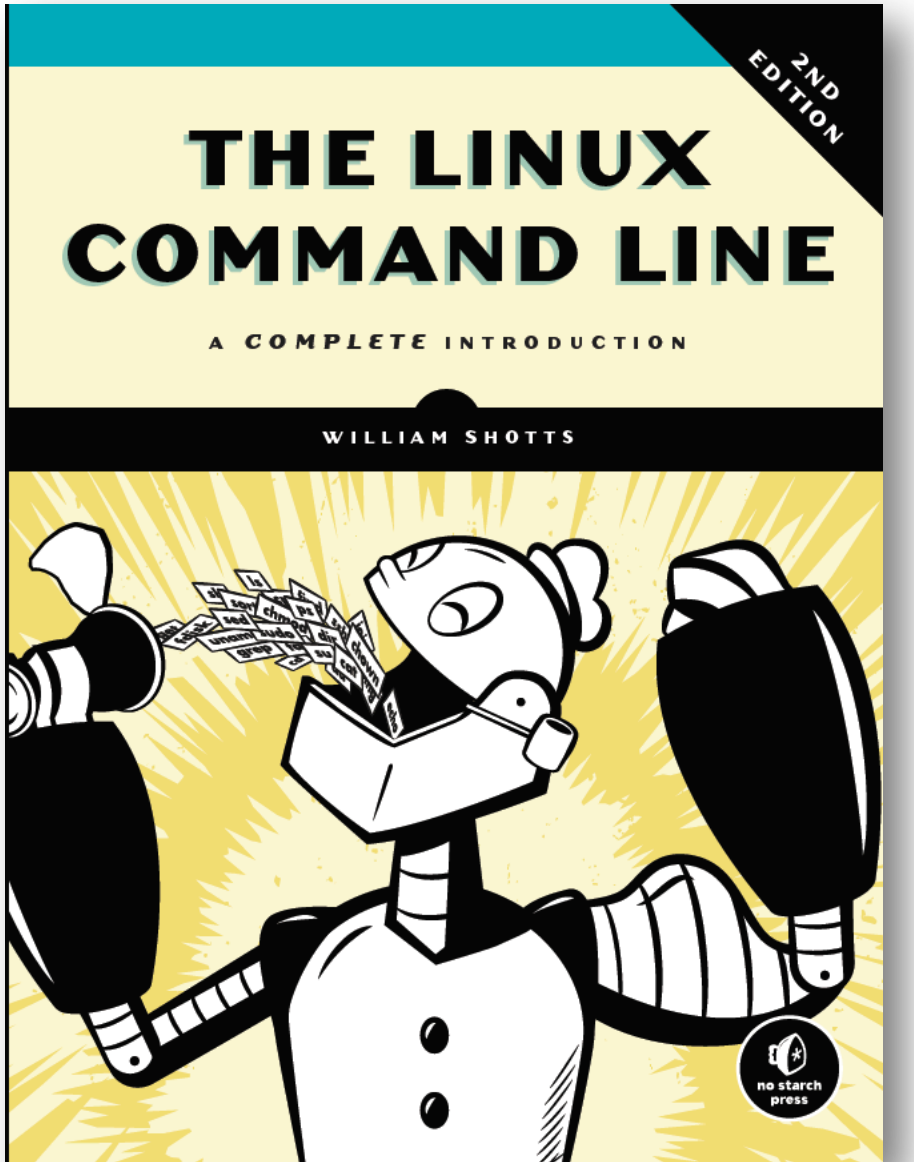
Nael I. Qaraeen

Approved By:

Computer Science department

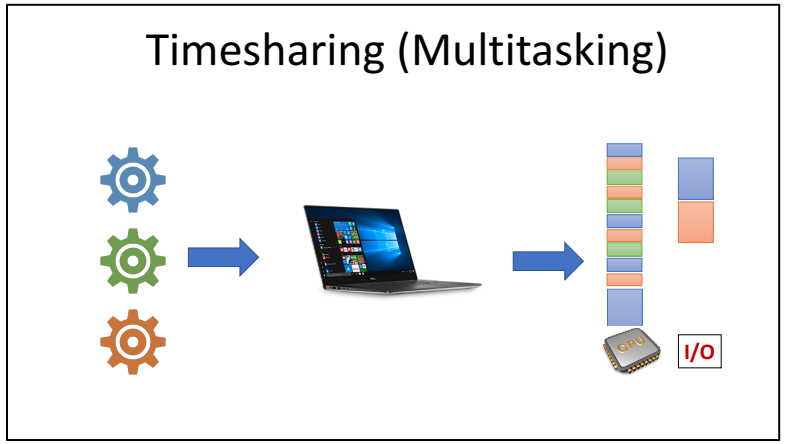
May 2015

+



Processes

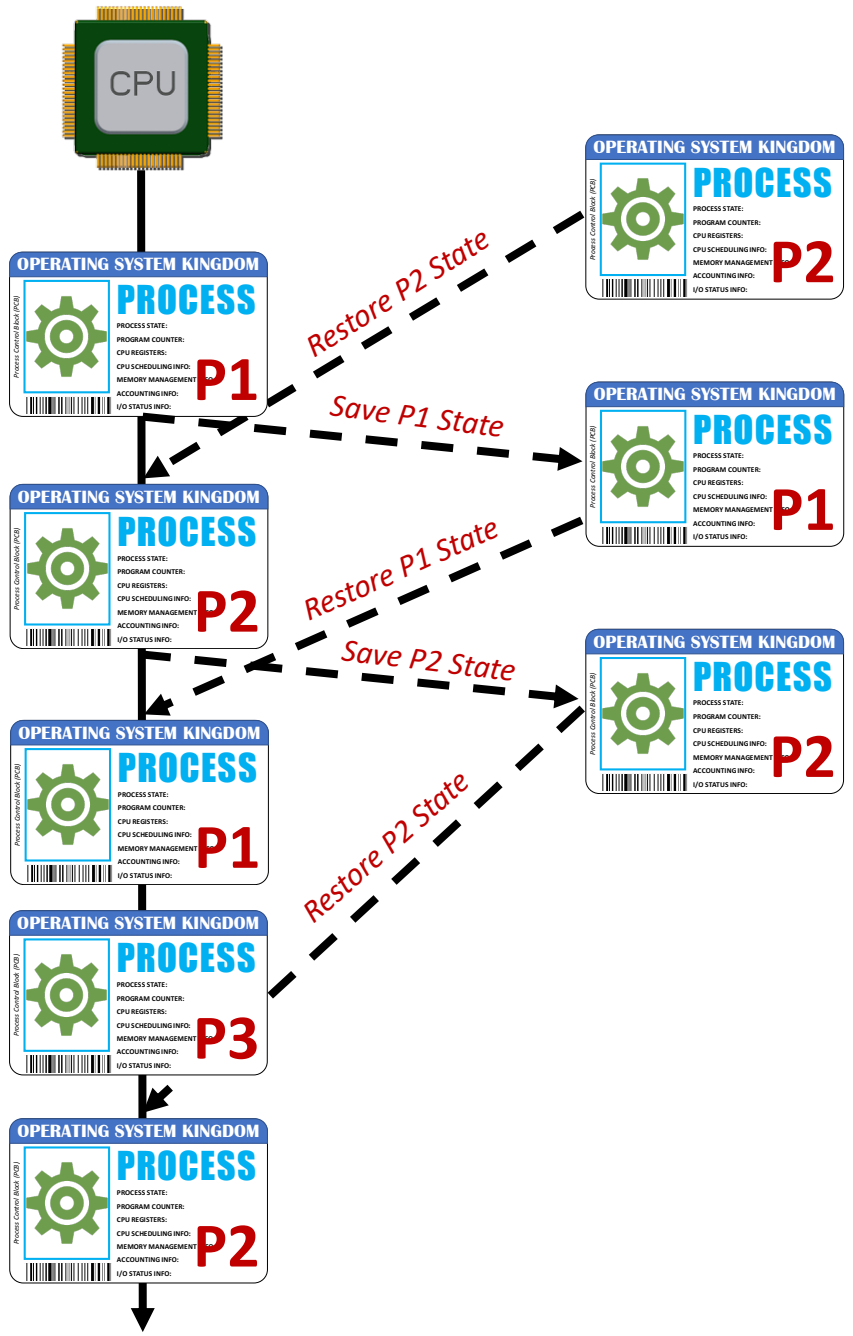
- Modern operating systems are usually **multitasking**, meaning they create the *illusion of doing more than one thing at once* by rapidly **switching from one executing program to another**. The kernel manages this through the use of processes.
- Processes are how Linux organizes the different programs waiting for their turn at the CPU.



Context Switching

enables multiple processes to share a single CPU

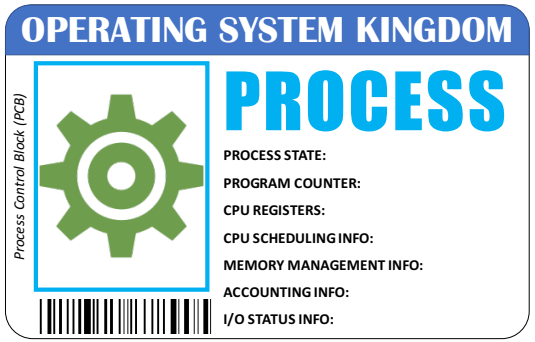
The mechanism to store and restore **the state or context** of a CPU in **Process Control Block** so that a process execution can be resumed from the same point later.



How a Process Works?

- When a system starts up, the kernel **initiates** a few of its own activities as processes and launches a program called `init`. `init`, in turn, runs a series of shell scripts (located in `/etc`) called `init` scripts, which start all the **system services**.
- Many of these services are implemented as **daemon** programs, programs that just sit in the background and do their thing without having any user interface. So, even if we are not logged in, the system is at least a little busy performing routine stuff.

Process identified and managed via a process identifier (PID) – **Unique ID**




```
[root@linoxide ~]# pstree
systemd─┬─NetworkManager─┬─dhclient
        │                └─3*[{NetworkManager}]
        ├─2*[agetty]
        ├─auditd───{auditd}
        ├─avahi-daemon───avahi-daemon
        ├─chronyd
        ├─crond
        ├─dbus-daemon
        ├─iprdump
        ├─iprinit
        ├─iprupdate
        ├─polkitd───5*[{polkitd}]
        ├─rsyslogd───2*[{rsyslogd}]
        ├─sshd─┬─sshd───bash───pstree
        │      └─sshd───sshd
        ├─systemd-journal
        ├─systemd-logind
        ├─systemd-network
        ├─systemd-udevd
        └─tuned───4*[{tuned}]
[root@linoxide ~]#
```

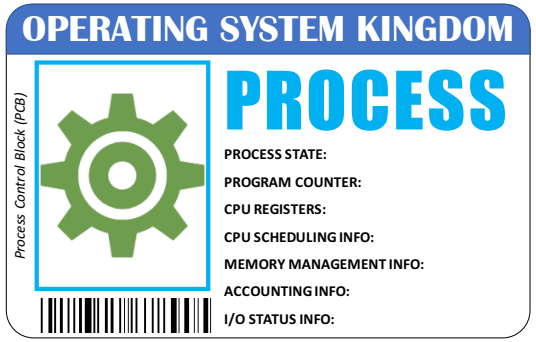
```
howtogeek@ubuntu: ~
top - 03:48:40 up 19 min,  1 user,  load average: 0.16, 0.09, 0.16
Tasks: 143 total,   1 running, 142 sleeping,   0 stopped,   0 zombie
Cpu(s):  2.6%us,  0.7%sy,  0.0%ni, 96.7%id,  0.0%wa,  0.0%hi,  0.0%si,
Mem:   1025656k total,   678580k used,   347076k free,    79936k buffer
Swap:        0k total,        0k used,        0k free,   310528k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1216 root      20   0 32624 3460 2860 S  0.7  0.3   0:05.31 vmtoolsd
 2025 howtogee  20   0 81456  23m  17m S  0.7  2.3   0:01.41 unity-2d-p
   17 root      20   0     0    0    0 S  0.3  0.0   0:00.34 kworker/0:
   36 root      20   0     0    0    0 S  0.3  0.0   0:00.10 scsi_eh_1
 1081 root      20   0  199m  60m 7340 S  0.3  6.0   0:13.42 Xorg
 1973 howtogee  20   0  6568 2832  916 S  0.3  0.3   0:06.24 dbus-daemo
 2153 howtogee  20   0  147m  16m 9820 S  0.3  1.7   0:03.63 unity-pane
 2313 howtogee  20   0  136m  13m  10m S  0.3  1.4   0:00.84 gnome-term
 2697 howtogee  20   0  2820 1148  864 R  0.3  0.1   0:00.05 top
    1 root      20   0  3456 1976 1280 S  0.0  0.2   0:02.31 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S  0.0  0.0   0:00.07 ksoftirqd/
```

First process to run is the "**init**" process that is started at **system boot**. This is the grand parent of all processes in the whole system

If a process dies, then its orphan children are re-parented to the "**init**" process

How a Process Works?

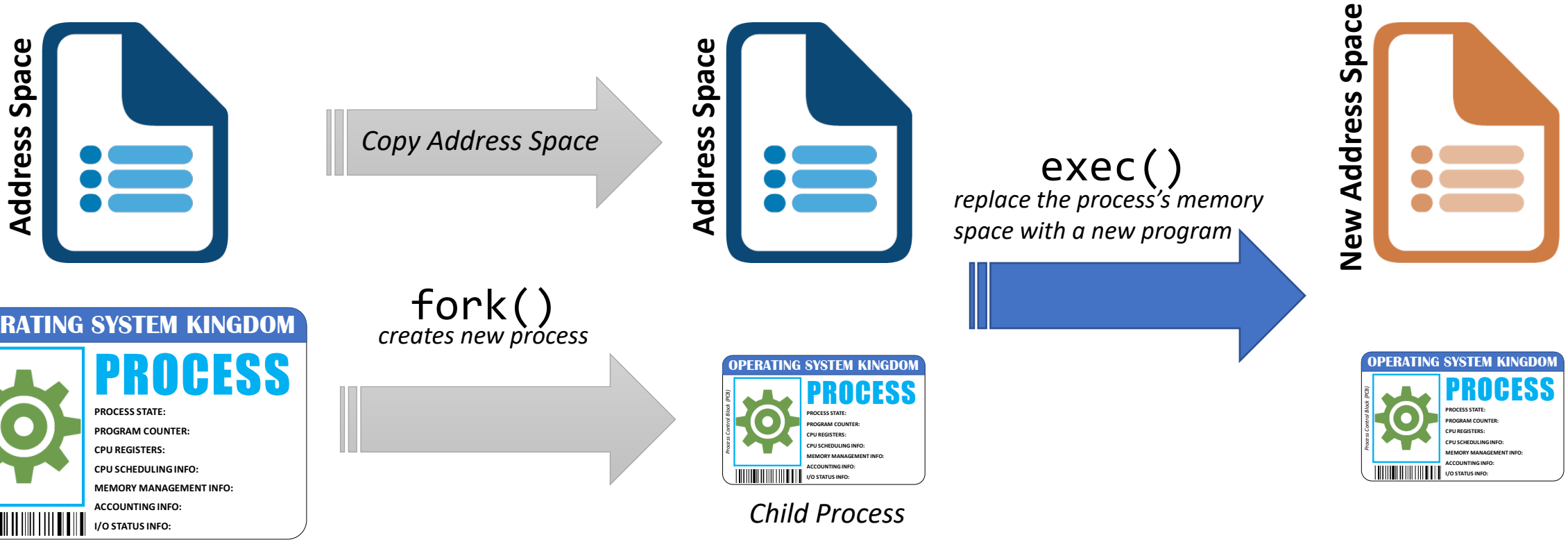


- The fact that a program can launch other programs is expressed in the process scheme as a **parent** process producing a **child** process.
- The kernel maintains information about each process to help keep things organized. For example, each process is assigned a number called a **process ID** (PID).
  - PIDs are assigned in ascending order, with `init` always getting PID 1.
- The kernel also keeps track of the memory assigned to each process, as well as the processes' readiness to resume execution.
- Like files, processes also have owners and user IDs, effective user IDs, and so on.

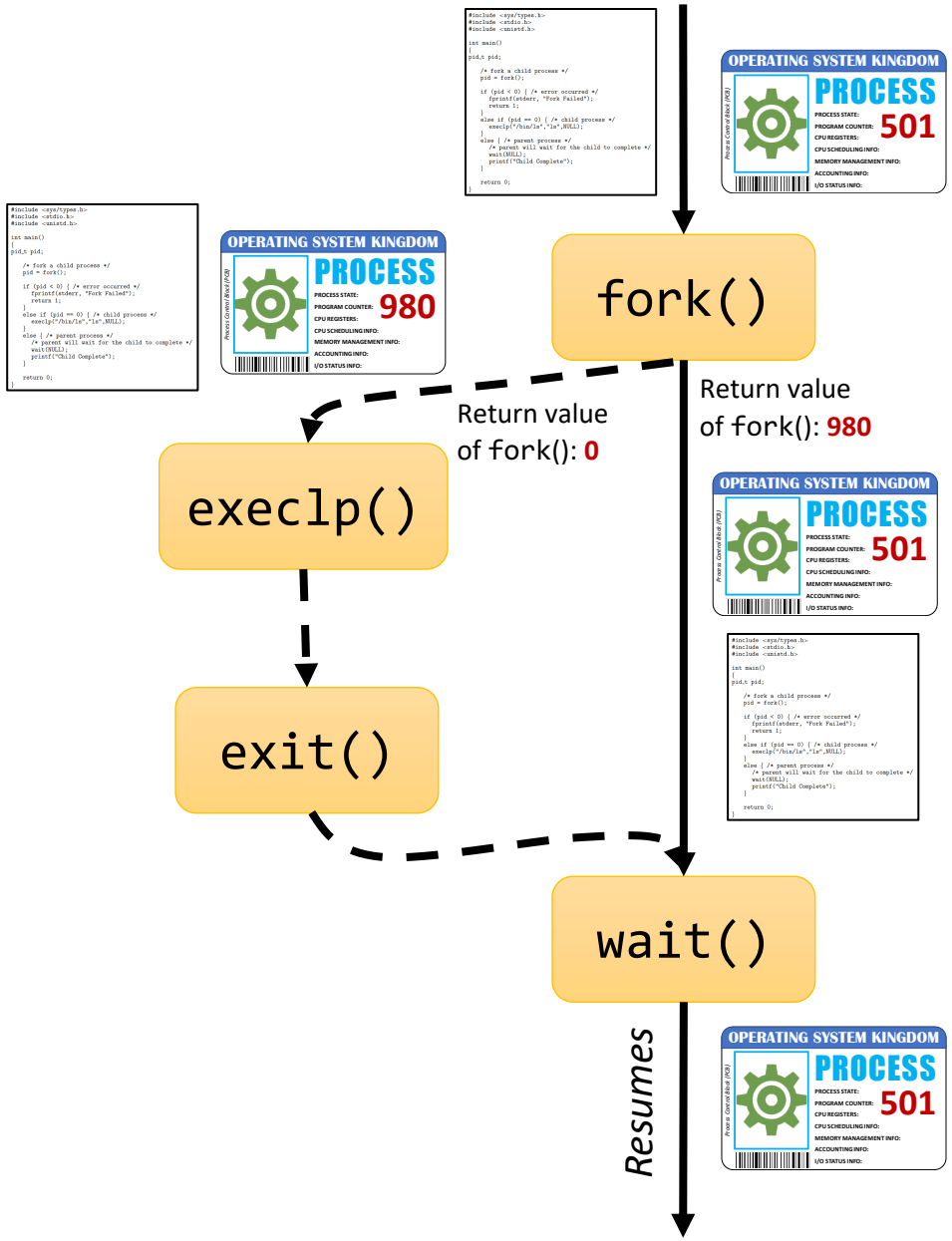
Process Creation

Parent *process creates* ***children*** *processes, which, in turn create other processes, forming a* ***tree of processes***

Process Creation



Process Creation



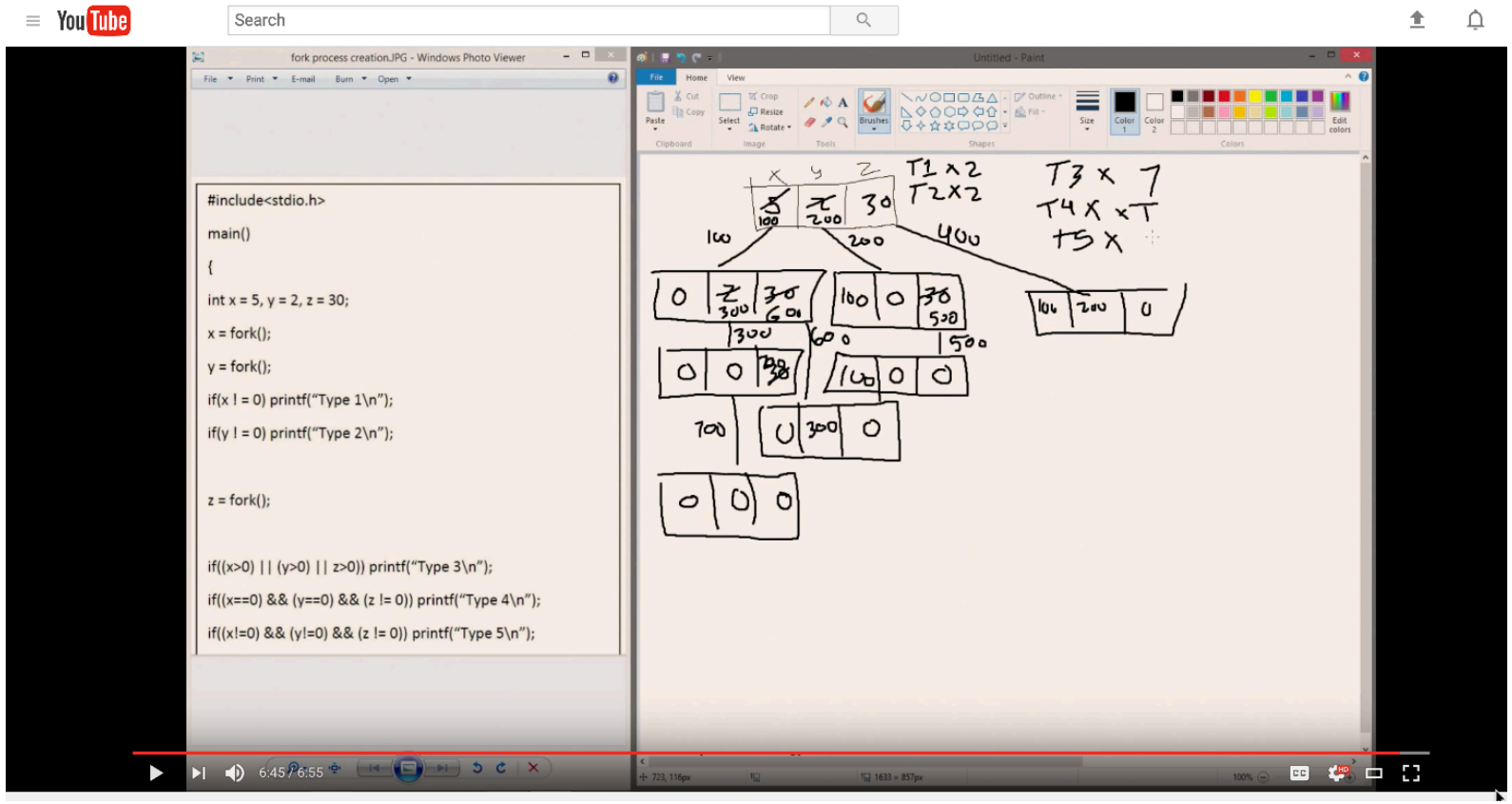

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```



https://www.youtube.com/watch?v=WcsZvdlLkPw

Viewing Processes

- The most used command to view processes (there are several) is `ps`.
- The `ps` program has a lot of options, but in its simplest form it is used like this:

```
[me@linuxbox ~]$ ps
  PID TTY          TIME CMD
 5198 pts/1    00:00:00 bash
10129 pts/1    00:00:00 ps
```

***TTY** is short for "teletype" and refers to the controlling terminal for the process.*

*The **TIME** field is the amount of CPU time consumed by the process.*

- The result in this example lists two processes, process 5198 and process 10129, which are `bash` and `ps`, respectively.
- As we can see, by default, `ps` doesn't show us very much, just the processes associated with the current terminal session.

Viewing Processes

- Adding the x option tells ps to show all our processes regardless of what terminal (if any) they are controlled by.

```
[me@linuxbox ~]$ ps x
  PID TTY      STAT   TIME COMMAND
 2799 ?        Ssl    0:00 /usr/libexec/bonobo-activation-server -ac
 2820 ?        Sl     0:01 /usr/libexec/evolution-data-server-1.10 --
15647 ?        Ss     0:00 /bin/sh /usr/bin/startkde
15751 ?        Ss     0:00 /usr/bin/ssh-agent /usr/bin/dbus-launch --
15754 ?        S      0:00 /usr/bin/dbus-launch --exit-with-session
15755 ?        Ss     0:01 /bin/dbus-daemon --fork --print-pid 4 -pr
15774 ?        Ss     0:02 /usr/bin/gpg-agent -s -daemon
15793 ?        S      0:00 start_kdeinit --new-startup +kcminit_start
15794 ?        Ss     0:00 kdeinit Running...
15797 ?        S      0:00 dcopserver -nosid
--snip--
```

The presence of a ? in the TTY column indicates no controlling terminal.

STAT reveals the current status of the process

State	Meaning
R	Running. This means the process is running or ready to run.
S	Sleeping. The process is not running; rather, it is waiting for an event, such as a keystroke or network packet.
D	Uninterruptible sleep. The process is waiting for I/O such as a disk drive.
T	Stopped. The process has been instructed to stop. You'll learn more about this later in the chapter.
Z	A defunct or "zombie" process. This is a child process that has terminated but has not been cleaned up by its parent.
<	A high-priority process. It's possible to grant more importance to a process, giving it more time on the CPU. This property of a process is called *niceness*. A process with high priority is said to be less nice because it's taking more of the CPU's time, which leaves less for everybody else.
N	A low-priority process. A process with low priority (a nice process) will get processor time only after other processes with higher priority have been serviced.

Viewing Processes

```
[me@linuxbox ~]$ ps aux
USER     PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root       1  0.0  0.0   2136   644 ?        Ss   Mar05   0:31 init
root       2  0.0  0.0      0     0 ?        S<   Mar05   0:00 [kt]
root       3  0.0  0.0      0     0 ?        S<   Mar05   0:00 [mi]
root       4  0.0  0.0      0     0 ?        S<   Mar05   0:00 [ks]
root       5  0.0  0.0      0     0 ?        S<   Mar05   0:06 [wa]
root       6  0.0  0.0      0     0 ?        S<   Mar05   0:36 [ev]
root       7  0.0  0.0      0     0 ?        S<   Mar05   0:00 [kh]
--snip--
```

Header	Meaning
USER	User ID. This is the owner of the process.
%CPU	CPU usage in percent.
%MEM	Memory usage in percent.
VSZ	Virtual memory size.
RSS	Resident set size. This is the amount of physical memory (RAM) the process is using in kilobytes.
START	Time when the process started. For values over 24 hours, a date is used.

Viewing Processes Dynamically with `top`

- While the `ps` command can reveal a lot about what the machine is doing, it provides only a **snapshot of the machine's state at the moment the `ps` command is executed**.
- To see a more dynamic view of the machine's activity, we use the `top` command.

```
[me@linuxbox ~]$ top
```

Viewing Processes Dynamically with `top`

Row	Field	Meaning
1	`top`	This is the name of the program.
	`14:59:20`	This is the current time of day.
	`up 6:30`	This is called *uptime*. It is the amount of time since the machine was last booted. In this example, the system has been up for six and a half hours.
	`2 users`	There are two users logged in.
	`load average:`	*Load average* refers to the number of processes that are waiting to run; that is, the number of processes that are in a runnable state and are sharing the CPU. Three values are shown, each for a different period of time. The first is the average for the last 60 seconds, the next the previous 5 minutes, and finally the previous 15 minutes. Values less than 1.0 indicate that the machine is not busy.
2	`Tasks:`	This summarizes the number of processes and their various process states.
3	`Cpu(s):`	This row describes the character of the activities that the CPU is performing.
	`0.7%us`	0.7 percent of the CPU is being used for *user processes*. This means processes outside the kernel.
	`1.0%sy`	1.0 percent of the CPU is being used for *system* (kernel) processes.
	`0.0%ni`	0.0 percent of the CPU is being used by "nice" (low-priority) processes.
	`98.3%id`	98.3 percent of the CPU is idle.
	`0.0%wa`	0.0 percent of the CPU is waiting for I/O.
4	`Mem:`	This shows how physical RAM is being used.
5	`Swap:`	This shows how swap space (virtual memory) is being used.

```
top - 14:59:20 up  6:30,  2 users,  load average: 0.07, 0.02, 0.00
Tasks: 109 total,   1 running, 106 sleeping,   0 stopped,   2 zombie
Cpu(s):  0.7%us,  1.0%sy,  0.0%ni, 98.3%id,  0.0%wa,  0.0%hi,  0.0%si
Mem:    319496k total,   314860k used,     4636k free,    19392k buff
Swap:   875500k total,   149128k used,   726372k free,   114676k cach

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 6244 me        39  19 31752 3124 2188 S  6.3  1.0  16:24.42 trackerd
11071 me        20   0  2304 1092  840 R  1.3  0.3   0:00.14 top
 6180 me        20   0  2700 1100  772 S  0.7  0.3   0:03.66 dbus-dae
 6321 me        20   0 20944 7248 6560 S  0.7  2.3   2:51.38 multiloa
 4955 root      20   0  104m 9668 5776 S  0.3  3.0   2:19.39 Xorg
    1 root      20   0  2976  528  476 S  0.0  0.2   0:03.14 init
    2 root      15  -5     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 migratio
    4 root      15  -5     0    0    0 S  0.0  0.0   0:00.72 ksoftirq
    5 root      RT  -5     0    0    0 S  0.0  0.0   0:00.04 watchdog
    6 root      15  -5     0    0    0 S  0.0  0.0   0:00.42 events/0
    7 root      15  -5     0    0    0 S  0.0  0.0   0:00.06 khelper
   41 root      15  -5     0    0    0 S  0.0  0.0   0:01.08 kblockd/
   67 root      15  -5     0    0    0 S  0.0  0.0   0:00.00 kseriod
  114 root      20   0     0    0    0 S  0.0  0.0   0:01.62 pdflush
  116 root      15  -5     0    0    0 S  0.0  0.0   0:02.44 kswapd0
```

Controlling Processes

- Now that we can see and monitor processes, let's gain some control over them.
- For our experiments, we're going to use a little program called `xlogo` as our guinea pig.
- The `xlogo` program is a sample program supplied with the X Window System (the underlying engine that makes the graphics on our display go), which simply displays a resizable window containing the X logo.

Controlling Processes

- First, we'll get to know our test subject.

```
[me@linuxbox ~]$ xlogo
```

- After entering the command, a small window containing the logo should appear somewhere on the screen. On some systems, `xlogo` might print a warning message, but it can be safely ignored.
- We can verify that `xlogo` is running by resizing its window. If the logo is redrawn in the new size, the program is running.

Controlling Processes

- Notice how our **shell prompt has not returned**?
  - This is because the shell is **waiting for the program to finish**, just like all the other programs we have used so far. If we close the `xlogo` window, the prompt returns.

Interrupting a Process

- Let's observe what happens when we run `xlogo` again.
  - First, enter the `xlogo` command and verify that the program is running.
  - Next, return to the terminal window and press CTRL+C.

```
[me@linuxbox ~]$ xlogo
[me@linuxbox ~]$
```

- In a terminal, pressing CTRL+C **interrupts** a program. This means we are politely asking the program to terminate.
- After we pressed CTRL+C, the `xlogo` window closed, and the shell prompt returned.

Putting a Process in the Background

- Let's say we wanted to get the shell prompt back without terminating the `xlogo` program.
- We can do this by placing the program in the **background**.
- Think of the terminal as having a **foreground** (with stuff visible on the surface like the shell prompt) and a **background** (with stuff hidden behind the surface).
- To launch a program so that it is **immediately placed in the background**, we follow the **command** with an & character.

```
[me@linuxbox ~]$ xlogo &
[1] 28236
[me@linuxbox ~]$
```

Putting a Process in the Background

```
[me@linuxbox ~]$ xlogo &
[1] 28236
[me@linuxbox ~]$
```

- After entering the command, the xlogo window appeared, and the shell prompt returned, but some funny numbers were printed too.
- This message is part of a shell feature called **job control**. With this message, the shell is telling us that we have started job number 1 ([1]) and that it has PID 28236.

```
[me@linuxbox ~]$ ps
  PID TTY          TIME CMD
10603 pts/1    00:00:00 bash
28236 pts/1    00:00:00 xlogo
28239 pts/1    00:00:00 ps
```

Putting a Process in the Background

- The shell's **job control facility** also gives us a way to list the jobs that have been launched from our terminal.
- Using the `jobs` command, we can see this list:

```
[me@linuxbox ~]$ jobs
[1]+  Running                 xlogo &
```

- The results show that we have **one job**, **numbered 1**; that it is **running**; and that the command was `xlogo &`.

Returning a Process to the Foreground

- A process in the **background** is *immune from terminal keyboard input*, including any attempt to interrupt it with CTRL+C. To return a process to the **foreground**, use the fg command:

```
[me@linuxbox ~]$ jobs
[1]+  Running                 xlogo &
[me@linuxbox ~]$ fg %1
xlogo
```

- The fg command followed by a percent sign and the job number (called a **jobspec**) does the trick. If we have only one background job, the jobspec is **optional**.
- Now to terminate xlogo, press CTRL+C.

Stopping (Pausing) a Process

- Sometimes we'll want to **stop** a process without terminating it. This is often done to allow a foreground process to be moved to the background.
- To stop a foreground process and place it in the background, press CTRL+Z.
- Let's try it. At the command prompt, type `xlogo`, press enter, and then press CTRL+Z.

```
[me@linuxbox ~]$ xlogo
[1]+  Stopped                 xlogo
[me@linuxbox ~]$
```

Stopping (Pausing) a Process

- After stopping `xlogo`, we can verify that the program has stopped by attempting to resize the xlogo window. We will see that it appears quite dead.
- We can either continue the program's execution in the foreground, using the `fg` command, or resume the program's execution in the background with the bg command.

```
[me@linuxbox ~]$ bg %1
[1]+ xlogo &
[me@linuxbox ~]$
```

Signals

- The `kill` command is used to "kill" processes. This allows us to **terminate programs that need killing** (that is, some kind of pausing or termination).

```
[me@linuxbox ~]$ xlogo &
[1] 28401
[me@linuxbox ~]$ kill 28401
[1]+  Terminated              xlogo
```

- We first launch `xlogo` in the background. The shell prints the jobspec and the PID of the background process.
- Next, we use the `kill` command and specify the PID of the process we want to terminate. We could have also specified the process using a jobspec (for example, %1) instead of a PID.

Signals

- The `kill` command doesn't exactly "kill" processes; rather, it **sends them signals**. Signals are one of several ways that the operating system communicates with programs.
- We have already seen signals in action with the use of CTRL+C and CTRL+Z. When the terminal receives one of these keystrokes, it sends a signal to the program in the **foreground**.
- In the case of CTRL+C, a signal called **INT (interrupt)** is sent; with CTRL+Z, a signal called **TSTP (terminal stop)** is sent.
- Programs, in turn, "listen" for signals and may act upon them as they are received and act upon signals allows a program to do things such as save work in progress when it is sent a termination signal.

Sending Signals to Processes with `kill`

- The `kill` command is used to send signals to programs. Its most common syntax looks like this:

```
kill -signal PID...
```

- If no signal is specified on the command line, then the **TERM (terminate) signal is sent by default**.

Number	Name	Meaning
1	HUP	Hang up. This is a vestige of the good old days when terminals were attached to remote computers with phone lines and modems. The signal is used to indicate to programs that the controlling terminal has "hung up." The effect of this signal can be demonstrated by closing a terminal session. The foreground program running on the terminal will be sent the signal and will terminate. This signal is also used by many daemon programs to cause a reinitialization. This means that when a daemon is sent this signal, it will restart and reread its configuration file. The Apache web server is an example of a daemon that uses the HUP signal in this way.
2	INT	Interrupt. This performs the same function as CTRL-C sent from the terminal. It will usually terminate a program.
9	KILL	Kill. This signal is special. Whereas programs may choose to handle signals sent to them in different ways, including ignoring them all together, the KILL signal is never actually sent to the target program. Rather, the kernel immediately terminates the process. When a process is terminated in this manner, it is given no opportunity to "clean up" after itself or save its work. For this reason, the KILL signal should be used only as a last resort when other termination signals fail.
15	TERM	Terminate. This is the default signal sent by the kill command. If a program is still "alive" enough to receive signals, it will terminate.
18	CONT	Continue. This will restore a process after a STOP or TSTP signal. This signal is sent by the bg and fg commands.
19	STOP	Stop. This signal causes a process to pause without terminating. Like the KILL signal, it is not sent to the target process, and thus it cannot be ignored.
20	TSTP	Terminal stop. This is the signal sent by the terminal when CTRL-Z is pressed. Unlike the STOP signal, the TSTP signal is received by the program, but the program may choose to ignore it.

Sending Signals to Processes with `kill`

- Let's try the `kill` command.

```
[me@linuxbox ~]$ xlogo &
[1] 13546
[me@linuxbox ~]$ kill -1 13546
[1]+  Hangup                  xlogo
```

- In this example, we start the xlogo program in the background and then send it a HUP signal with kill. The xlogo program terminates, and the shell indicates that the background process has received a hang-up signal.
- We may need to press ENTER a couple of times before the message appears.

Sending Signals to Processes with `kill`

- Note that signals may be specified either by number or by name, including the name prefixed with the letters `SIG`.

```
[me@linuxbox ~]$ xlogo &
[1] 13601
[me@linuxbox ~]$ kill -INT 13601
[1]+  Interrupt               xlogo
[me@linuxbox ~]$ xlogo &
[1] 13608
[me@linuxbox ~]$ kill -SIGINT 13608
[1]+  Interrupt               xlogo
```

Processes, like files, have **owners**, and you must be the owner of a process (or the superuser) to send it signals with `kill`.

Sending Signals to Multiple Processes

- It's also possible to send signals to multiple processes matching a specified program or username by using the `killall` command.

```
[me@linuxbox ~]$ xlogo &
[1] 18801
[me@linuxbox ~]$ xlogo &
[2] 18802
[me@linuxbox ~]$ killall xlogo
[1]-  Terminated              xlogo
[2]+  Terminated              xlogo
```

More Process-Related Commands

Command	Description
`pstree`	Outputs a process list arranged in a tree-like pattern showing the parent-child relationships between processes.
`vmstat`	Outputs a snapshot of system resource usage including memory, swap, and disk I/O. To see a continuous display, follow the command with a time delay (in seconds) for updates. Here's an example: `vmstat 5`. Terminate the output with CTRL-C.
`xload`	A graphical program that draws a graph showing system load over time.
`tload`	Similar to the `xload` program but draws the graph in the terminal. Terminate the output with CTRL-C.

Important Notes for Lab Solution Submission

- **What do you need to submit for each lab?**
  - Reply to the lab message on Ritaj with the following two items:
    - **[OPTIONAL]** *Your lab solution on a separate document.*
    - **[MUST]** *Script log file*; the `script` log file will prove that you have literally solved the lab tasks yourself as it contains timestamps as well as your username and machine name.
  - If you do not send your `script` log file, then you will be rewarded with **zero**.

Important Notes for Lab Solution Submission

- **How to generate the script log file?**
  - While solving lab tasks, perform the following per each task part:
    - `script –a UniversityID_FirstnameLastName.log`
    - solve the lab task using proper command, scripts, etc.
    - `exit`
  - At the beginning of your log file, you need to execute the following commands – not doing so will result on **zero** grade.

```
uname  -a
cd; pwd; whoami; finger
tail  -2  /etc/passwd
echo FullName
echo UniveristyId
```