The Process

Overview

- Process Basics
- Process Status:ps
- System processes
- Mechanism of process creation
- Internal and external command
- Running jobs in background
- Job execution with low priority
- Killing processes with signals
- Job control

Processes

- An instance of a running program.
- Example: grep command, process named “grep” is created.
- If multiple processes, Kernel manages the processes(not the shell).
- Attributes of a process :
  - Process-id (PID): Process is uniquely identified
  - Parent PID(PPID): PID of the parent process.

The shell process

- Shell maintains a set of environment variables. The shell's pathname is stored in shell, but its's PID is stored in?

- In a special variable $$

```
$ echo $$
```

- PID of your login shell changes, when logged out.

Parent and children

- Example: cat emp.lst

- Shell becomes the parent and cat becomes the child process.
- Since every process has a parent, we can't have a “orphaned” process .
- Like file, process can have only one parent, and can have one or more children.

- Cat emp.lst | grep 'director'

Wait or Not wait?

- Two different attitudes that can be taken by the parent toward child.
- Wait for the child to die.
- It may not wait for child to die (init)
- Built-in commands of the shell like pwd,cd etc don't create processes.

Process Status: ps

- Command to display some process attributes. The command reads through the kernal's data structures and process tables to fetch the characteristics of processes.

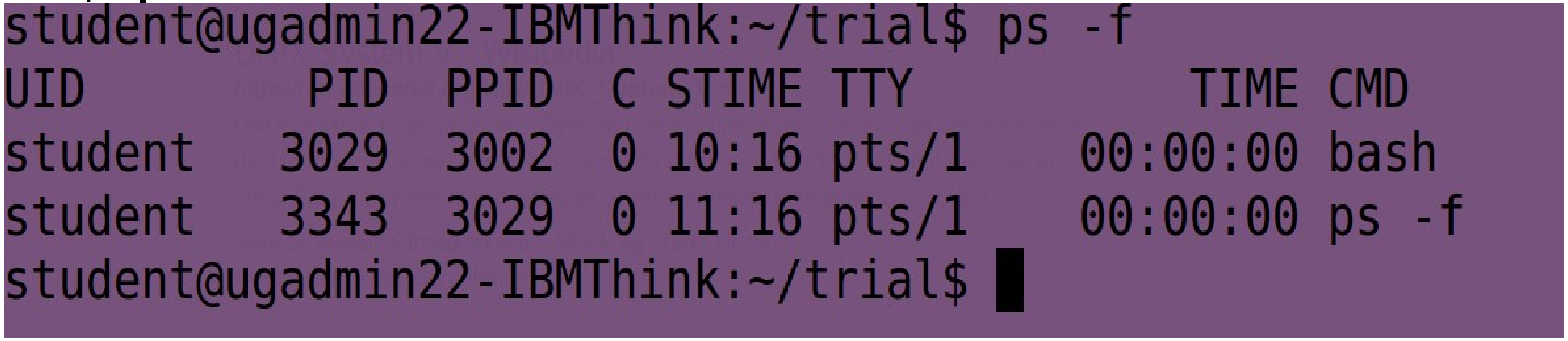
$$- The process number of the current shell. For shell scripts, this is the process ID under which they are executing.

ps options

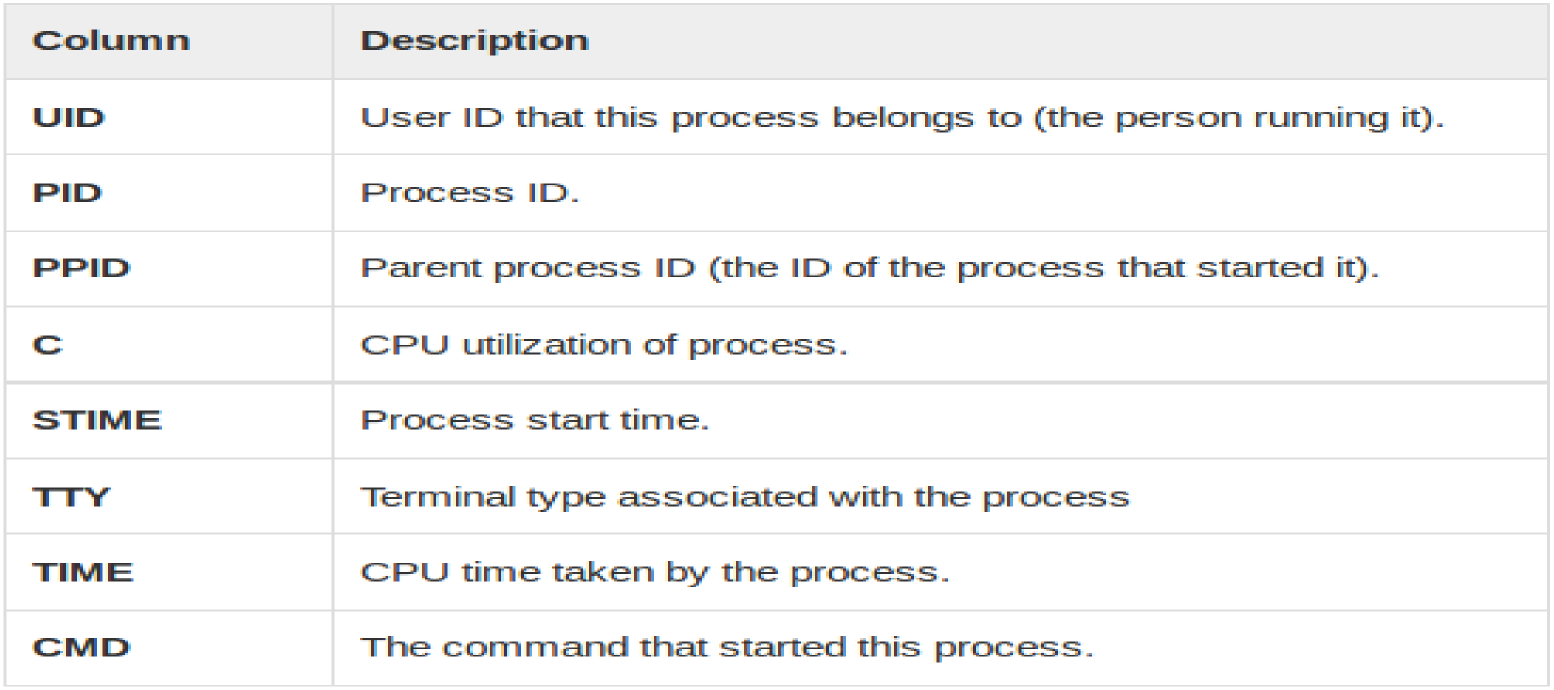
POSIX option	Significance
-f	Full listing showing the PPID of each process
-e or -A	All processes including user and system processes
-u usr	Processes of user usr only
-a	Processes of all users excluding processes not associated with terminal
-l	A long listing showing memory-related information
-t term	Processes running on terminal term (say, /dev /console)

ps options

$ ps -f



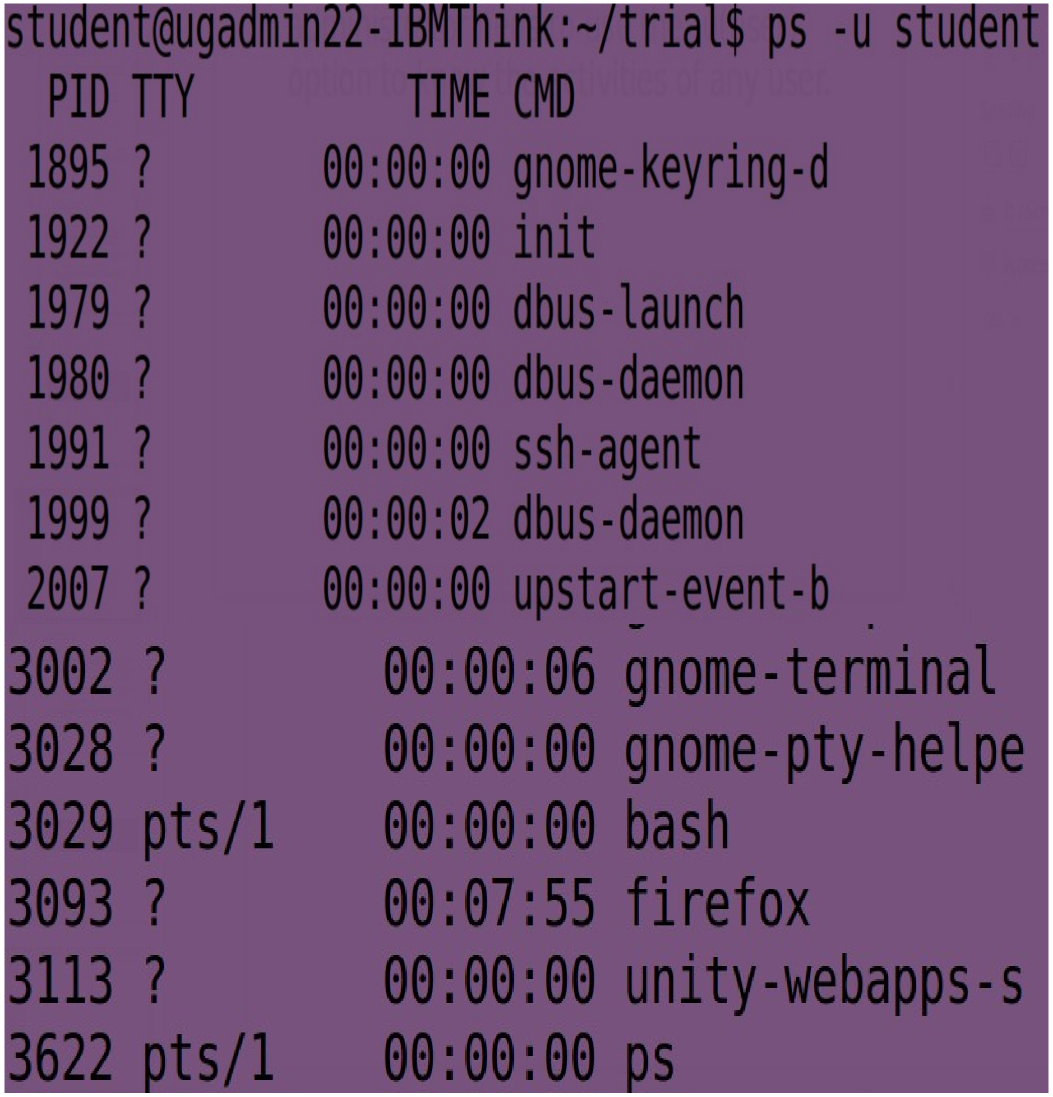
```
student@ugadmin22-IBMThink:~/trial$ ps -f
UID          PID   PPID  C STIME TTY          TIME CMD
student     3029   3002  0 10:16 pts/1    00:00:00 bash
student     3343   3029  0 11:16 pts/1    00:00:00 ps -f
student@ugadmin22-IBMThink:~/trial$
```



Column	Description
UID	User ID that this process belongs to (the person running it).
PID	Process ID.
PPID	Parent process ID (the ID of the process that started it).
C	CPU utilization of process.
STIME	Process start time.
TTY	Terminal type associated with the process
TIME	CPU time taken by the process.
CMD	The command that started this process.

ps options

$ ps -u usr : The system administrator needs to use the -u(user) option to know the activities of any user.

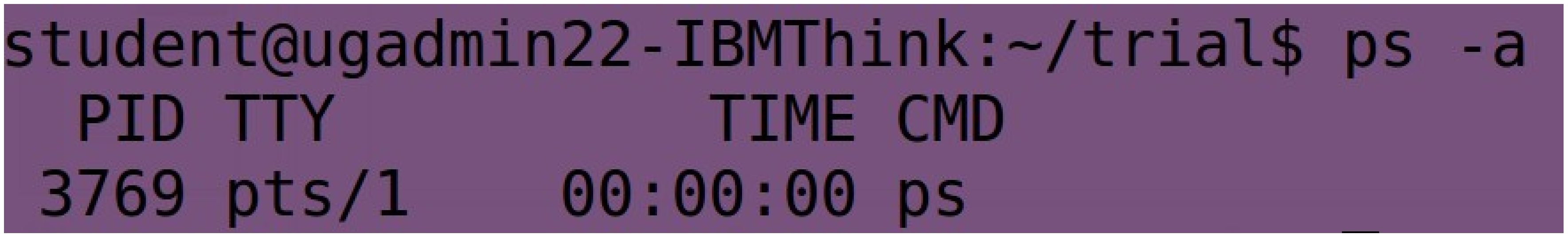


```
student@ugadmin22-IBMThink:~/trial$ ps -u student
  PID TTY          TIME CMD
 1895 ?        00:00:00 gnome-keyring-d
 1922 ?        00:00:00 init
 1979 ?        00:00:00 dbus-launch
 1980 ?        00:00:00 dbus-daemon
 1991 ?        00:00:00 ssh-agent
 1999 ?        00:00:02 dbus-daemon
 2007 ?        00:00:00 upstart-event-b
3002 ?        00:00:06 gnome-terminal
3028 ?        00:00:00 gnome-pty-helpe
3029 pts/1    00:00:00 bash
3093 ?        00:07:55 firefox
3113 ?        00:00:00 unity-webapps-s
3622 pts/1    00:00:00 ps
```

ps options

$ ps -a

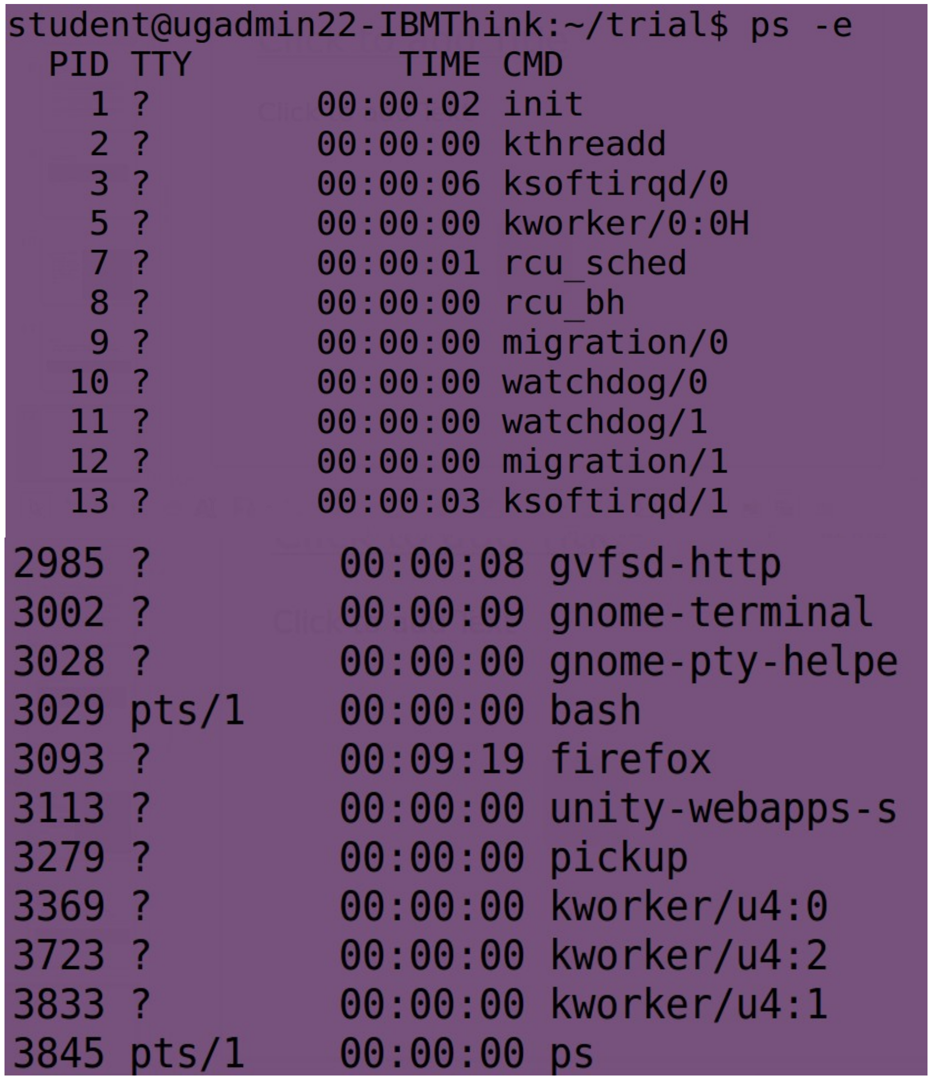
Lists the processes of all users but doesn't display the system processes.



```
student@ugadmin22-IBMThink:~/trial$ ps -a
  PID TTY          TIME CMD
 3769 pts/1    00:00:00 ps
```

ps options

$ ps -e



```
student@ugadmin22-IBMThink:~/trial$ ps -e
  PID TTY          TIME CMD
    1 ?        00:00:02 init
    2 ?        00:00:00 kthreadd
    3 ?        00:00:06 ksoftirqd/0
    5 ?        00:00:00 kworker/0:0H
    7 ?        00:00:01 rcu_sched
    8 ?        00:00:00 rcu_bh
    9 ?        00:00:00 migration/0
   10 ?        00:00:00 watchdog/0
   11 ?        00:00:00 watchdog/1
   12 ?        00:00:00 migration/1
   13 ?        00:00:03 ksoftirqd/1
2985 ?        00:00:08 gvfsd-http
3002 ?        00:00:09 gnome-terminal
3028 ?        00:00:00 gnome-pty-helpe
3029 pts/1    00:00:00 bash
3093 ?        00:09:19 firefox
3113 ?        00:00:00 unity-webapps-s
3279 ?        00:00:00 pickup
3369 ?        00:00:00 kworker/u4:0
3723 ?        00:00:00 kworker/u4:2
3833 ?        00:00:00 kworker/u4:1
3845 pts/1    00:00:00 ps
```

Mechanism of process creation

There are three distinct phases and uses three important system calls.

Fork: A new process is created by means of the **fork()** - system call

Exec: The parent then overwrites the image with the copy of the program that has to be executed.

Wait: The parent then execute the wait system call to wait for the child process to complete.

Creating a new process

- In UNIX, a new process is created by means of the **fork()** - system call. The OS performs the following functions:
    - It allocates a slot in the process table for the new process
    - It assigns a unique ID to the new process
    - It makes a copy of process image of the parent.
    - It returns the ID of the child to the parent process, and 0 to the child.

  - Note, the fork() call actually is called once but returns twice - namely in the parent and the child process.

Fork()

- Pid_t fork(void) is the prototype of the fork() call.
- Remember that fork() returns twice
  - in the newly created (child) process with return value 0
  - in the calling process (parent) with return value = pid of the new process.
    - A negative return value (-1) indicates that the call has failed
- Different return values are the key for distinguishing parent process from child process!
- The child process is an exact copy of the parent, yet, it is a copy i.e. an identical but separate process image.

A fork() Example

```
#include <unistd.h>
#include <sys/types.h>
main()
{
    pid_t pid     /* process id */
    printf("just one process before the fork()\n");
    pid = fork();
    if(pid == 0)
         printf("I am the child process\n");
    else if(pid > 0)
         printf("I am the parent process\n");
    else
         printf(" The fork() has failed\n")
}
```

How the Shell is created?

$$\begin{tikzcd}
\text{init} \arrow[r, "\text{fork}"'] & \text{getty} \arrow[r, "\text{fork-exec}"'] & \text{login} \arrow[r, "\text{fork-exec}"'] & \text{shell}
\end{tikzcd}$$

Internal and External Commands

External commands: Commonly used commands like **cat**, **ls** etc. The shell creates a process for each of these commands while remaining their parent.

Shell scripts: The shell executes these scripts by spawning another shell, which then executes the commands listed in the script. The child shell becomes the parent of the commands that feature in the shell.

Internal commands: When an internal command is entered, it is directly executed by the shell. Similarly, variable assignment like x=5, doesn’t generate a process either.

Zombie and Orphan processes

A process in zombie state is not alive; it does not use any resources nor does any work. But it is not allowed to die until the exit is acknowledged by the parent process.

It is possible for the parent itself to die before the child dies. In such case, the child becomes an ***orphan*** *and the kernel makes* ***init*** *the parent of the orphan. When this adopted child dies,* ***init*** *waits for its death.*

Running Jobs in Background

It's a program that can run without prompts or other manual interaction and can run in parallel with other active processes.

&: No Logging out

You can run a command that takes a long time to finish as a background job, so that you can be doing something else. To do this, use the & symbol at the end of command.

Running Jobs in Background

Example

```
$ sort -o emp.lst emp.lst &
[1] 1413                                   The job’s PID
```

i) First number [1] is the *job ID* of this command. The other number 1413 is the PID.

ii) When you specify a command line in a pipeline to run in the background, all the commands are run in the background, not just the last command.

Iii) The shell remains the parent of the background process.

Running Jobs in Background

nohup: Log out Safely

The UNIX system provides nohup statement which when prefixed to a command, permits execution of the process even after the user has logged out. You must use the & with it as well.

The syntax:

```
nohup command-string [input-file] output-file &
```

Running Jobs in Background

Example

```
$ nohup sort sales.dat &
1252
Sending output to nohup.out
```

The kernel handles the situations by reassigning the PPID of the orphan to the system’s init process (PID 1) - the parent of all shells.

nice: Job Execution with Low Priority

The ***nice*** command is used to control background process dispatch priority. *nice* values are system dependent and typically range from 1 to 19.

A high *nice* value implies a lower priority.

Example:

$ nice wc –l hugefile.txt

OR $ nice wc –l hugefile.txt &

The default nice value is set to 10.

nice: Job Execution with Low Priority

We can specify the nice value explicitly with –n *number* option where *number* is an offset to the default.

Example:

```
$ nice –n 5 wc –l hugefile.txt &
```

Killing Processes with Signals

When a process ends normally, the program returns its *exit status* to the parent. This exit status is a number returned by the program providing the results of the program's execution.

Sometimes, you want or need to terminate a process.

If the process to be stopped is a background process, use the *kill* command to get out of these situations.

Killing Processes with Signals

To use kill, use either of these forms:

kill PID(s)

To kill a process whose PID is 123 use,

```
$ kill 123
```

To kill several processes whose PIDs are 123, 342, and 73 use,

```
$ kill 123 342 73
```

The kill command sends a signal to a process. The default signal is SIGTERM signal (15).

Killing Processes with Signals

The system variable $! stores the PID of the last background job. You can kill the last background job without knowing its PID by specifying

$ kill $!

If the process ignores the signal SIGTERM, you can kill it with SIGKILL signal (9) as,

$ kill -9 123 OR **$ kill –s KILL 123**

Job Control

Job control facilities to,

- Relegate a job to the background (**bg**)
- Bring it back to the foreground (**fg**)
- List the active jobs (**jobs**)
- Suspend a foreground job (***[Ctrl-z]***)
- Kill a job (**kill**)

Job Control

Assume a process is taking a long time. You can suspend it by pressing *[Ctrl-z].*

```
[1] + Suspended       wc –l hugefile.txt
```

A suspended job is not terminated. You can now relegate it to background by,

```
$ bg
```

You can start more jobs in the background any time:

```
$ sort employee.dat > sortedlist.dat &
[2]    530
$ grep ‘director’ emp.dat &
[3]    540
```

Job Control

You can see a listing of these jobs using ***jobs*** command,

```
$ jobs
[3]  +  Running      grep ‘director’ emp.dat &
[2]  -  Running      sort employee.dat > sortedlist.dat &
[1]     Suspended    wc –l hugefile.txt
```

You can bring a job to foreground using

$ fg %2 OR $ fg %sort

at And batch: Execute Later

UNIX provides facilities to schedule a job to run at a specified time of day. If the system load varies greatly throughout the day, it makes sense to schedule less important jobs at a time when the system load is low.

at: One-Time Execution

To schedule one or more commands for a specified time, use the at command. With this command, you can specify a time, a date, or both.

For example,

```
$ at 14:23 Friday
at> lp /usr/sales/reports/*
at> echo “Files printed, Boss!” | mail -s”Job
  done” boss
[Ctrl-d]
commands will be executed using /usr/bin/bash
job 1041198880.a at Fri Oct 12 14:23:00 2007
```

batch: Execute in Batch Queue

The batch command lets the operating system decide an appropriate time to run a process. When you schedule a job with batch, UNIX starts and works on the process whenever the system load isn’t too great.

```
$ batch < empawk.sh
```