List of Slides

– LPI 101 –

Create, Monitor, and Kill Processes [7]

(Linux Professional Institute Certification)

a


```
   .~.
   /V\     by: Andrew Eager
  // \\        andy@linuxivr.com
  @._.@
```


$Id: gl1.103.5.slides.tex,v 1.2 2003/05/30 05:06:17 waratah Exp $

Create, Monitor, and Kill Processes

Objective

Candidate should be able to manage processes. This includes knowing how to run jobs in the foreground and background, bring a job from the background to the foreground and vice versa, start a process that will run without being connected to a terminal and signal a program to continue running after logout. Tasks also include monitoring active processes, selecting and sorting processes for display, sending signals to processes, killing processes and identifying and killing X applications that did not terminate after the X session closed.

Create, Monitor, and Kill Processes

Key files, terms, and utilities

```
&
bg
fg
jobs
kill
nohup
ps
top
```

Create, Monitor, and Kill Processes

Resources of interest

Processes

- A process is an executable loaded in memory.
- Linux is a multitasking operating system and so runs many processes concurrently.

Processes

- A process is an executable loaded in memory.
- Linux is a multitasking operating system and so runs many processes concurrently.
- INIT (PID 1) is the mother of all processes.

Processes

- A process is an executable loaded in memory.
- Linux is a multitasking operating system and so runs many processes concurrently.
- INIT (PID 1) is the mother of all processes.
- Programs, daemons, shells and commands are all processes.

Processes

- A process is an executable loaded in memory.
- Linux is a multitasking operating system and so runs many processes concurrently.
- INIT (PID 1) is the mother of all processes.
- Programs, daemons, shells and commands are all processes.
- The kernel automatically manages processes.

Processes

- A process is an executable loaded in memory.
- Linux is a multitasking operating system and so runs many processes concurrently.
- INIT (PID 1) is the mother of all processes.
- Programs, daemons, shells and commands are all processes.
- The kernel automatically manages processes.
- Normally processes live, execute and die without intervention from users.

Process Attributes and Concepts

The kernel starts the first process: `init` which has PID 1

Process Attributes and Concepts

The kernel starts the first process: `init` which has PID 1

Lifetime: Each process starts when it's command is executed, and lives till it dies or is killed.

Process Attributes and Concepts

The kernel starts the first process: `init` which has PID 1

Lifetime: Each process starts when it's command is executed, and lives till it dies or is killed.

Process ID (PID): Every process has a unique number assigned to it when it is started.

Process Attributes and Concepts

The kernel starts the first process: `init` which has PID 1

Lifetime: Each process starts when it's command is executed, and lives till it dies or is killed.

Process ID (PID): Every process has a unique number assigned to it when it is started.

User ID and Group ID: Processes have the privileges associated with the user / group who started them.

Process Attributes and Concepts

The kernel starts the first process: `init` which has PID 1

Lifetime: Each process starts when it's command is executed, and lives till it dies or is killed.

Process ID (PID): Every process has a unique number assigned to it when it is started.

User ID and Group ID: Processes have the privileges associated with the user / group who started them.

Parent processes (PPID): Shell processes are descendants of init and commands run from them are child processes.

Process Attributes and Concepts

The kernel starts the first process: `init` which has PID 1

Lifetime: Each process starts when it's command is executed, and lives till it dies or is killed.

Process ID (PID): Every process has a unique number assigned to it when it is started.

User ID and Group ID: Processes have the privileges associated with the user / group who started them.

Parent processes (PPID): Shell processes are descendants of init and commands run from them are child processes.

Environment: Each process inherits a set of *environmental variables* from it's parent process.

Process Attributes and Concepts

The kernel starts the first process: `init` which has PID 1

Lifetime: Each process starts when it's command is executed, and lives till it dies or is killed.

Process ID (PID): Every process has a unique number assigned to it when it is started.

User ID and Group ID: Processes have the privileges associated with the user / group who started them.

Parent processes (PPID): Shell processes are descendants of init and commands run from them are child processes.

Environment: Each process inherits a set of *environmental variables* from it's parent process.

Current Working Directory: Each process starts with a default directory.

Process Monitoring

Processes have to be monitored so as to check their health and use of system resources.

Process Monitoring

Processes have to be monitored so as to check their health and use of system resources.

- ps

```
$ ps aux |grep ssh
root  866  0.0  0.3  2676 1268 ?   S  07:56 0:00 /usr/sbin/sshd
```

Process Monitoring

Processes have to be monitored so as to check their health and use of system resources.

- ps

```
$ ps aux |grep ssh
root  866  0.0  0.3  2676 1268 ?   S  07:56 0:00 /usr/sbin/sshd
```

- pstree

```
$ pstree
init-+-alarmd
     |-apmd
     |-kdeinit-+-autorun
     |         |-kdeinit---emacs
```

Process Monitoring

Processes have to be monitored so as to check their health and use of system resources.

- ps

```
$ ps aux |grep ssh
root  866  0.0  0.3  2676 1268 ?   S  07:56 0:00 /usr/sbin/sshd
```

- pstree

```
$ pstree
init-+-alarmd
     |-apmd
     |-kdeinit-+-autorun
     |         |-kdeinit---emacs
```

- top

```
$ top
  PID USER     PRI NI  SIZE  RSS SHARE STAT %CPU %MEM  TIME COMMAND
 1792 geoffrey  11  0  8796 8796  7932 S     0.3  2.2  0:01 kdeinit
 1590 root      14  0 57512  13M  2572 R     0.1  3.6  0:41 X
 2857 geoffrey  14  0  1056 1056   836 R     0.1  0.2  0:01 top
```

Process Management

Normally the kernel automatically manages processes. However sometimes processes have to be started, stopped, restarted and killed.

Process Management

Normally the kernel automatically manages processes. However sometimes processes have to be started, stopped, restarted and killed.

- Starting a process:

```
#  /usr/sbin/httpd
ps aux |grep httpd
root      2987  0.0  0.4  4512 1584 ?   /usr/sbin/httpd
apache    3003  0.0  0.4  4656 1672 ?   /usr/sbin/httpd
```

Process Management

Normally the kernel automatically manages processes. However sometimes processes have to be started, stopped, restarted and killed.

- Starting a process:

```
#  /usr/sbin/httpd
ps aux |grep httpd
root      2987  0.0  0.4  4512 1584 ?   /usr/sbin/httpd
apache    3003  0.0  0.4  4656 1672 ?   /usr/sbin/httpd
```

- Occasionally processes die and have to be restarted.

Process Management

Normally the kernel automatically manages processes. However sometimes processes have to be started, stopped, restarted and killed.

- Starting a process:

```
#  /usr/sbin/httpd
ps aux |grep httpd
root      2987  0.0  0.4  4512 1584 ?   /usr/sbin/httpd
apache    3003  0.0  0.4  4656 1672 ?   /usr/sbin/httpd
```

- Occasionally processes die and have to be restarted.
- Processes may go berserk and have to be killed.

```
# kill -9 1234
```

Process Management

Normally the kernel automatically manages processes. However sometimes processes have to be started, stopped, restarted and killed.

- Starting a process:

```
#  /usr/sbin/httpd
ps aux |grep httpd
root      2987  0.0  0.4  4512 1584 ?   /usr/sbin/httpd
apache    3003  0.0  0.4  4656 1672 ?   /usr/sbin/httpd
```

- Occasionally processes die and have to be restarted.
- Processes may go berserk and have to be killed.

```
# kill -9 1234
```

- After configuration changes processes may have to be restarted so as to re-read their configuration files.

```
# service xinetd restart
Stopping xinetd:                [  OK  ]
Starting xinetd:                [  OK  ]
```

What is multitasking?

Multitasking is used to describe the situation where one processor (CPU) is used to perform multiple tasks concurrently.

What is multitasking?

Multitasking is used to describe the situation where one processor (CPU) is used to perform multiple tasks concurrently.

- Only one task or program is executing instructions on the CPU.

What is multitasking?

Multitasking is used to describe the situation where one processor (CPU) is used to perform multiple tasks concurrently.

- Only one task or program is executing instructions on the CPU.
- The CPU must be regularly switched between each program and others.

What is multitasking?

Multitasking is used to describe the situation where one processor (CPU) is used to perform multiple tasks concurrently.

- Only one task or program is executing instructions on the CPU.
- The CPU must be regularly switched between each program and others.
- This process is known as a *task switch*.

What is multitasking?

Multitasking is used to describe the situation where one processor (CPU) is used to perform multiple tasks concurrently.

- Only one task or program is executing instructions on the CPU.
- The CPU must be regularly switched between each program and others.
- This process is known as a *task switch*.
- At each *task switch* the Linux kernel must save the *context* of the CPU.

What is multitasking?

Multitasking is used to describe the situation where one processor (CPU) is used to perform multiple tasks concurrently.

- Only one task or program is executing instructions on the CPU.
- The CPU must be regularly switched between each program and others.
- This process is known as a *task switch*.
- At each *task switch* the Linux kernel must save the *context* of the CPU.
- The operating system uses the saved context when it switches back to the task the next time it gets some CPU time scheduled to it.

Task Scheduling

The total number of slices, when, how often and for how long the CPU is switched is determined by the multitasking algorithm and is handled by a software component within the kernel know as the *task scheduler*.

Task Scheduling

The total number of slices, when, how often and for how long the CPU is switched is determined by the multitasking algorithm and is handled by a software component within the kernel know as the *task scheduler*.

There are three basic types of task scheduling:

Task Scheduling

The total number of slices, when, how often and for how long the CPU is switched is determined by the multitasking algorithm and is handled by a software component within the kernel know as the *task scheduler*.

There are three basic types of task scheduling:

Nonpreemptive: A task must relinquish the CPU before a task switch occurs.

Task Scheduling

The total number of slices, when, how often and for how long the CPU is switched is determined by the multitasking algorithm and is handled by a software component within the kernel know as the *task scheduler*.

There are three basic types of task scheduling:

Nonpreemptive: A task must relinquish the CPU before a task switch occurs.

Preemptive: The kernel takes away the CPU from a task without notice.

Task Scheduling

The total number of slices, when, how often and for how long the CPU is switched is determined by the multitasking algorithm and is handled by a software component within the kernel know as the *task scheduler*.

There are three basic types of task scheduling:

Nonpreemptive: A task must relinquish the CPU before a task switch occurs.

Preemptive: The kernel takes away the CPU from a task without notice.

Realtime: Tasks are prioritised. High priority tasks must complete before a task switch.

What is a Process?

The term process is a fundamental abstraction.

What is a Process?

The term process is a fundamental abstraction.

- Two of the more traditional definitions of a process are:

  “A program in execution.”

  “A single program running in its own virtual address space”

What is a Process?

The term process is a fundamental abstraction.

- Two of the more traditional definitions of a process are:

  "A program in execution."

  "A single program running in its own virtual address space"

- In practice, a process is simply an executable that has been loaded into memory and is either running or ready to run on the system.

Process types

Processes under Linux fall into three basic categories:

Process types

Processes under Linux fall into three basic categories:

Interactive Process: An interactive process is a process initiated from (and controlled by) a shell. Interactive processes may be in foreground or background.
(Example: `ls, ls &`)

Process types

Processes under Linux fall into three basic categories:

Interactive Process: An interactive process is a process initiated from (and controlled by) a shell. Interactive processes may be in foreground or background.
(Example: `ls, ls &`)

Batch Process: A batch process is a process that is not associated with a terminal but is submitted to a queue to be executed sequentially.
(Example `slocate` started by `cron`)

Process types

Processes under Linux fall into three basic categories:

Interactive Process: An interactive process is a process initiated from (and controlled by) a shell. Interactive processes may be in foreground or background.
(Example: `ls, ls &`)

Batch Process: A batch process is a process that is not associated with a terminal but is submitted to a queue to be executed sequentially.
(Example `slocate` started by `cron`)

Daemon Process: A daemon process is a process that runs in the background until it's required. This kind of processes is usually initiated when Linux boots.
(Example: `inetd`, `lpd`)

Elements associated with a process

For each process running on the system, the kernel needs to keep a list of resources used by that process.

Some of these resources include:

Elements associated with a process

For each process running on the system, the kernel needs to keep a list of resources used by that process.

Some of these resources include:

- tty association (`tty_struct`)

Elements associated with a process

For each process running on the system, the kernel needs to keep a list of resources used by that process.

Some of these resources include:

- tty association (`tty_struct`)
- file system (eg current directory & open files) (`fs_struct`, `files_struct`)

Elements associated with a process

For each process running on the system, the kernel needs to keep a list of resources used by that process.

Some of these resources include:

- tty association (`tty_struct`)
- file system (eg current directory & open files) (`fs_struct`, `files_struct`)
- memory allocation (`mm_struct`)

Elements associated with a process

For each process running on the system, the kernel needs to keep a list of resources used by that process.

Some of these resources include:

- tty association (`tty_struct`)
- file system (eg current directory & open files) (`fs_struct`, `files_struct`)
- memory allocation (`mm_struct`)
- Signals received (`signal_struct`)

Process States

At any given point in time, a process is in one of 5 states:

Process States

At any given point in time, a process is in one of 5 states:

TASK_RUNNING: The process is either executing on the CPU or waiting to be executed.

Process States

At any given point in time, a process is in one of 5 states:

TASK_RUNNING: The process is either executing on the CPU or waiting to be executed.

TASK_INTERRUPTIBLE: The process is sleeping until something becomes true.
Raising a hardware interrupt, waiting for a system resource etc are examples of a condition that might wake the process up. If a signal is received by the process (eg KILL -HUP) the process will also be woken up.

Process States

At any given point in time, a process is in one of 5 states:

TASK_RUNNING: The process is either executing on the CPU or waiting to be executed.

TASK_INTERRUPTIBLE: The process is sleeping until something becomes true. Raising a hardware interrupt, waiting for a system resource etc are examples of a condition that might wake the process up. If a signal is received by the process (eg KILL -HUP) the process will also be woken up.

TASK_UNINTERRUPTIBLE: Like the previous state except that delivering a signal will not wake the process up.

Process States

At any given point in time, a process is in one of 5 states:

TASK_RUNNING: The process is either executing on the CPU or waiting to be executed.

TASK_INTERRUPTIBLE: The process is sleeping until something becomes true. Raising a hardware interrupt, waiting for a system resource etc are examples of a condition that might wake the process up. If a signal is received by the process (eg KILL -HUP) the process will also be woken up.

TASK_UNINTERRUPTIBLE: Like the previous state except that delivering a signal will not wake the process up.

TASK_STOPPED: Process execution has stopped. A process enters this state after receiving a SIGSTOP signal. A debugger may use this to step through a program.

Process States

At any given point in time, a process is in one of 5 states:

TASK_RUNNING: The process is either executing on the CPU or waiting to be executed.

TASK_INTERRUPTIBLE: The process is sleeping until something becomes true. Raising a hardware interrupt, waiting for a system resource etc are examples of a condition that might wake the process up. If a signal is received by the process (eg KILL -HUP) the process will also be woken up.

TASK_UNINTERRUPTIBLE: Like the previous state except that delivering a signal will not wake the process up.

TASK_STOPPED: Process execution has stopped. A process enters this state after receiving a SIGSTOP signal. A debugger may use this to step through a program.

TASK_ZOMBIE: Process execution has stopped but the kernel has not yet ?cleaned up? the resources allocated to the process.

The Process Family Tree

Every process (with the sole exception of the kernel), must be created by another process. The terms *parent*, *child* and *sibling* (or sometimes *father*, *son* and *brother* in a patriarchal sense) are used to describe the relationships between processes.

As an example consider the following line executed from the bash prompt:

```
[andy@Node4] andy]$ ls & df -h &
```

The following relationships are true:

The Process Family Tree

Every process (with the sole exception of the kernel), must be created by another process. The terms *parent*, *child* and *sibling* (or sometimes *father*, *son* and *brother* in a patriarchal sense) are used to describe the relationships between processes.

As an example consider the following line executed from the bash prompt:

```
[andy@Node4] andy]$ ls & df -h &
```

The following relationships are true:

- The `ls` and `df` processes are both siblings to each other.

The Process Family Tree

Every process (with the sole exception of the kernel), must be created by another process. The terms *parent*, *child* and *sibling* (or sometimes *father*, *son* and *brother* in a patriarchal sense) are used to describe the relationships between processes.

As an example consider the following line executed from the bash prompt:

```
[andy@Node4] andy]$ ls & df -h &
```

The following relationships are true:

- The `ls` and `df` processes are both siblings to each other.
- The `bash` process (ie the shell) is the parent to both `ls` and `df`.

The Process Family Tree

Every process (with the sole exception of the kernel), must be created by another process. The terms *parent*, *child* and *sibling* (or sometimes *father*, *son* and *brother* in a patriarchal sense) are used to describe the relationships between processes.

As an example consider the following line executed from the bash prompt:

```
[andy@Node4] andy]$ ls & df -h &
```

The following relationships are true:

- The `ls` and `df` processes are both siblings to each other.
- The `bash` process (ie the shell) is the parent to both `ls` and `df`.
- The `ls` process has `bash` as its parent.

The Process Family Tree

Every process (with the sole exception of the kernel), must be created by another process. The terms *parent*, *child* and *sibling* (or sometimes *father*, *son* and *brother* in a patriarchal sense) are used to describe the relationships between processes.

As an example consider the following line executed from the bash prompt:

```
[andy@Node4] andy]$ ls & df -h &
```

The following relationships are true:

- The `ls` and `df` processes are both siblings to each other.
- The `bash` process (ie the shell) is the parent to both `ls` and `df`.
- The `ls` process has `bash` as its parent.
- The `df` process has `bash` as its parent.

The Kernel is at the Top of the Family Tree

```
Kernel -->
          Init -->
                  all other processes -->
                                          even more processes -->
```

The Kernel is at the Top of the Family Tree

- When Linux boots, the first thing it does is load the kernel into memory and start executing itself.

```
Kernel -->
          Init -->
                  all other processes -->
                                          even more processes -->
```

The Kernel is at the Top of the Family Tree

- When Linux boots, the first thing it does is load the kernel into memory and start executing itself.
- One of the first things it does once execution starts, is to spawn a process called init, which in turn spawns other processes.

```
Kernel -->
          Init -->
                  all other processes -->
                                          even more processes -->
```

The Kernel is at the Top of the Family Tree

- When Linux boots, the first thing it does is load the kernel into memory and start executing itself.
- One of the first things it does once execution starts, is to spawn a process called init, which in turn spawns other processes.
- In this sense, the kernel is at the top of the family tree, with only one child process called init.

```
Kernel -->
          Init -->
                  all other processes -->
                                          even more processes -->
```

The Kernel is at the Top of the Family Tree

- When Linux boots, the first thing it does is load the kernel into memory and start executing itself.
- One of the first things it does once execution starts, is to spawn a process called init, which in turn spawns other processes.
- In this sense, the kernel is at the top of the family tree, with only one child process called init.
- Init in turn has many children and probably many grandchildren.

```
Kernel -->
          Init -->
                  all other processes -->
                                          even more processes -->
```

Process IDs

In order for the kernel to keep track of all processes and their descendants, a process ID is assigned to every process running on the system. Process IDs are just numbers and run from 0 to 32767. The number 32767 is the largest signed integer available with a sixteen bit word size and is used to maintain backward compatibility with 16 bit architectures.

There are two PIDs (process IDs) that are always the same:

- kernel PID is always 0
- init PID is always 1

Process IDs

Each time a new process is created, a new PID is allocated and is equal to the last PID issued plus one. Once the last PID is reached, the PID wraps back around to zero and the next available PID is used (note that 0 and 1 will never be available). This scheme is a little like the assignment of telephone numbers: When a telephone service is disconnected, rather than just assigning the old telephone number to a new subscriber, the old number remains out of use until all other numbers have been used up. This saves “wrong numbers” to the new subscriber from callers who have not yet realised that the old number is no longer connected to the person they were trying to reach. In a similar vein, the kernel does this to minimise “wrong numbers” from other processes who have not yet worked out that their intended process no longer exists. This is especially true for Interprocess Communication (IPC) which uses the PID to identify a target process.

Displaying Process Information

There are three utilities used to display the state of running processes:

- `ps`
- `pstree`
- `top`

Displaying Process Information

There are three utilities used to display the state of running processes:

- `ps`
- `pstree`
- `top`
- The ps command is used to display a “snapshot” of all processes running on the system at the time the `ps` command was executed.

Displaying Process Information

There are three utilities used to display the state of running processes:

- `ps`
- `pstree`
- `top`
- The ps command is used to display a “snapshot” of all processes running on the system at the time the `ps` command was executed.
- `pstree` gives a tree view of the processes.

Displaying Process Information

There are three utilities used to display the state of running processes:

- `ps`
- `pstree`
- `top`
- The ps command is used to display a “snapshot” of all processes running on the system at the time the `ps` command was executed.
- `pstree` gives a tree view of the processes.
- The `top` command is used to display a real-time display of all processes running on the system. Top can also be used in interactive mode to `kill` or `renice` (change priority) of a process.

Process Monitoring—ps

usage: `ps [options]`

The `ps` command has a huge number of switches. The switches can be subdivided into two main groups:

- Process selection (which processes to display)
- Output control (how and what output should be displayed)

ps options

```
$ ps ?
ERROR: Garbage option.
********* simple selection *********  ********* selection by list *********
-A all processes                      -C by command name
-N negate selection                   -G by real group ID (supports names)
-a all w/ tty except session leaders  -U by real user ID (supports names)
-d all except session leaders         -g by session leader OR by group name
-e all processes                      -p by process ID
T  all processes on this terminal     -s processes in the sessions given
a  all w/ tty, including other users  -t by tty
g  all, even group leaders!           -u by effective user ID (supports names)
r  only running processes             U  processes for specified users
x  processes w/o controlling ttys     t  by tty
*********** output format **********  *********** long options ***********
-o,o user-defined  -f full            --Group --User --pid --cols
-j,j job control   s  signal          --group --user --sid --rows
-O,O preloaded -o  v  virtual memory  --cumulative --format --deselect
-l,l long          u  user-oriented   --sort --tty --forest --version
                   X  registers       --heading --no-heading
                    ********* misc options *********
-V,V show version       L  list format codes  f  ASCII art forest
-m,m show threads       S  children in sum    -y change -l format
-n,N set namelist file  c  true command name  n  numeric WCHAN,UID
-w,w wide output        e  show environment   -H process heirarchy
```

ps options

The switches that need to be known for the purposes of LPIC are as follows:

a Display processes for all users

txx Display processes within controlling terminal `txx`

u Display user information for the process

l Display in long format with detailed information

s Display signal information

m Display memory information

x Display processes without a controlling terminal

S Display CPU time and page faults of child processes

-C cmd Search for instances of command `cmd.`

-f Forest mode shows process family trees.

-w Wide format

ps field names & their meanings

USER The user who started the process

PID The process ID

%CPU Shows the cputime / realtime percentage.

%MEM The fraction of RSS divided by the total size of RAM

VSZ Size of virtual memory used by the process

RSS Resident set size (Data & Text segments only) in Kb

TTY The TTY associated with this process

STAT The current status (DRSTZW< NL) (details next slide)

TIME CPU time in MINS:SECS

COMMAND The full command line used to start the process

ps Status Field

```
$ ps aux
USER     PID %CPU %MEM   VSZ  RSS TTY      STAT START   TIME COMMAND
root       1  0.0  0.0  1304   72 ?        S    Mar21   0:19 init
```

D uninterruptible sleep (usually IO)

R runnable (on run queue)

S sleeping

T traced or stopped

Z a defunct (“zombie”) process

W has no resident pages

< high-priority process

N low-priority task

L has pages locked into memory (for real-time and custom IO)

ps Status Field

```
$ ps aux
USER       PID %CPU %MEM    VSZ  RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.2   1384  516 ?        S    11:43   0:04 init [5]
root         2  0.0  0.0      0    0 ?        SW   11:43   0:00 [keventd]
root         3  0.0  0.0      0    0 ?        SW   11:43   0:00 [kapm-idled]
root         5  0.0  0.0      0    0 ?        SW   11:43   0:00 [kswapd]
root         6  0.0  0.0      0    0 ?        SW   11:43   0:00 [kreclaimd]
root         7  0.0  0.0      0    0 ?        SW   11:43   0:00 [bdflush]
root         8  0.0  0.0      0    0 ?        SW   11:43   0:00 [kupdated]
root         9  0.0  0.0      0    0 ?        SW<  11:43   0:00 [mdrecoveryd]
root       103  0.0  0.0      0    0 ?        SW   11:44   0:00 [kjournald]
root       474  0.0  0.2   1444  620 ?        S    11:44   0:00 syslogd -m 0
root       479  0.0  0.4   2080 1152 ?        S    11:44   0:00 klogd -2
rpc        497  0.0  0.2   1632  708 ?        S    11:44   0:00 portmap
rpcuser    525  0.0  0.3   1624  796 ?        S    11:44   0:00 rpc.statd
ntp        735  0.0  0.8   2088 2080 ?        SL   11:44   0:00 ntpd -U ntp
root       759  0.0  0.3   5784  856 ?        S    11:44   0:00 ypbind
root       763  0.0  0.3   5784  856 ?        S    11:44   0:00 ypbind
.....
andy      1176  0.0  0.5   2620 1508 pts/0    S    11:46   0:00 bash
root      1343  0.0  0.7   3000 1816 tty1     S    15:21   0:00 ssh node10
andy      1664  0.0  0.3   2824  924 pts/1    R    21:52   0:00 ps -aux
```

Process Monitoring—`pstree`

```
$ pstree
init-+-anacron---run-parts---cfengine
     |-5*[apache-ssl]
     |-atd
     |-bash---startx---xinit-+-X
     |                       `-enlightenment-+-E-Clock.epplet
     |                                       |-E-Cpu.epplet
     |                                       |-Emix.epplet
     |                                       |-Eterm---bash-+-abiword---AbiWord
     |                                       |              `-mozilla-bin---moz
     |                                       |-Eterm---bash---bash
     |                                       |-Eterm---bash
     |                                       |-Eterm---bash---gv---gs
     |                                       |-Eterm---bash---mutt
     |                                       |-Eterm---bash---emacs-+-ispell
     |                                       |                      `-xdvi---gs
     |                                       |-Eterm---bash---pstree
     |                                       `-Eterm---bash---man---pager
     |-cron
     |-gcache
     |-6*[getty]
     |-inetd---nmbd
     |-junkbuster
```

`pstree` options

Three commonly used options for `pstree`:

`pstree` options

Three commonly used options for `pstree`:

`-a` Show command line arguments.

```
    |-xfs -daemon
    |-xfstt --port 7101 --daemon --user nobody
    ‘-zope-z2 /usr/sbin/zope-z2
       ‘-python /usr/sbin/zope-z2
```

`pstree` options

Three commonly used options for `pstree`:

`-a` Show command line arguments.

```
    |-xfs -daemon
    |-xfstt --port 7101 --daemon --user nobody
    ‘-zope-z2 /usr/sbin/zope-z2
       ‘-python /usr/sbin/zope-z2
```

`-n` Sort processes with the same ancestor by PID

`pstree` options

Three commonly used options for `pstree`:

`-a` Show command line arguments.

```
|-xfs -daemon
|-xfstt --port 7101 --daemon --user nobody
‘-zope-z2 /usr/sbin/zope-z2
   ‘-python /usr/sbin/zope-z2
```

`-n` Sort processes with the same ancestor by PID

`-p` Show PIDs.

```
init(1)-+-anacron(27095)---run-parts(27755)---cfengine(27765)
        |-apache-ssl(27188)
        |-apache-ssl(27189)
```

Process Monitoring—`top`

The ”top” command provides a continuously updated, real-time look at process activity, memory and swap file usage plus CPU activity.

It also shows what processes are running and by whom.

Process Monitoring—`top`

The ”top” command provides a continuously updated, real-time look at process activity, memory and swap file usage plus CPU activity.

It also shows what processes are running and by whom.

- Its primary use is as an administration and system information tool. It provides an extension to the functionality of the ”ps” command.

Process Monitoring—`top`

The ”top” command provides a continuously updated, real-time look at process activity, memory and swap file usage plus CPU activity.

It also shows what processes are running and by whom.

- Its primary use is as an administration and system information tool. It provides an extension to the functionality of the ”ps” command.
- It makes it easy to find an errand process and ”kill” that process. It also has an interactive interface whereby options can be passed while the command is actually running. All in all, a very useful tool.

top

```
 9:16am  up 13 days,  8:05,  8 users,  load average: 0.05, 0.05, 0.00
86 processes: 84 sleeping, 1 running, 1 zombie, 0 stopped
CPU states:  2.3% user,  0.7% system,  0.0% nice, 96.8% idle
Mem:  900236K av, 546472K used, 353764K free,  0K shrd,  37552K buff
Swap: 329324K av,  34784K used, 294540K free          190764K cached

  PID USER  PRI  NI  SIZE  RSS SHARE STAT LIB %CPU %MEM  TIME COMMAND
10281 root   16 -10 97952 6452  1584 S <    0  3.9  0.7 56:57 X
12547 geoff  16   0  1728 1728   764 R      0  0.9  0.1  0:01 top
10284 geoff  12   0  3012 2568  1352 S      0  0.7  0.2 50:49 enlight
12173 geoff  10   0  9340 9340  3768 S      0  0.3  1.0  0:11 emacs
12543 geoff   9   0  3328 3328  2072 S      0  0.1  0.3  0:00 Eterm
    1 root    9   0   116   72    52 S      0  0.0  0.0  0:19 init
    2 root    9   0     0    0     0 SW     0  0.0  0.0  0:01 keventd
```

top's basic command line options

Note: dashes not required.

`-b` Batch mode. Useful for sending output from top to other programs or to a file. Output is plain text.

`-d` Delay between screen updates. (default 5 seconds)

`-i` Start top ignoring any idle or zombie processes.

`-p` Monitor only processes with given process id. (x20)

`-q` This causes top to refresh without any delay.

top's upper screen

```
 9:16am  up 13 days,  8:05,  8 users,  load average: 0.05, 0.05, 0.00
86 processes: 84 sleeping, 1 running, 1 zombie, 0 stopped
CPU states:  2.3% user,  0.7% system,  0.0% nice, 96.8% idle
Mem:  900236K av, 546472K used, 353764K free,  0K shrd,  37552K buff
Swap: 329324K av,  34784K used, 294540K free          190764K cached
```

top's upper screen

```
 9:16am  up 13 days,  8:05,  8 users,  load average: 0.05, 0.05, 0.00
86 processes: 84 sleeping, 1 running, 1 zombie, 0 stopped
CPU states:  2.3% user,  0.7% system,  0.0% nice, 96.8% idle
Mem:  900236K av, 546472K used, 353764K free,  0K shrd,  37552K buff
Swap: 329324K av,  34784K used, 294540K free          190764K cached
```

- The current system time:

top's upper screen

```
 9:16am  up 13 days,  8:05,  8 users,  load average: 0.05, 0.05, 0.00
86 processes: 84 sleeping, 1 running, 1 zombie, 0 stopped
CPU states:  2.3% user,  0.7% system,  0.0% nice, 96.8% idle
Mem:  900236K av, 546472K used, 353764K free,  0K shrd,  37552K buff
Swap: 329324K av,  34784K used, 294540K free          190764K cached
```

- The current system time:
- The "up time" of the system:

top's upper screen

```
 9:16am  up 13 days,  8:05,  8 users,  load average: 0.05, 0.05, 0.00
86 processes: 84 sleeping, 1 running, 1 zombie, 0 stopped
CPU states:  2.3% user,  0.7% system,  0.0% nice, 96.8% idle
Mem:  900236K av, 546472K used, 353764K free,  0K shrd,  37552K buff
Swap: 329324K av,  34784K used, 294540K free          190764K cached
```

- The current system time:
- The "up time" of the system:
- How many users are logged in.

top's upper screen

```
 9:16am  up 13 days,  8:05,  8 users,  load average: 0.05, 0.05, 0.00
86 processes: 84 sleeping, 1 running, 1 zombie, 0 stopped
CPU states:  2.3% user,  0.7% system,  0.0% nice, 96.8% idle
Mem:  900236K av, 546472K used, 353764K free,  0K shrd,  37552K buff
Swap: 329324K av,  34784K used, 294540K free          190764K cached
```

- The current system time:
- The "up time" of the system:
- How many users are logged in.
- The "load average" : the average number of processes ready to run over the last 1,5 and 15 minutes

top's upper screen

```
 9:16am  up 13 days,  8:05,  8 users,  load average: 0.05, 0.05, 0.00
86 processes: 84 sleeping, 1 running, 1 zombie, 0 stopped
CPU states:  2.3% user,  0.7% system,  0.0% nice, 96.8% idle
Mem:  900236K av, 546472K used, 353764K free,  0K shrd,  37552K buff
Swap: 329324K av,  34784K used, 294540K free          190764K cached
```

- The current system time:
- The "up time" of the system:
- How many users are logged in.
- The "load average" : the average number of processes ready to run over the last 1,5 and 15 minutes
- "CPU States" shows the percentage of CPU time spent in usermode, system mode and at idle.

top's upper screen

```
 9:16am  up 13 days,  8:05,  8 users,  load average: 0.05, 0.05, 0.00
86 processes: 84 sleeping, 1 running, 1 zombie, 0 stopped
CPU states:  2.3% user,  0.7% system,  0.0% nice, 96.8% idle
Mem:  900236K av, 546472K used, 353764K free,  0K shrd,  37552K buff
Swap: 329324K av,  34784K used, 294540K free          190764K cached
```

- The current system time:
- The "up time" of the system:
- How many users are logged in.
- The "load average" : the average number of processes ready to run over the last 1,5 and 15 minutes
- "CPU States" shows the percentage of CPU time spent in usermode, system mode and at idle.
- "MEM" shows a complete set of statistics on current memory usage.

top's upper screen

```
 9:16am  up 13 days,  8:05,  8 users,  load average: 0.05, 0.05, 0.00
86 processes: 84 sleeping, 1 running, 1 zombie, 0 stopped
CPU states:  2.3% user,  0.7% system,  0.0% nice, 96.8% idle
Mem:  900236K av, 546472K used, 353764K free,  0K shrd,  37552K buff
Swap: 329324K av,  34784K used, 294540K free          190764K cached
```

- The current system time:
- The "up time" of the system:
- How many users are logged in.
- The "load average" : the average number of processes ready to run over the last 1,5 and 15 minutes
- "CPU States" shows the percentage of CPU time spent in usermode, system mode and at idle.
- "MEM" shows a complete set of statistics on current memory usage.
- "SWAP" gives us the same details as "MEM" but for the swap space.

top’s lower screen

```
  PID USER  PRI  NI  SIZE  RSS SHARE STAT LIB %CPU %MEM  TIME COMMAND
10281 root   16 -10 97952 6452  1584 S <    0  3.9  0.7 56:57 X
12547 geoff  16   0  1728 1728   764 R      0  0.9  0.1  0:01 top
```

ctd…

top's lower screen

```
  PID USER  PRI  NI  SIZE  RSS SHARE STAT LIB %CPU %MEM  TIME COMMAND
10281 root   16 -10 97952 6452  1584 S <    0  3.9  0.7 56:57 X
12547 geoff  16   0  1728 1728   764 R      0  0.9  0.1  0:01 top
```

PID The process ID of each task.

ctd…

top’s lower screen

```
  PID USER  PRI  NI  SIZE  RSS SHARE STAT LIB %CPU %MEM  TIME COMMAND
10281 root   16 -10 97952 6452  1584 S <    0  3.9  0.7 56:57 X
12547 geoff  16   0  1728 1728   764 R      0  0.9  0.1  0:01 top
```

PID The process ID of each task.

USER The user name of the task’s owner.

ctd…

top's lower screen

```
  PID USER  PRI  NI  SIZE  RSS SHARE STAT LIB %CPU %MEM  TIME COMMAND
10281 root   16 -10 97952 6452  1584 S <    0  3.9  0.7 56:57 X
12547 geoff  16   0  1728 1728   764 R      0  0.9  0.1  0:01 top
```

PID The process ID of each task.

USER The user name of the task's owner.

PRI The priority of the task.

ctd…

top's lower screen

```
  PID USER  PRI  NI  SIZE  RSS SHARE STAT LIB %CPU %MEM  TIME COMMAND
10281 root   16 -10 97952 6452  1584 S <    0  3.9  0.7 56:57 X
12547 geoff  16   0  1728 1728   764 R      0  0.9  0.1  0:01 top
```

PID The process ID of each task.

USER The user name of the task's owner.

PRI The priority of the task.

NI The nice value of the task. Negative nice values are higher priority.

ctd…

top's lower screen

```
  PID USER  PRI  NI  SIZE  RSS SHARE STAT LIB %CPU %MEM  TIME COMMAND
10281 root   16 -10 97952 6452  1584 S <    0  3.9  0.7 56:57 X
12547 geoff  16   0  1728 1728   764 R      0  0.9  0.1  0:01 top
```

PID The process ID of each task.

USER The user name of the task's owner.

PRI The priority of the task.

NI The nice value of the task. Negative nice values are higher priority.

SIZE The size of the task's code plus data plus stack space, in kilobytes, is shown here.

ctd…

top's lower screen

```
  PID USER  PRI  NI  SIZE  RSS SHARE STAT LIB %CPU %MEM  TIME COMMAND
10281 root   16 -10 97952 6452  1584 S <    0  3.9  0.7 56:57 X
12547 geoff  16   0  1728 1728   764 R      0  0.9  0.1  0:01 top
```

PID The process ID of each task.

USER The user name of the task's owner.

PRI The priority of the task.

NI The nice value of the task. Negative nice values are higher priority.

SIZE The size of the task's code plus data plus stack space, in kilobytes, is shown here.

RSS The total amount of physical memory used by the task,in kilobytes, is shown here. For ELF processes used library pages are counted here, for a.out processes not.

ctd…

top's lower screen

```
  PID USER  PRI  NI  SIZE  RSS SHARE STAT LIB %CPU %MEM  TIME COMMAND
10281 root   16 -10 97952 6452  1584 S <    0  3.9  0.7 56:57 X
12547 geoff  16   0  1728 1728   764 R      0  0.9  0.1  0:01 top
```

PID The process ID of each task.

USER The user name of the task's owner.

PRI The priority of the task.

NI The nice value of the task. Negative nice values are higher priority.

SIZE The size of the task's code plus data plus stack space, in kilobytes, is shown here.

RSS The total amount of physical memory used by the task,in kilobytes, is shown here. For ELF processes used library pages are counted here, for a.out processes not.

SHARE The amount of shared memory used by the task is shown in this column.

ctd…

top’s lower screen

```
  PID USER  PRI  NI  SIZE  RSS SHARE STAT LIB %CPU %MEM  TIME COMMAND
10281 root   16 -10 97952 6452  1584 S <    0  3.9  0.7 56:57 X
12547 geoff  16   0  1728 1728   764 R      0  0.9  0.1  0:01 top
```

top's lower screen

```
  PID USER  PRI  NI  SIZE  RSS SHARE STAT LIB %CPU %MEM  TIME COMMAND
10281 root   16 -10 97952 6452  1584 S <    0  3.9  0.7 56:57 X
12547 geoff  16   0  1728 1728   764 R      0  0.9  0.1  0:01 top
```

STAT The state of the task is shown here.

The state is either

S sleeping

D uninterruptible sleep

R running

Z zombies

T stopped or trace

These states are modified by trailing < for a process with negative nice value, N for a process with positive nice value, W for a swapped out process (this does not work correctly for kernel processes).

top's lower screen

```
  PID USER  PRI  NI  SIZE  RSS SHARE STAT LIB %CPU %MEM  TIME COMMAND
10281 root   16 -10 97952 6452  1584 S <    0  3.9  0.7 56:57 X
12547 geoff  16   0  1728 1728   764 R      0  0.9  0.1  0:01 top
```

STAT The state of the task is shown here.

The state is either

S sleeping

D uninterruptible sleep

R running

Z zombies

T stopped or trace

These states are modified by trailing < for a process with negative nice value, N for a process with positive nice value, W for a swapped out process (this does not work correctly for kernel processes).

%CPU The task's share of the CPU time since the last screen update, expressed as a percentage of total CPU time per processor.

top's lower screen

```
  PID USER  PRI  NI  SIZE  RSS SHARE STAT LIB %CPU %MEM  TIME COMMAND
10281 root   16 -10 97952 6452  1584 S <    0  3.9  0.7 56:57 X
12547 geoff  16   0  1728 1728   764 R      0  0.9  0.1  0:01 top
```

STAT The state of the task is shown here.

The state is either

S sleeping

D uninterruptible sleep

R running

Z zombies

T stopped or trace

These states are modified by trailing < for a process with negative nice value, N for a process with positive nice value, W for a swapped out process (this does not work correctly for kernel processes).

%CPU The task's share of the CPU time since the last screen update, expressed as a percentage of total CPU time per processor.

%MEM The task's share of the physical memory.

`top`: selected interactive commands

^L Redraw the screen

`top`: selected interactive commands

^L Redraw the screen

f|F Add and remove fields

`top`: selected interactive commands

^L Redraw the screen

f|F Add and remove fields

h|? Displays a help screen

`top`: selected interactive commands

^L Redraw the screen

f|F Add and remove fields

h|? Displays a help screen

S Toggle cumulative mode

`top`: selected interactive commands

^L Redraw the screen

f|F Add and remove fields

h|? Displays a help screen

S Toggle cumulative mode

I Toggle between Irix and Solaris views (SMP-only)

`top`: selected interactive commands

^L Redraw the screen

f|F Add and remove fields

h|? Displays a help screen

S Toggle cumulative mode

I Toggle between Irix and Solaris views (SMP-only)

k Kill a task (with any signal)

`top`: selected interactive commands

^L Redraw the screen

f|F Add and remove fields

h|? Displays a help screen

S Toggle cumulative mode

I Toggle between Irix and Solaris views (SMP-only)

k Kill a task (with any signal)

r Renice a task

top: selected interactive commands

^L Redraw the screen

f|F Add and remove fields

h|? Displays a help screen

S Toggle cumulative mode

I Toggle between Irix and Solaris views (SMP-only)

k Kill a task (with any signal)

r Renice a task

T Sort by time / cumulative time

`top`: selected interactive commands

^L Redraw the screen

f|F Add and remove fields

h|? Displays a help screen

S Toggle cumulative mode

I Toggle between Irix and Solaris views (SMP-only)

k Kill a task (with any signal)

r Renice a task

T Sort by time / cumulative time

s Set the delay in seconds between updates

`top`: selected interactive commands

^L Redraw the screen

f|F Add and remove fields

h|? Displays a help screen

S Toggle cumulative mode

I Toggle between Irix and Solaris views (SMP-only)

k Kill a task (with any signal)

r Renice a task

T Sort by time / cumulative time

s Set the delay in seconds between updates

q Quit

top's interactive commands

space Update display

^L Redraw the screen

f|F Add and remove fields

o|O Change order of displayed fields

h|? Displays a help screen

S Toggle cumulative mode

i Toggle display of idle processes

I Toggle between Irix and Solaris views (SMP-only)

c Toggle display of command name/line

l Toggle display of load average

m Toggle display of memory information

t Toggle display of summary information

k Kill a task (with any signal)

r Renice a task

N Sort by pid (Numerically)

A Sort by age

P Sort by CPU usage

M Sort by resident memory usage

T Sort by time / cumulative time

u Show only a specific user

n|# Set the number of process to show

s Set the delay in seconds between updates

W Write configuration file /.toprc

q Quit

~/.toprc

```
$ cat toprc ↩
AbcDgHIjklMnoTP|qrsuzyV{EFWx
2
```

Killing Processes

Job Control

There are three commands and a pretzel used for job control.

- `jobs`
- `fg`
- `bg`
- `&`

Job Control

There are three commands and a pretzel used for job control.

- `jobs`
- `fg`
- `bg`
- `&`

They are bash built-ins:

```
$ type jobs fg bg  ↩
jobs is a shell builtin
fg is a shell builtin
bg is a shell builtin
```

For more information, see the Job Control section of man bash.

&— Direct the shell to execute a command in the background.

Example:

```
$ xeyes ↩
```

Notice the `xeyes` process is started in the foreground and you have no prompt. The user is locked out of further interaction with the shell until a process is stopped, terminated or completed.

Now start the xeyes process in the background.

```
$ xeyes & ↩
[1] 1650
$
```

Two numbers are listed and the prompt is now also displayed waiting for another command.

Job Control

```
$ xeyes & ↩
[1] 1650
$
```

Job Control

```
$ xeyes & ↩
[1] 1650
$
```

- The [1] is the programs job id, a unique number for the shell starting from 1.

Job Control

```
$ xeyes & ↩
[1] 1650
$
```

- The [1] is the programs job id, a unique number for the shell starting from 1.
- The 1650 is the process id (pid), which identifies the process across the entire system.

Job Control

```
$ xeyes & ↩
[1] 1650
$
```

- The [1] is the programs job id, a unique number for the shell starting from 1.
- The 1650 is the process id (pid), which identifies the process across the entire system.
- Either of these numbers can be used to interact with the program through bash.

Background Processing

The best candidates for background processing are programs that do not require user input, as these programs will keep on waiting until input is provided.

Programs that send their results to standard output (The screen), will do so even if running in the background. If the user is performing another operation, the results may be difficult to interpret. The output from these processes can be redirected to a file.

```
$ wc bigfile > bigfile.wc & ↩
[1] 1654
$
```

The `jobs` command

The `jobs` command

`$ jobs` ↩ :

Lists all commands stopped, or running in the background.

The `jobs` command

`$ jobs` ↩ :

Lists all commands stopped, or running in the background.

Options :

`-l` List pid

The `jobs` command

`$ jobs` ↩ :

Lists all commands stopped, or running in the background.

Options :

`-l` List pid

Example :

Start some processes in the background and suspend a foreground process.

```
$ jobs ↩
[1]+  Stopped                 less job_control.txt
[2]-  Running                 xeyes &
$
```

The `fg` command

`$ fg` ↩ :

Shell built-in used to force a suspended or background process to continue running in the foreground.

The `fg` command

`$ fg` ↩ :

Shell built-in used to force a suspended or background process to continue running in the foreground.

Example :

- Use the 'jobs' command to find job id.

```
$ jobs ↩
[1]+  Stopped                 less job_control.txt
[2]-  Running                 xeyes &
$
```

The `fg` command

`$ fg` ↩ :

Shell built-in used to force a suspended or background process to continue running in the foreground.

Example :

- Use the 'jobs' command to find job id.

```
$ jobs ↩
[1]+  Stopped                 less job_control.txt
[2]-  Running                 xeyes &
$
```

- Use `fg` to bring xeyes to foreground.

```
$ fg 2 ↩
xeyes
```

The `fg` command

`$ fg` ↩ :

Shell built-in used to force a suspended or background process to continue running in the foreground.

Example :

- Use the 'jobs' command to find job id.

```
$ jobs ↩
[1]+  Stopped                 less job_control.txt
[2]-  Running                 xeyes &
$
```

- Use `fg` to bring xeyes to foreground.

```
$ fg 2 ↩
xeyes
```

- A % used with the job id is equivalent to `fg 2`.

```
$ %2 ↩
xeyes
```

The `fg` command

A job can also be referred to by a string that uniquely identifies the beginning of the command line used to start a job. A '%' can also be used with a unique string.

```
$ fg x ↩
xeyes
```

or

```
$ %x ↩
xeyes
```

If `fg` is issued without any argument, the job with the '+' in the job list is brought to the foreground.

```
$ fg ↩
xeyes
```

The bg command

$ bg ↩ :

Used to force a suspended process to continue running in the background.

Job 1 shows the ’find’ command was started in the foreground and then suspended.To start ’find’ in the background, use the ’bg’ command or ’%’.

The bg command

$ bg ↩ :

Used to force a suspended process to continue running in the background.

Example :

Use the ’jobs’ command to find job id.

```
$ jobs ↩
[1]-  Stopped                 find -name myfile >myfile.found  (wd: /)
[2]+  Stopped                 less job_control.txt
[3]   Running                 xeyes &
$
```

Job 1 shows the ’find’ command was started in the foreground and then suspended.To start ’find’ in the background, use the ’bg’ command or ’%’.

The bg command

`$ bg` ↩ :

Used to force a suspended process to continue running in the background.

Example :

Use the ’jobs’ command to find job id.

```
$ jobs ↩
[1]-  Stopped          find -name myfile >myfile.found  (wd: /)
[2]+  Stopped          less job_control.txt
[3]   Running          xeyes &
$
```

Job 1 shows the ’find’ command was started in the foreground and then suspended.To start ’find’ in the background, use the ’bg’ command or ’%’.

Example :

`$ bg 1 ↩` or `$ bg f ↩` or `$ %1 & ↩` or `$ %f & ↩`

The End