Revised: 3/23/2020

Managing Processes

Michael C. Hackett

Computer Science Department

Community College of Philadelphia

Lecture Topics

- Processes
  - Monitoring Processes
  - Stopping Processes
  - Process Priority
- Background Processes
- Scheduling Processes

Processes

- **Program** and **process** are used interchangeably.
  - A **program** is an executable file
  - A **process** is a program that is currently running

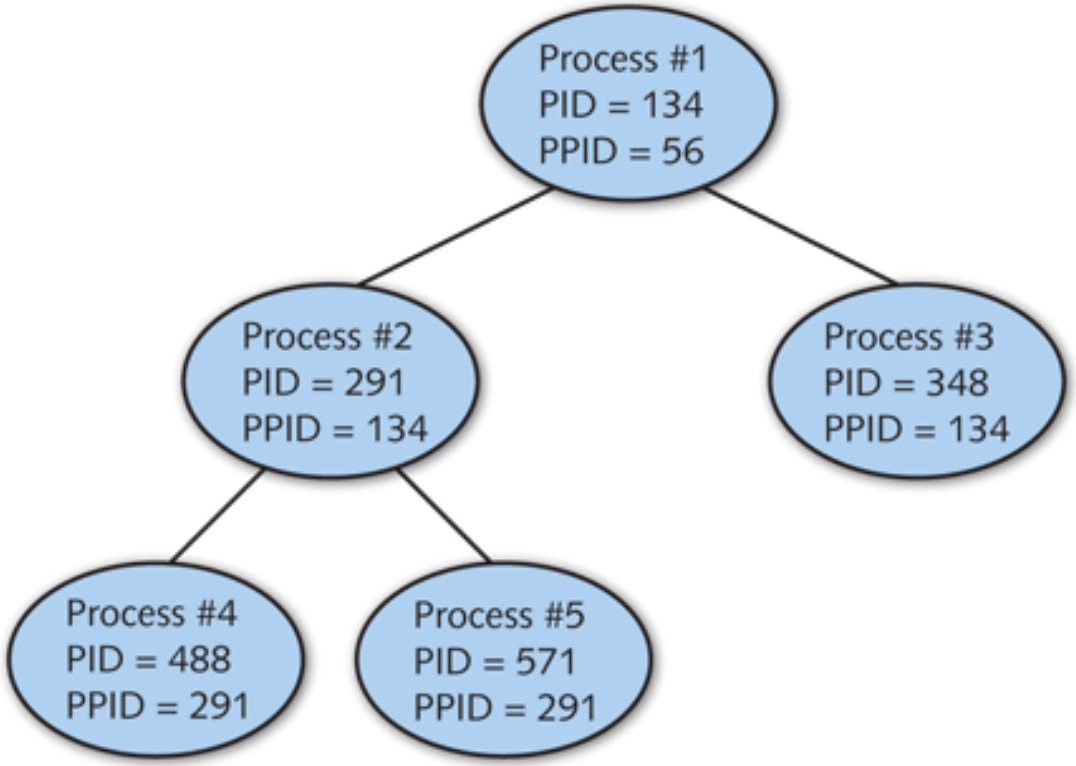
- A **user process** is a program that was started in the terminal by the logged in user
  - **`ls`**, **`grep`**, **`find`** would all be examples of user processes
- A **daemon process** is a program that provides a system service
  - Usually started on system startup
  - Daemons were discussed in the last lecture

Processes

- **Process IDs** (**PID**s) are unique numbers that allow the kernel to identify each process
  - Every process has a unique PID

- A **child process** is a process started by another process
  - A child process's **parent process** is the process that started it
  - A process's **Parent Process ID** (**PPID**) identifies the process's parent.

Processes



Parent/Child Process Relationships

Processes

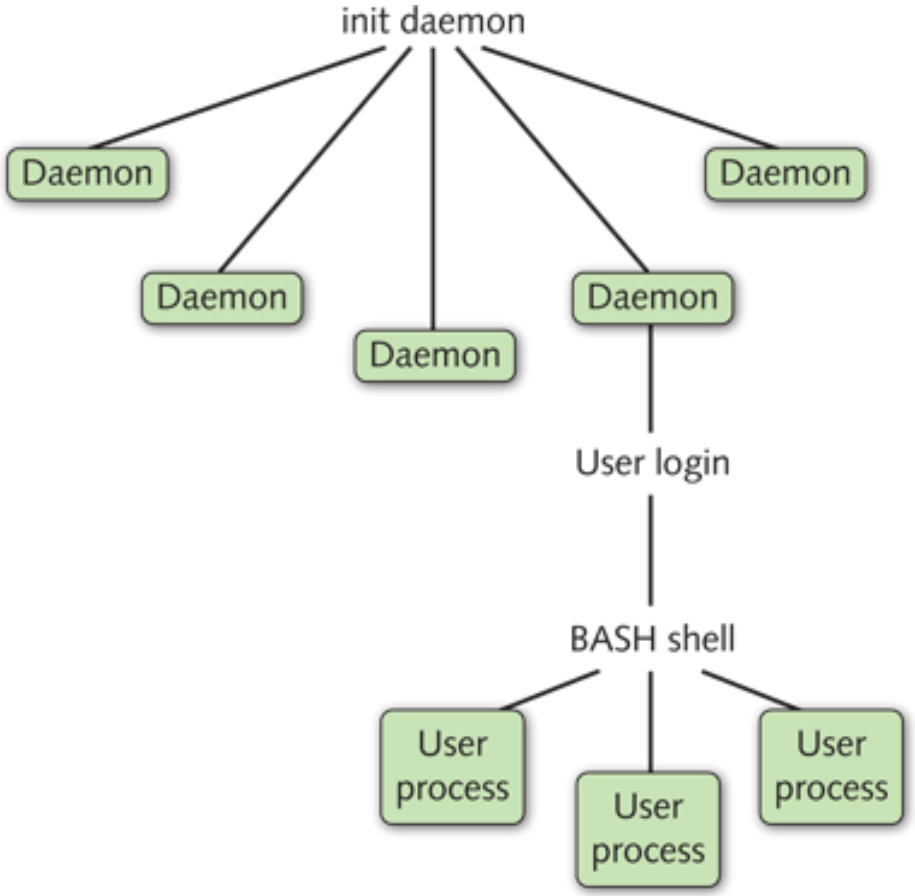
- PIDs are not assigned in sequential order
- A process can have unlimited child processes
- A process can only have one parent process

Processes

- The init deamon is always the first process started by the kernel.
  - Always has a PID of 1 and a PPID of 0
  - The PID 0 refers to the kernel itself

- Every process can be traced back to the init daemon

Processes



Process Relationships

Processes

- Processes can be either
  - Binary Programs
    - Programs and commands like **`ls`** and **`find`**
  - Shell Functions
    - Commands built into BASH like **`cd`**
  - Shell Scripts
    - A program made up of BASH/shell instructions

Processes

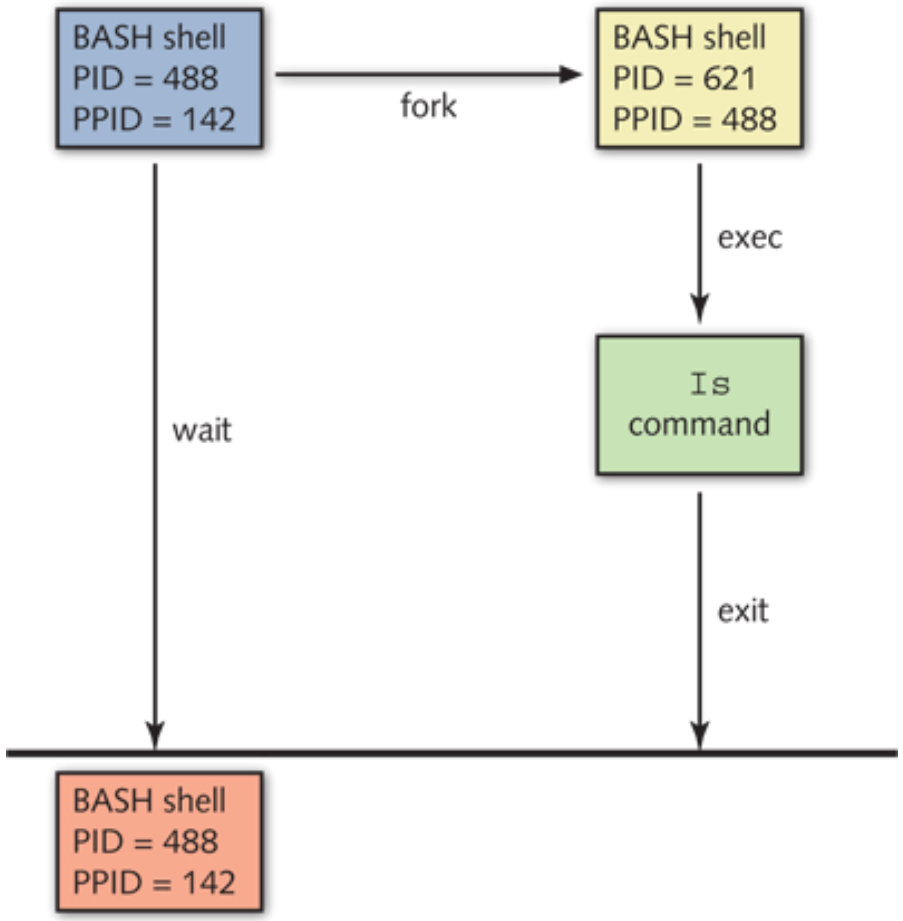
- When a process is started, the original shell:
  1. Starts a new BASH shell (subshell)
  2. Executes the command in the subshell
  3. Exits the subshell and returns to the current shell

- This procedure is called **forking**

Processes

- When executing the **`ls`** command:
  1. The current shell starts a new BASH shell (subshell)
  2. The **`ls`** command is executed in the subshell
  3. The subshell is exited and returns to the current shell

Processes


BASH shell
PID = 488
PPID = 142
fork
BASH shell
PID = 621
PPID = 488
exec
ls
command
wait
exit
BASH shell
PID = 488
PPID = 142

Monitoring Processes

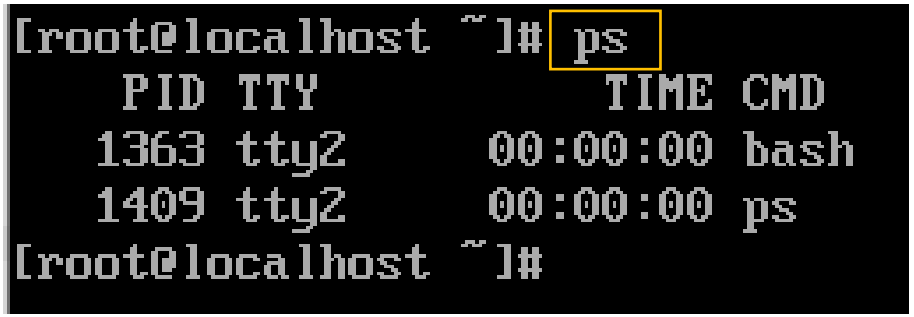
- The **ps** command is used to view processes.
  - **P**rocess **S**napshot

ps [options]

```
[root@localhost ~]# ps
    PID TTY          TIME CMD
   1363 tty2     00:00:00 bash
   1409 tty2     00:00:00 ps
[root@localhost ~]#
```

Monitoring Processes

- The **ps** command without any options will display the following for processes running in the current shell:
  - The process's Process ID (PID)
  - The terminal it was started on (TTY)
  - The time it has taken on the CPU (TIME)
  - The command that started the process (CMD)

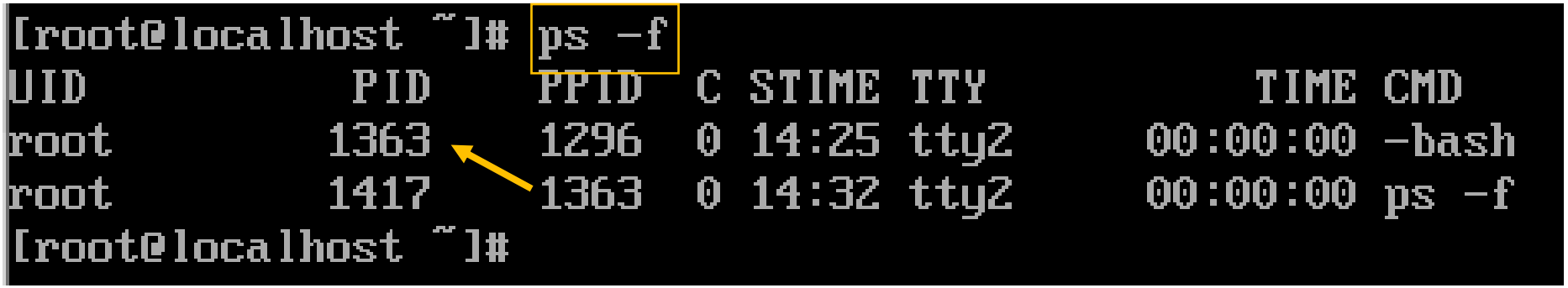


Monitoring Processes

- The **ps** command with the **-f** option will display the **f**ull listing for processes running in the current shell:
  - The user that started the process (UID)
  - The process's Process ID (PID)
  - The process's Parent Process ID (PPID)
  - The CPU utilization (C)
  - The time the process was started (STIME)
  - The terminal it was started on (TTY)
  - The time it has taken on the CPU (TIME)
  - The command that started the process (CMD)

Monitoring Processes

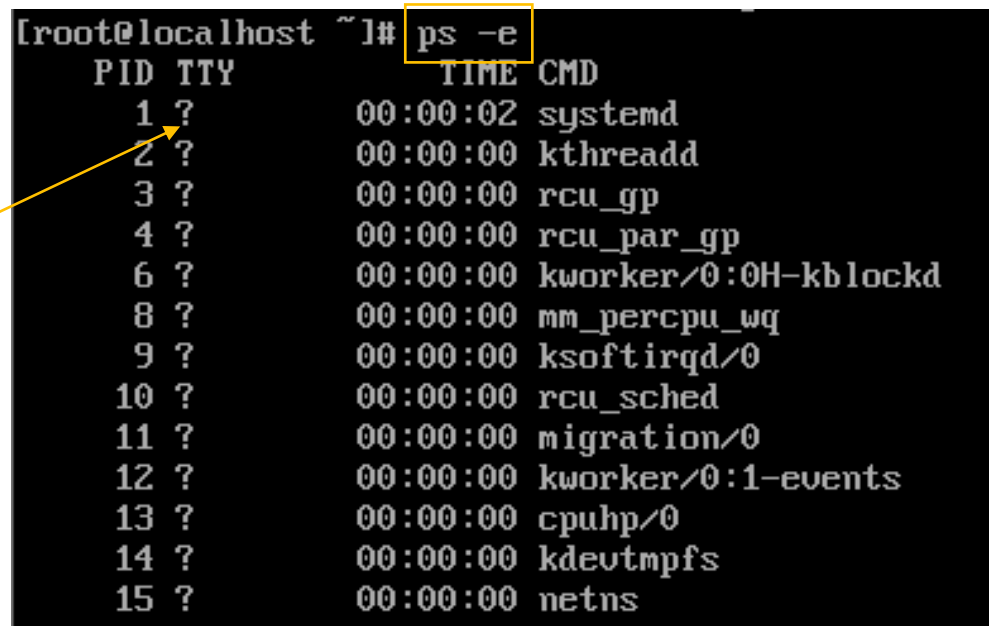
ps -f



Monitoring Processes

- The **ps** command with the **-e** option will display a listing of processes running on the **e**ntire system:

Was not started in a terminal



Monitoring Processes

- The **ps** command with the **-e** and **-f** options will display a **f**ull listing of processes running on the **e**ntire system:

```
[root@localhost ~]# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 14:23 ?        00:00:02 /usr/lib/systemd/systemd --switched-root --syste
root           2       0  0 14:23 ?        00:00:00 [kthreadd]
root           3       2  0 14:23 ?        00:00:00 [rcu_gp]
root           4       2  0 14:23 ?        00:00:00 [rcu_par_gp]
root           6       2  0 14:23 ?        00:00:00 [kworker/0:0H-kblockd]
root           8       2  0 14:23 ?        00:00:00 [mm_percpu_wq]
root           9       2  0 14:23 ?        00:00:00 [ksoftirqd/0]
root          10       2  0 14:23 ?        00:00:00 [rcu_sched]
root          11       2  0 14:23 ?        00:00:00 [migration/0]
root          12       2  0 14:23 ?        00:00:00 [kworker/0:1-events]
root          13       2  0 14:23 ?        00:00:00 [cpuhp/0]
root          14       2  0 14:23 ?        00:00:00 [kdevtmpfs]
root          15       2  0 14:23 ?        00:00:00 [netns]
root          16       2  0 14:23 ?        00:00:00 [rcu_tasks_kthre]
root          17       2  0 14:23 ?        00:00:00 [kauditd]
```

Monitoring Processes

- As seen below, the init daemon (systemd) has a PID of 1
- The kernel thread daemon (kthreadd) has a PID of 2 and is responsible for starting most kernel subprocesses
  - Most of the early processes were started by this process (shown by their PPID of 2)

```
[root@localhost ~]# ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 14:23 ?        00:00:02 /usr/lib/systemd/systemd
root           2       0  0 14:23 ?        00:00:00 [kthreadd]
root           3       2  0 14:23 ?        00:00:00 [rcu_gp]
root           4       2  0 14:23 ?        00:00:00 [rcu_par_gp]
root           6       2  0 14:23 ?        00:00:00 [kworker/0:0H-kblockd]
root           8       2  0 14:23 ?        00:00:00 [mm_percpu_wq]
```

Monitoring Processes

- For easier reading of the **`ps`** command's output, you can…

- Pipe the output to the **`less`** command to break up the output into pages

```
ps -ef | less
```

- Pipe the output to the **`grep`** command for searching

```
ps -ef | grep bash
```

Monitoring Processes

ps -ef | grep bash

```
[root@localhost ~]# ps -ef | grep bash
root        1363    1296  0 14:25 tty2     00:00:00 -bash
root        1451    1363  0 14:47 tty2     00:00:00 grep --color=auto bash
[root@localhost ~]#
```

Monitoring Processes

- The **`ps`** command with the **`-l`** option will display a **l**ong listing of processes running in the current shell:
  - Includes the same columns as the **`-f`** option
  - Also displays:
  - The process flag (F)
  - The process state (S)
  - The process's priority (PRI)
  - The process's nice value (NI)
  - The process's memory address (ADDR)
  - The process's size in memory (SZ)
  - What the process is waiting for while sleeping (WCHAN)

Monitoring Processes

ps -l

```
[root@localhost ~]# ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0    1363    1296  0  80   0 - 56674 -      tty2     00:00:00 bash
4 R     0    1461    1363  0  80   0 - 54671 -      tty2     00:00:00 ps
[root@localhost ~]# _
```

Monitoring Processes

```
[root@localhost ~]# ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0    1363    1296  0  80   0 - 56674 -      tty2     00:00:00 bash
4 R     0    1461    1363  0  80   0 - 54671 -      tty2     00:00:00 ps
[root@localhost ~]# _
```

- The Process State (S) indicates what the process is currently doing
  - S – Sleeping
  - R – Running
  - D – Waiting for disk access
  - T – Stopped or is being traced by another program
  - Z – Zombie process (Finished process waiting to be released by its parent process)

Monitoring Processes

```
[root@localhost ~]# ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0    1363    1296  0  80   0 - 56674 -      tty2     00:00:00 bash
4 R     0    1461    1363  0  80   0 - 54671 -      tty2     00:00:00 ps
[root@localhost ~]# _
```

- The Process Priority (PRI) indicates what the priority of the process is to the kernel
  - 0 – Highest Priority
  - 127 – Lowest Priority

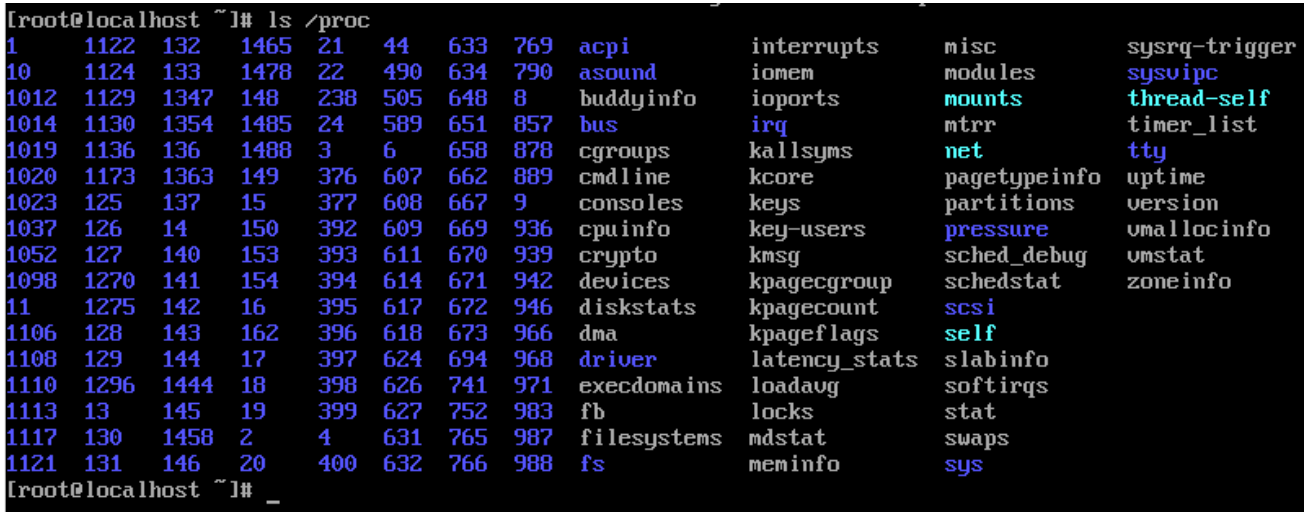
Monitoring Processes

```
[root@localhost ~]# ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0    1363    1296  0  80   0 - 56674 -      tty2     00:00:00 bash
4 R     0    1461    1363  0  80   0 - 54671 -      tty2     00:00:00 ps
[root@localhost ~]# _
```

- The Nice value (NI) indicates what the niceness of the process
  - -20 – Greater chance of being given a higher priority
  - 19 – Lesser chance of being given a higher priority

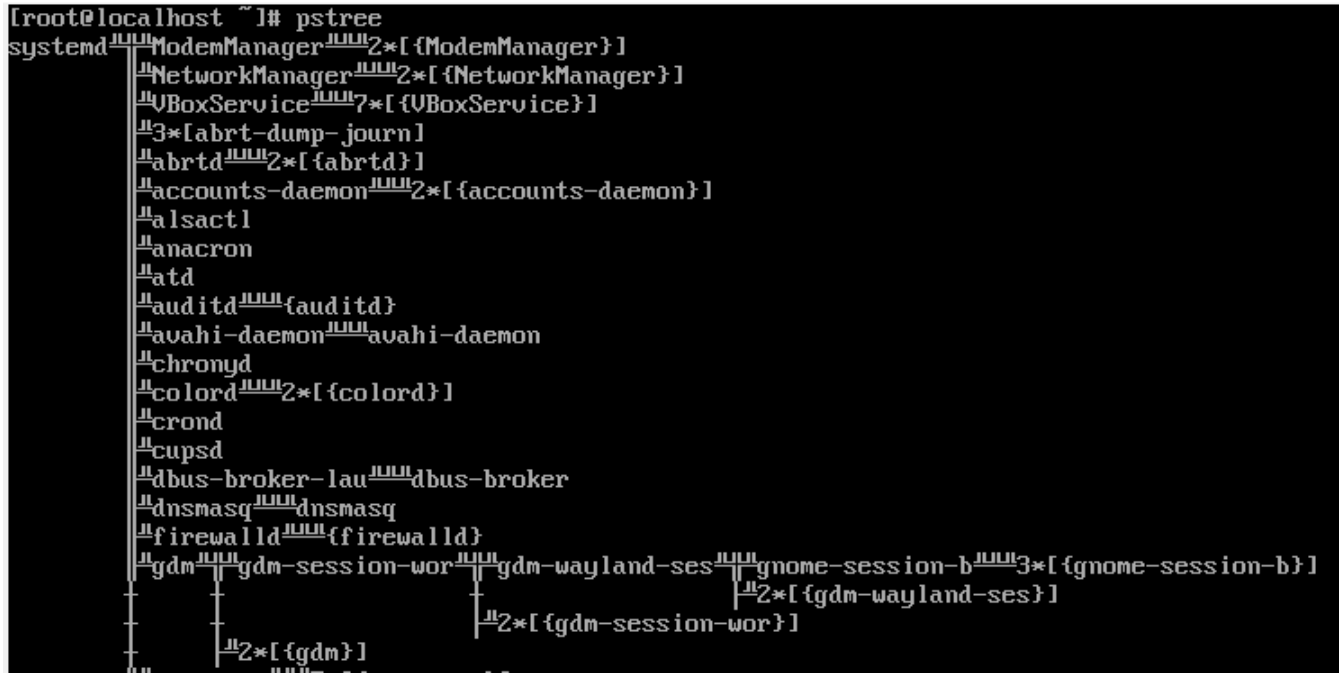
Monitoring Processes

- The contents of the **/proc** directory can also be used for viewing process information
  - Each subdirectory is the PID



Monitoring Processes

- The **`pstree`** command shows the lineage of all processes back to the init daemon



Monitoring Processes

- The **`top`** command is an interactive program for viewing and managing system processes

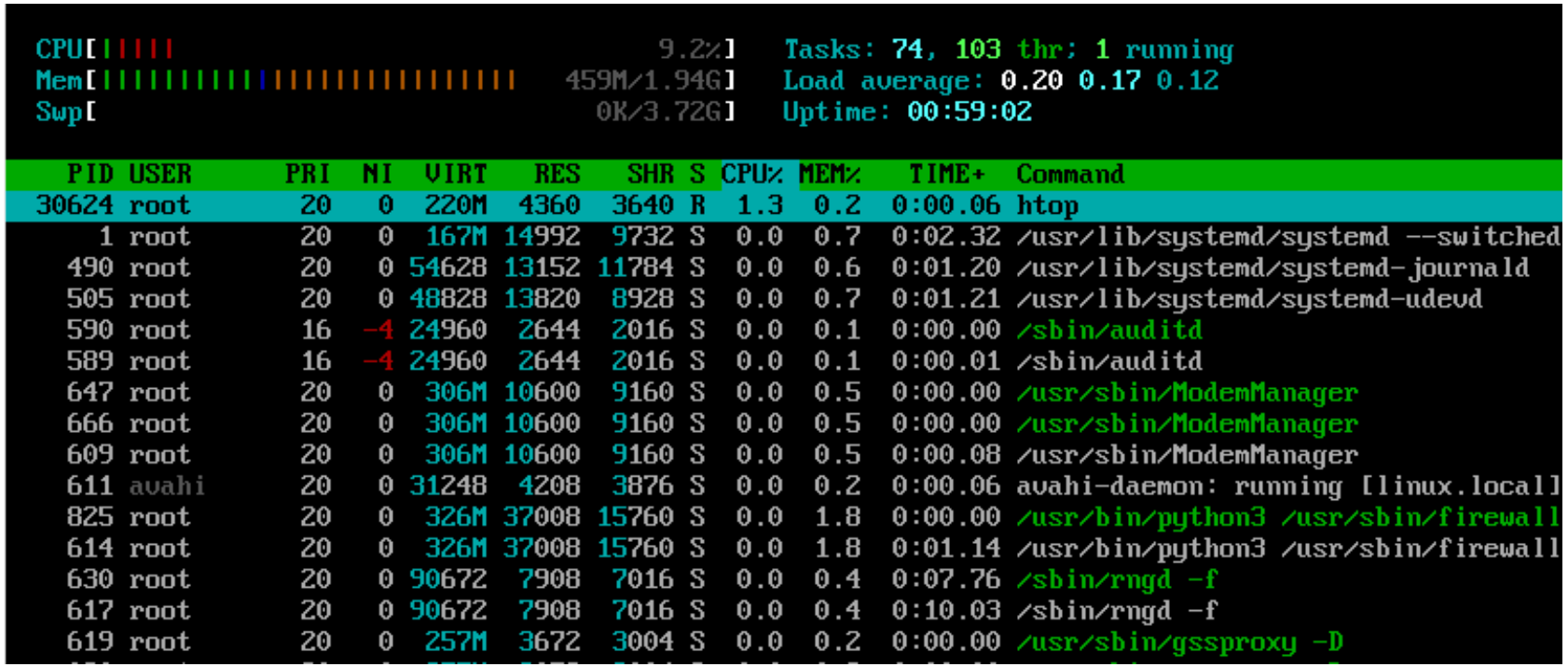
```
top - 15:17:43 up 53 min,  1 user,  load average: 0.00, 0.15, 0.11
Tasks: 136 total,   1 running, 135 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.7 sy,  0.0 ni, 94.0 id,  0.0 wa,  3.0 hi,  2.3 si,  0.0 st
MiB Mem :   1987.5 total,    858.2 free,    444.0 used,    685.3 buff/cache
MiB Swap:   3814.0 total,   3814.0 free,      0.0 used.   1381.5 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   1489 root      20   0       0      0      0 I   0.7   0.0   0:00.14 kworker/0:3-events
    741 root      20   0  620508  20000  17164 S   0.3   1.0   0:00.34 NetworkManager
   1498 root      20   0  227400   4048   3516 R   0.3   0.2   0:00.01 top
      1 root      20   0  171260  14964   9732 S   0.0   0.7   0:02.29 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-kblockd
      8 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
      9 root      20   0       0      0      0 S   0.0   0.0   0:00.21 ksoftirqd/0
     10 root      20   0       0      0      0 I   0.0   0.0   0:00.23 rcu_sched
     11 root      rt   0       0      0      0 S   0.0   0.0   0:00.00 migration/0
```

Monitoring Processes

- The **`htop`** command is an improved version of top

  `dnf install htop`



Stopping Processes

- Processes can be terminated using kill signals
- The **`kill`** command is used to send a kill signal to a process
- For a **l**isting of all kill signals, use **`kill -l`**

Stopping Processes

```
[root@localhost ~]# kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
[root@localhost ~]#
```

Stopping Processes

- The most common kill signals used are:

Kill Signal	Usage
SIGHUP (1)	Stops a process and restarts it with the same PID
SIGINT (2)	Interrupt signal (can be done with Ctrl+C in a terminal)
SIGQUIT (3)	Core dump; Sends the process information in memory to a file in the current directory
SIGKILL (9)	Absolute kill signal; Forces the process to stop executing and releases the process's resources
SIGTERM (15)	Termination signal; default kill signal

- Some processes may ignore (or **trap**) kill signals
  - SIGKILL is never ignored.

Stopping Processes

- To kill a process, the kill command requires at least the PID

`kill` *PID*

- For example: **`kill 1234`**
- Would send a SIGTERM kill signal (the default) to the process with the PID 1234

Stopping Processes

- To specify the type of kill signal:

`kill` *`-n PID`*

- For example: **`kill -9 1234`**
- Would send a SIGKILL kill signal to the process with the PID 1234

  `kill -SIGKILL 1234` would do the same thing

Stopping Processes

- Provide a comma separated list of PIDs to kill more than one process as a time:

`kill` *`–n PID, PID, PID, ...`*

- For example: **`kill -9 1234, 4321, 5467, 2322`**
- Would send a SIGKILL kill signal to the processes with the PIDs 1234, 4321, 5467, and 2322

Stopping Processes

- The **`pgrep`** command is used to search for processes based on a regular expression

- Returns a list of PIDs matching the search expression

- For example: **`pgrep "^bash"`** would return a list of PIDs for any processes whose name begins with **“bash”**

Stopping Processes

```
[root@localhost ~]# pgrep "^bash"
1363
[root@localhost ~]#
```

- The **-u** option allows **pgrep** to only search processes started by a certain user

- For example: **pgrep -u admin "^bash"** would return a list of PIDs for any processes started by the user **admin** where the process name(s) begins with **“bash”**

Stopping Processes

- The **`killall`** command is used just like the **`kill`** command, but the process name is provided instead of the PID

- For example: **`killall -9 bash`**
- Would send a SIGKILL kill signal to any processes named **bash**

Stopping Processes

- The **`pkill`** command is used to kill a process by name, using a regular expression
- For example: **`pkill -9 "^bash"`**
  - Would send a SIGKILL kill signal to any processes whose names begin with "**bash**“
- For example: **`pkill -u admin -9 "^bash"`**
  - Would send a SIGKILL kill signal to any processes whose names begin with "**bash**“ that were started by the user **admin**

Stopping Processes

- If you kill a process that has children:
  1. Any child processes will be killed
  2. The process itself is killed

- Processes executed with the **nohup** command will be started without association with the parent process that started it
  - This prevents the process from being killed in the even the process that started it is killed
- Example: **nohup** *command*

Stopping Processes

- In **`top`**, a process can be killed by:
  1. Pressing the k key
  2. Entering the PID
  3. Entering the kill signal

Process Priority

- **Time slices** are the amount of time (usually in milliseconds) that a process has with the computer's CPU

- The more time slices given to a process, the more time it has with the CPU, and the faster the process finishes.

- A process's **priority** dictates how often it is given CPU time slices

Process Priority

- The priority of a process cannot be changed directly.

- However, the priority of a process can be influenced based on the process's **niceness** value
  - A process's niceness value has a range of -20 to 19
  - -20 is the least nice value (Most likely to be given higher priority)
  - 19 is the most nice value (Most likely to be given lower priority)

- The idea is that "nicer" processes don't hog the CPU time slices

Process Priority

- Processes are started with a niceness value of 0 by default.

```
[root@localhost ~]# ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0    1363    1296  0  80   0 - 56674 -      tty2     00:00:00 bash
4 R     0   30720    1363  0  80   0 - 54671 -      tty2     00:00:00 ps
[root@localhost ~]#
```

Process Priority

- The **`nice`** command is used to start a process and specify its niceness value
- Usage: **`nice -n`** *X command*
  - Where *X* is the niceness value (-20 through 19)

```
[root@localhost ~]# nice -n 15 ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0    1363    1296  0  80   0 - 56674 -      tty2     00:00:00 bash
4 R     0   30753    1363  0  95  15 - 54671 -      tty2     00:00:00 ps
```

- Only root can run the **`nice`** command

Process Priority

- Using the **`nice`** command without the **`-n`** option will assume a niceness of 10

```
[root@localhost ~]# nice ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0    1363    1296  0  80   0 - 56674 -      tty2     00:00:00 bash
4 R     0   30757    1363  0  90  10 - 54671 -      tty2     00:00:00 ps
```

Process Priority

- The **`renice`** command is used to change the niceness value of an already running process or processes
- Usage: **`renice`** *`±X PIDs`*
  - Where *±X* is the niceness value (-20 through +19)

- If changing the niceness of more than one process, provide a comma separated list of PIDs
- Only root can run the **`renice`** command

Process Priority

- Examples:

  `renice -15 1234`

  - Changes the niceness of process with PID 1234 to -15

  `renice +7 1234`

  - Changes the niceness of process with PID 1234 to 7

  `renice +16 1234, 4321, 2233`

  - Changes the niceness of processes with PIDs 1234, 4321, and 2233 to 16

Process Priority

- The **`renice`** command is can also be used to change the niceness values of any processes by user (**`-u`** option) and/or group (**`-g`** option)
- Examples:

  `renice -15 -u admin`
  - Changes the niceness of all processes started by the user **admin** to -15

  `renice +7 -g sys`
  - Changes the niceness of all processes started by users in the **sys** group to 7

Process Priority

- Examples:

  `renice -15 -u admin, account`

  - Changes the niceness of all processes started by the users **admin** and **account** to -15

  `renice +7 -g sys -u admin, account`

  - Changes the niceness of all processes started by users in the **sys** group to 7 and all processes started by the users **admin** and **account** to 7

Process Priority

- Examples:

  `renice +2 -g sys -u admin, account -p 3546, 2156`

  - Changes the niceness of
    - All processes started by users in the **sys** group to 2
    - All processes started by the users **admin** and **account** to 2
    - Processes with the PIDs **3546** and **2156** to 2

Background Processes

- When you execute a command in the shell, the command executes in a subshell before returning to the original shell

- While the command is executing, the original shell waits until the command in the subshell finishes

- As an example, think about running the **`dnf update`** command to update the system.
  - You can't enter commands into the shell's prompt until the **`dnf`** command is finished

Background Processes

- A **foreground process** (like the previous example) is a process that causes the shell that started it to wait until the process is finished before accepting new commands.

- A **background process** is when a command or process does not force the shell that started to wait until it is finished.

Background Processes

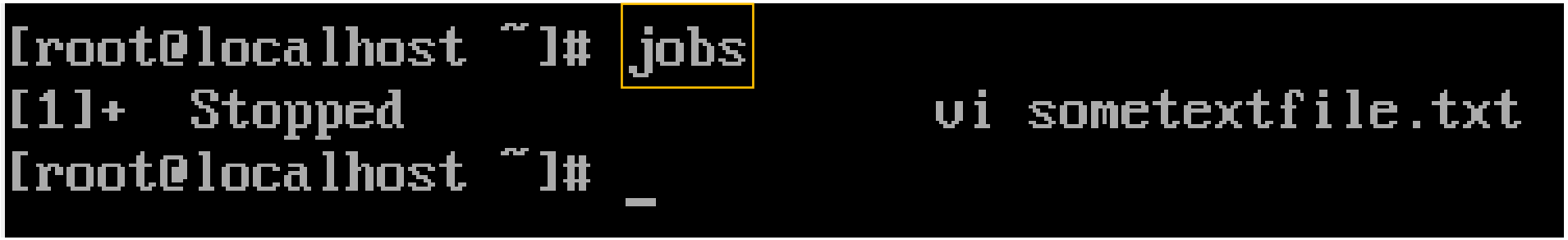
- To start a command in the background, place an ampersand (**&**) at the end of the command

```
[root@localhost ~]# vi sometextfile.txt &
[1] 30902
[root@localhost ~]# _
```

- The output is:
  - Background job ID (the number in brackets)
  - The background job's PID

Background Processes

- The **jobs** command will display a list of all background processes



Background Processes

- To bring a job in the background to the foreground, the **`fg`** command is used
- Usage: **`fg %`***X*
- Where *X* is the background job ID

- In this example:

  `fg %1`

  would bring vi back to the foreground

Background Processes

```
[root@localhost ~]# vi sometextfile.txt &
[1] 30902
[root@localhost ~]# jobs
[1]+  Stopped                 vi sometextfile.txt
[root@localhost ~]# fg %1
```

Background Processes

- To pause any running process and send it to the background, press Ctrl+Z

Ctrl+Z Pressed

Background Processes

- When a process is paused and sent to the background using Ctrl+Z, the **bg** command is used to start the process and allow it to continue running while it is in the background
- Usage: **bg** *%X*
- Where *X* is the background job ID

Background Processes

- The kill command is again used to kill background processes
- Usage: **`kill –`*`n`*` %`*`X`*
- Where *n* is the kill signal number
- Where *X* is the background job ID

- For example:

```
kill –9 %1
```

would send the SIGKILL signal to the process with background job number 1

Scheduling Processes

- Processes can be scheduled to automatically run in the future.

- The **at daemon** (**atd**) is used to schedule a command or process to run *once* in the future
- The **cron daemon** (**crond**) is used to schedule a *recurring* command or process to run

Scheduling Processes

- The **`at`** command schedules a command or process to run once the time it was provided

- Usage Examples:

  `at 10:30pm`

  - Schedules a command/process to run at 10:30pm

  `at 10:30pm July 4`

  - Schedules a command/process to run July 4th at 10:30pm

  `at 10:30pm 07/04/2022`

  - Schedules a command/process to run July 4th, 2022 at 10:30pm

Scheduling Processes

- Usage Examples:

  `at tomorrow`

  - Schedules a command/process to run at the current time tomorrow

  `at now`

  - Schedules a command/process to run immediately

  `at noon`

  - Schedules a command/process to run at 12:00pm

  `at midnight`

  - Schedules a command/process to run at 12:00am

Scheduling Processes

- Usage Examples:

  `at now + 5 minutes`

  - Schedules a command/process to run 5 minutes in the future

  `at now + 6 hours`

  - Schedules a command/process to run 6 hours in the future

  `at now + 2 weeks`

  - Schedules a command/process to run 2 weeks in the future

  `at now + 4 months`

  - Schedules a command/process to run 4 months in the future

Scheduling Processes

- When the **at** command is used, an **at>** prompt appears
- Enter the commands you wish to have executed at the specified time
- Press Ctrl+D when finished

Ctrl+D Pressed

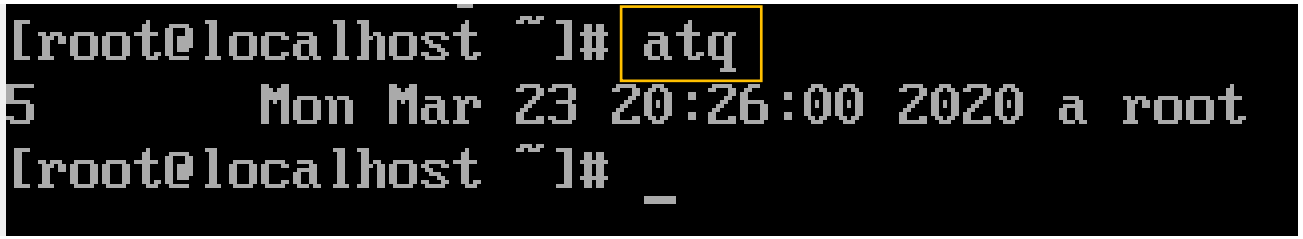
```
[root@localhost ~]# at now + 5 minutes
warning: commands will be executed using /bin/sh
at> ps -ef | grep bash > atfile
at> who >> atfile
at> <EOT>
job 5 at Mon Mar 23 20:26:00 2020
[root@localhost ~]# _
```

5 minutes later…

```
[root@localhost ~]# cat atfile
root       31064   31049  0 20:20 tty2     00:00:00 -bash
root       31132   31124  0 20:25 ?        00:00:00 /bin/bash
root       31134   31132  0 20:25 ?        00:00:00 grep bash
root     tty2         2020-03-23 20:20
[root@localhost ~]#
```

Scheduling Processes

- To list the jobs scheduled with **`at`** command, use the **`atq`** or **`at -l`** commands



Scheduling Processes

- To remove a job scheduled with **`at`**, use the following:

  `at -d` *X*

  - Where *X* is the job ID
  - The job ID is displayed when the process is first scheduled and can be found again using **`atq`** or **`at -l`**

```
[root@localhost ~]# at now + 5 minutes
warning: commands will be executed using /bin/sh
at> ps -ef | grep bash > atfile
at> who >> atfile
at> <EOT>
job 5 at Mon Mar 23 20:26:00 2020
[root@localhost ~]# _
```

```
[root@localhost ~]# atq
5       Mon Mar 23 20:26:00 2020 a root
[root@localhost ~]# _
```

Scheduling Processes

- If the following two files do not exist, then only the root user may schedule jobs using **`at`**
  - /etc/at.allow
  - /etc/at.deny

- If only /etc/at.allow exists, then only users listed in /etc/at.allow are allowed to schedule jobs with **`at`**
- If only /etc/at.deny exists, then only users not listed in /etc/at.deny are allowed to schedule jobs with **`at`**
- If both files exist, only /etc/at.allow is used

Scheduling Processes

- The cron daemon is used to schedule recurring processes.
- Unlike the at daemon which only schedules a process to run once in the future
- Cron manages scheduled processes in files that contain information on when to run the scheduled process
- The file organizes this information in a format called a **cron table**

Scheduling Processes

- Each entry ("*cron job*") in a cron table consists of six elements, each separated by a space or tab
  - The first value is the minutes past the hour (0-59)
  - The second value is the hour (0-23)
  - The third value is the day of the month (1-31)
  - The fourth value is the month of the year (1-12)
  - The fifth value is the day of the week (0-6)
    - 0 or 7 = Sunday
    - 1 = Monday
    - 2 = Tuesday
    - etc.
  - The sixth value is the absolute path the program to run
- An asterisk (*) indicates not applicable.

Scheduling Processes

- Example cron table entry:

```
15   13   *   *   *   /bin/program
```

Minute	Hour	Day of Month	Month	Day of Week	Command
15	13 (1:00PM)	N/A	N/A	N/A	/bin/program

- This cron table entry will run /bin/program every day at 13:15 (1:15PM), regardless of what month it is, what day of the month it is, and what day of the week it is

Scheduling Processes

- Example cron table entry:

```
15   13   *   *   4   /bin/program
```

Minute	Hour	Day of Month	Month	Day of Week	Command
15	13 (1:00PM)	N/A	N/A	Thursday	/bin/program

- This cron table entry will run /bin/program every Thursday at 13:15 (1:15PM), regardless of what month it is and what day of the month it is

Scheduling Processes

- Example cron table entry:

```
15   13   *   2   4   /bin/program
```

Minute	Hour	Day of Month	Month	Day of Week	Command
15	13 (1:00PM)	N/A	February	Thursday	/bin/program

- This cron table entry will run /bin/program every Thursday at 13:15 (1:15PM) in the month of February, regardless of what day of the month it is

Scheduling Processes

- Example cron table entry:

```
15   13   *   2   4,5   /bin/program
```

Minute	Hour	Day of Month	Month	Day of Week	Command
15	13 (1:00PM)	N/A	February	Thursday, Friday	/bin/program

- This cron table entry will run /bin/program every Thursday and Friday at 13:15 (1:15PM) in the month of February, regardless of what day of the month it is

Scheduling Processes

- Example cron table entry:

```
30  2  1  *  *  /bin/program
```

Minute	Hour	Day of Month	Month	Day of Week	Command
30	2 (2:00AM)	1	N/A	N/A	/bin/program

- This cron table entry will run /bin/program on the first of each month at 2:30AM, regardless of month it is or what day of the week it is

Scheduling Processes

- Example cron table entry:

```
15,30  2  1,15  *  *  /bin/program
```

Minute	Hour	Day of Month	Month	Day of Week	Command
15, 30	2 (2:00AM)	1, 15	N/A	N/A	/bin/program

- This cron table entry will run /bin/program on the first and fifteenth of the month at 2:15AM and 2:30AM, regardless of month it is or what day of the week it is

Scheduling Processes

- The cron daemon uses two types of cron tables.

- **User cron tables** contain jobs scheduled by individual user accounts
  - Fedora: /var/spool/cron
  - Ubuntu: /var/spool/cron/crontabs

- **System cron tables** are scheduled system-wide jobs and tasks
  - /etc/crontab and /etc/cron.d/

Scheduling Processes

- A user can edit their cron table file using the **`crontab`** command with the **`-e`** option
  - This will open the user's cron table file in vi
  - Cron job entries (in the format previously shown) can be added, edited, or removed from the file

```
crontab –e
```

- File location: /var/spool/cron/*username*

Scheduling Processes

- A user can list their cron table file's contents using the **`crontab`** command with the **`-l`** option

```
crontab -l
```

Scheduling Processes

- A user can clear their cron table using the **`crontab`** command with the **`-r`** option
  - This will remove all of their cron table entries

```
crontab -r
```

Scheduling Processes

- The root user can access a user's cron table file using the **`crontab`** command with the **`-u`** option
  - This can be used with the **`-l`**, **`-e`**, or **`-r`** options

`crontab –e –u` *`user`*

`crontab –l –u` *`user`*

`crontab –r –u` *`user`*

Scheduling Processes

- To schedule system-wide jobs and tasks, the system cron table file is /etc/crontab
- This file allows seven values for each entry
  - Before the command, the user account to run the command as can be specified
- Example cron table entry:

```
30  2  1  *  *  admin /bin/program
```

Minute	Hour	Day of Month	Month	Day of Week	User to run command as	Command
30	2 (2:00AM)	1	N/A	N/A	admin	/bin/program

Scheduling Processes

- Another way to schedule system-wide jobs and tasks is to put scripts (or links to scripts) in the following directories:
  - /etc/cron.hourly – Executes its scripts 1 minute past every hour
  - The /etc/cron.hourly/0anacron script starts the anacron daemon which executes the scripts in:
    - /etc/cron.daily – Configured /etc/anacron
    - /etc/cron.weekly – Configured /etc/anacron
    - /etc/cron.monthly – Configured /etc/anacron

Scheduling Processes

- The cron daemon assumes the system is running continuously without interruption
  - If the system is down/powered off, scheduled processes that did not execute at their specified time will NOT be executed when the system restarts

- The anacron daemon is like cron, but WILL execute scheduled processes that were not started because the system was powered off
  - Anacron can’t schedule tasks on a minute or hour basis
  - It can only schedule in terms of days, weeks, or months

Scheduling Processes

- If the following two files do not exist, then *any* user may schedule jobs using **`crontab`**
  - /etc/cron.allow
  - /etc/cron.deny

- If only /etc/cron.allow exists, then only users listed in /etc/cron.allow are allowed to schedule jobs with **`crontab`**
- If only /etc/cron.deny exists, then only users not listed in /etc/cron.deny are allowed to schedule jobs with **`crontab`**
- If both files exist, only /etc/cron.allow is used