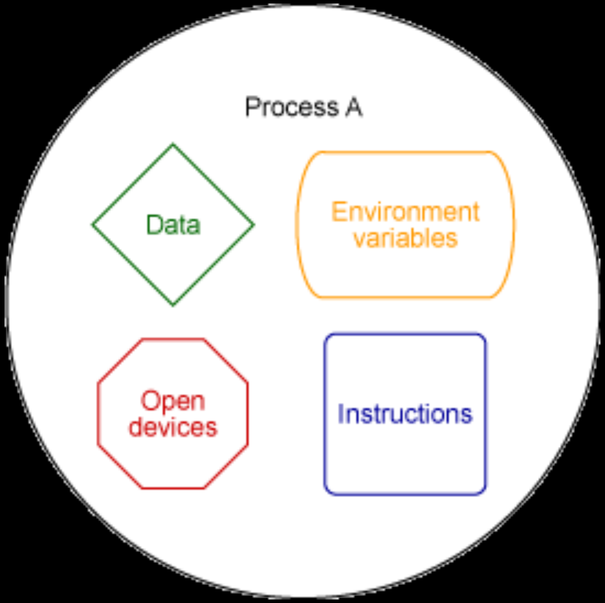
Process Management

forks, bombs, zombies, and daemons!

Lecture 5, Hands-On Unix System Administration DeCal
2012-10-01

what is a process?

- an *abstraction!*
- you can think of it as a program in the midst of execution
    - but also much more than just that!



what is a process?

- "living result of running program code"
- processes are born, they give birth to other processes, and also die.
- kernel is responsible for their management
  - resource allocation, process scheduling, etc.

relationships among processes

- A process is always created by another process.
- with the exception of `init`, executed directly by the kernel during the boot process.
  - `init` is the parent/grandparent of all processes, responsible for spawning all necessary processes upon system startup
- A process can spawn multiple children
- hierarchical structure
  - `pstree`, `ps auxf`

exit status

- children processes return a numeric status value to their parents
- exit statuses can tell the parent process if the command succeeded or failed
- Many (but not all) commands return a status of 0 if it succeeded or non-zero if something went wrong
- in `bash`, `` `echo $?' `` to obtain exit status of previous command
- common exit codes:

```
0       -- success!
1       -- a catch-all for general errors
127     -- command not found
130     -- termination by Ctrl+C
```

process attributes

- process information stored internally in a **process table**
- A process keeps its entry in the process table until it dies (properly)
- Some process attributes include:
  - `PID` (process-id): each process identified by a unique integer
  - `PPID` (parent-PID): PID of the parent
  - process states (see ``man ps'` for a complete list)
    - **(R) Running:** running or ready to run
    - **(S) Interruptible:** a **blocked** state of a process and waiting for an event or signal from another process
    - **(D) Uninterruptible**: a blocked state; process can't be killed or interrupted, usually
    - **(T) Stopped**: Process is stopped or halted and can be restarted by some other process
    - **(Z) Zombie**: process terminated, but information is still there in the process table.

ps

- "process status"
- obtain information on processes currently running on the system
- options vary! they differ among different distributions
- read the man page!! fields are also explained there!
  - in particular there are 3 sets of options in ps,
    - UNIX options, preceded by a -
      - ps -ef # display all processes running on the system, in full format listing
      - ps -u # display processes you are running (or specify a user)
    - BSD options, no dash!
      - ps aux # display all processes running on the system
    - GNU long options, -- (two dashes)

why you can't kill zombies.

- How do zombie processes arise? What's a zombie process?
  - harmless dead child process that whose entry still exists in the process table
  - can't exactly kill them because they're **already dead**.
- parent usually picks up its children's exit statuses
- To remove these process table entries occupied by zombies, try sending a SIGCHLD signal to the parent manually (`kill -s CHLD <parent pid>`)
  - if a misbehaving parent doesn't pick up its dead child's exit status
    - child turns into zombie.
  - a good parent reaps its dead children.

orphans

- a process becomes an orphan when its parent dies before it does
- kernel makes `init` the parent of all orphans
- the orphan gets adopted by init

daemons

- system-related background processes, no direct user interaction needed
- often started on system startup
- often run with the permissions of root
- services requests from other processes.
- usually waiting for something to happen
  - eg, printer daemon is waiting for print commands.
- examples:
  - sshd (listens for ssh connections from clients),
  - cupsd (printing system daemon)
  - httpd (web server daemon)

fork bombs

- `fork()` -- create new, identical child process
- form of denial of service (DoS) attack
- ‘explodes’ by recursively spawning copies of itself rapidly
- exhausts process table entries
  - can't create anymore processes

The classic example

```
:(){ :|: & };:
```

which is basically (in human readable form)

```
bomb() {
 bomb | bomb &
}; bomb
```

disclaimer: I am not responsible if you crash your laptop.

preventing fork bombs

- limit resource usage.
- limiting the number of processes a user can have
- examples:
  - `/etc/security/limit.conf`
  - `ulimit -u`

process management

- cron
- kill
- job control

process signal handling

- processes can receive **signals**
- provides limited inter-process communication
- often used to communicate **occurrence of an event**
- represented by numeric values (system-dependent)
- `kill -l` to see available signals + corresponding numeric values on your system
- commonly used signals (See `` `man 7 signal' `` for more!)

```
    1           SIGHUP         hangup
    2           SIGINT         keyboard interrupt
    9           SIGKILL        kill signal
    15          SIGTERM        termination signal
    19,18,25    SIGSTOP        stop process
    18,20,24    SIGSTP         stop typed at tty
    17,19,23    SIGCONT        continue if stopped
```

`ctrl+c` `sends` **`SIGINT`** `to a process (interrupt)`

`ctrl+z` `sends` **`SIGSTP`**

signal handling, cont.

- processes can react to received signals
  - terminate
  - ignore it
  - trap the signal (process invokes a signal handling function)

`kill`

- kill processes
  - (but only processes you have permission to kill)
- but can do more than just that!
  - send signals to processes
  - `kill -l` lists all the signals you can send
  - `kill -s <signal> <pid>`
  - alternatively, kill -<signal number> <pid>
    - `kill -s SIGKILL <pid>`
    - `kill -9 <pid>`
    - without args, default is to send `SIGTERM`

`SIGTERM vs. SIGKILL`

- what's the difference between:
  - `kill <PID>`
  - `kill -9 <PID>`
- A note about kill -9:
  - generally, you should kill -15 (default) before kill -9 to give process chance to clean up after itself (SIGTERM is more "polite")
    - release file handles
    - remove temporarily files, etc.
  - processes can't catch or ignore `SIGKILL`,
  - but often ignore or catch `SIGTERM`

stubborn processes

- when kill -9 doesn't work
  - perhaps process is already a zombie
  - perhaps process is in uninterruptible sleep (D)
  - killing the zombie's parent process will re-parent the zombie to init, which regularly reaps its zombie children. (btw, that's another one of init's jobs)

killing processs -- other useful commands

- `killall`, `pkill` – send signals/kill process based on name instead of pid
- `pgrep` -- find processes based on name
- `pgrep -l` shows both process name and PID

Job control

- **job** -- group of processes
- multitasking -- we can run more than one job at a time
- relegate jobs to the **background**, run jobs in the **foreground**
- appending ampersand (&) after a command runs it in the background, in parallel with the shell
    - foreground processes prevents shell from running another command and returning the prompt until it terminates.
- shell keeps track of all bg processes it spawns
    - type `` `jobs' `` to see a list

examples

```
$ sleep 10 & sleep 10 & sleep 10 &
[1] 16843
[2] 16844
[3] 16845

$ jobs
[1] Running      sleep 10 &
[2] Running      sleep 10 &
[3] Running      sleep 10 &
```

job control, cont.

- job identified by its **job-id**
- this is different from the PID
- bring a job back to the foreground with fg, background with bg
- you can suspend a foreground process with `ctrl+z (SIGSTP)`
- refer to a job with %

  ```
  $ fg %<job id>
  ```

  - make background job run in the foreground

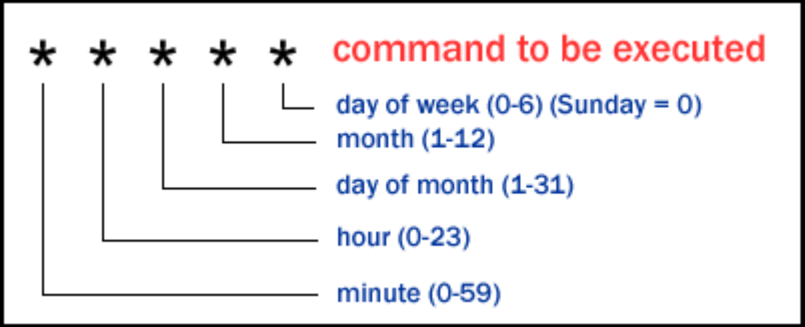
  ```
  $ bg % <job id>
  ```

  - make process running in the foreground run in the background.
  - you'd typically suspend the foreground process with ctrl+z, and then run bg to let the job continuing running in the background

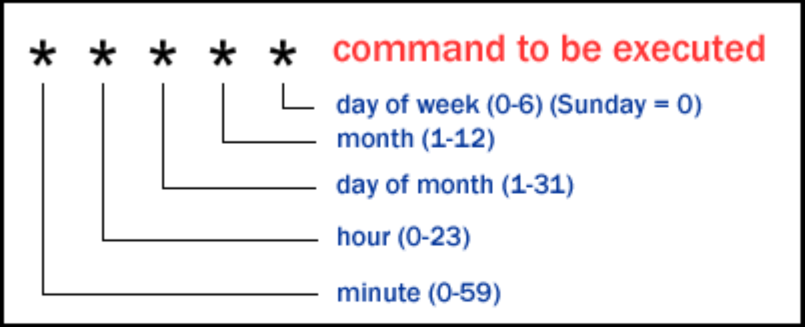
```
$ sleep 10 & sleep 10 & sleep 10 &
[1] 16843
[2] 16844
[3] 16845
```

cron

- periodic scheduler
- every scheduled job is specified as a single line in a crontab
- to edit entries in a crontab, run crontab -e
- each user typically has their own crontab (although you probably don't have permission to do this on your instructional accounts)
- Components of a crontab entry:


* * * * * command to be executed
day of week (0-6) (Sunday = 0)
month (1-12)
day of month (1-31)
hour (0-23)
minute (0-59)

cron, cont.


* * * * * command to be executed
day of week (0-6) (Sunday = 0)
month (1-12)
day of month (1-31)
hour (0-23)
minute (0-59)

```
* = matches any valid value
* * * * * = every minute, every hour, every day of the
month, every month, every day of the week

you can specify ranges, groups of values:
   00-10 17 * 3,6,9,12 * <command>
<command> runs every minute from 17:00 - 17:10 every day
for march,june,sept, dec.
```

system run levels

- refers to a mode of operation, determines which programs are executed at startup
- exact run levels vary across distributions
- changing runlevels (can't run this without proper privileges, of course)
  - `telinit <run level>` or
  - `init <run level>`
- Typical run levels:

0	halt
1	single user mode
2-5	typically multi-user-mode, with various options disabled/enabled (eg., networking)
6	reboot