# Process Management & Zombies

Benjamin Brewster

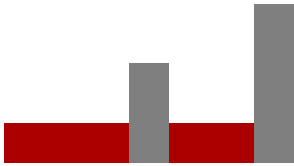
# Running Processes

- How can we tell which processes are running? Use the `ps` command to get information about currently running processes

- Ps by itself is really boring, and not all that useful:

```
$ ps
  PID TTY          TIME CMD
18779 pts/8    00:00:00 bash
18934 pts/8    00:00:00 ps
```

- I've put together my two favorite ways to run it on the next few slides



# ps For Me

Reminder: these aliases go into your ~/.bashrc file

```
$ alias
…
alias psme='ps -o ppid,pid,euser,stat,%cpu,rss,args | head -n 1; ps -eH -o
ppid,pid,euser,stat,%cpu,rss,args | grep brewsteb'
…
$ psme
 PPID   PID EUSER    STAT %CPU   RSS COMMAND
 4533 18776 root     Ss    0.2  4284     sshd: brewsteb [priv]
18776 18778 brewsteb S     0.0  2112       sshd: brewsteb@pts/8
18778 18779 brewsteb Ss    0.0  2044         -bash
18779 18911 brewsteb R+    4.0  1840           ps -eH -o ppid,pid,euser,stat,%cpu,rss,args
18779 18912 brewsteb S+    0.0   820           grep brewsteb
```

- PPID Parent Process ID
- PID Process ID
- EUSER Effective User ID
- STAT Execution State
- %CPU Percentage of CPU time this process occupies
- RSS Real Set Size - kilobytes of RAM in use by this process
- Command The actual command the user entered

# ps For Me

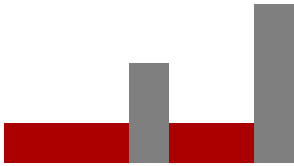
```
$ psme
 PPID   PID EUSER    STAT %CPU   RSS COMMAND
 4533 18776 root     Ss    0.2  4284     sshd: brewsteb [priv]
18776 18778 brewsteb S     0.0  2112       sshd: brewsteb@pts/8
18778 18779 brewsteb Ss    0.0  2044         -bash
18779 18911 brewsteb R+    4.0  1840           ps -eH -o ppid,pid,euser,stat,%cpu,rss,args
18779 18912 brewsteb S+    0.0   820           grep brewsteb
```

## First State Character:

- `D` Uninterruptible sleep (usually IO)
- `R` Running or runnable (on run queue)
- `S` Interruptible sleep (waiting for an event to complete)
- `T` Stopped, either by a job control signal or because it is being traced
- `Z` Defunct ("zombie") process, terminated but not reaped by its parent

## Second State Character (Optional):

- `<` High-priority (not nice to other users)
- `N` Low-priority (nice to other users)
- `L` Has pages locked into memory (for real-time and custom IO)
- `s` Is a session leader (closes all child processes on termination)
- `L` Is multi-threaded (Uses `pthread`)
- `+` Is in the foreground process group



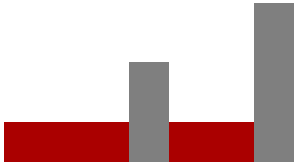
# ps For All


**$ alias**

```
…
alias psall='ps -eH -o ppid,pid,euser,stat,%cpu,rss,args | awk '\''$1!=0'\'' | awk '\''$1!=1'\'' | awk
'\''$1!=2'\'' | more'
…
```

**$ psall**

```
 PPID   PID EUSER    STAT %CPU   RSS COMMAND
 …
4533 21922 root     Ss    0.0  4288     sshd: meadosc [priv]
21922 21936 meadosc  S     0.0  2128        sshd: meadosc@pts/11
21936 21937 meadosc  Ss    0.0  2024          -tcsh
21937 21962 meadosc  S+    0.0  1900            bash
4533 25083 root     Ss    0.4  4284     sshd: brewsteb [priv]
25083 25104 brewsteb S     0.0  2112        sshd: brewsteb@pts/8
25104 25105 brewsteb Ss    0.0  2040          -bash
25105 25761 brewsteb R+    8.0  1852            ps -eH -o ppid,pid,euser,stat,%cpu,rss,args
25105 25762 brewsteb S+    0.0   908            awk $1!=0
25105 25763 brewsteb S+    0.0   908            awk $1!=1
25105 25764 brewsteb S+    0.0   908            awk $1!=2
25105 25765 brewsteb S+    0.0   708            more
 4982  5194 root     Ss    0.0  5136     /opt/dell/srvadmin/sbin/dsm_sa_datamgrd
 5339  5340 root     Sl    0.1 244404     /opt/dell/srvadmin/sbin/dsm_om_connsvcd -run
 5461 25756 root     Zs    0.0     0     [check_nfs.sh] <defunct>
23087 23088 groveed  Ss+   0.0  1784     -bin/tcsh
…
```



# Zombie?

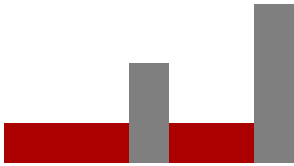
- When a child process terminates, but its parent does not wait for it, the process becomes known as a zombie (aka *defunct*)

# Zombies!?!

- Child processes must report to their parents before their resources will be released by the OS

- If the parents aren't waiting for their children, the processes become the *living undead – forever consuming, forever enslaved to a non-life of waiting and watching.*

- The purpose of a zombie process is to retain the state that `wait()` can retrieve; they *want* to be harvested



# Makin' Zombies

```
$ cat forkyouzombie.c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void main()
{
        pid_t spawnPid = -5;
        int childExitStatus = -5;

        spawnPid = fork();
        switch (spawnPid)
        {
                case -1:
                        perror("Hull Breach!\n");
                        exit(1);
                        break;
                case 0:
                        printf("CHILD: Terminating!\n");
                        break;
                default:
                        printf("PARENT: making child a zombie for ten seconds;\n");
                        printf("PARENT: Type \"ps -elf | grep \'username\'\" to see the defunct child\n");
                        printf("PARENT: Sleeping...\n");
                        fflush(stdout);
                        sleep(10);
                        waitpid(spawnPid, &childExitStatus, 0);
                        break;
        }
        printf("This will be executed by both of us!\n");
        exit(0);
}
```

Make sure all text is outputted before sleeping

# Output - Makin' Zombies

```
$ gcc -o forkyouzombie forkyouzombie.c
$ forkyouzombie
PARENT: making child a zombie for ten seconds;
PARENT: Type "ps -elf | grep 'username'" to see the defunct child
PARENT: Sleeping...
CHILD: Terminating!
This will be executed by both of us!
This will be executed by both of us!
```

Dramatic ten-second pause right here…

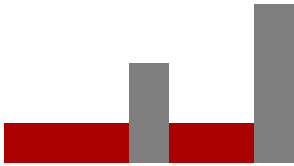
```
// In a second terminal…
$ ps -elf | grep 'brewsteb'
```

Nice, reasonable, short way to do a useful `ps`

```
4 S root     15296  4443  0  80   0 - 32719 unix_s 10:55 ?        00:00:00 sshd: brewsteb [priv]
5 S brewsteb 15298 15296  0  80   0 - 32719 poll_s 10:55 ?        00:00:00 sshd: brewsteb@pts/40
0 S brewsteb 15299 15298  0  80   0 - 30233 wait   10:55 pts/40   00:00:00 -bash
0 S brewsteb 17053 27991  0  80   0 -   981 hrtime 11:15 pts/9    00:00:00 forkyouzombie
1 Z brewsteb 17054 17053  0  80   0 -     0 exit   11:15 pts/9    00:00:00 [forkyouzombie] <defunct>
0 R brewsteb 17057 15299 12  80   0 - 30674 -      11:15 pts/40   00:00:00 ps -elf
0 S brewsteb 17058 15299  0  80   0 - 25829 pipe_w 11:15 pts/40   00:00:00 grep brewsteb
4 S root     27987  4443  0  80   0 - 32719 unix_s 08:51 ?        00:00:00 sshd: brewsteb [priv]
5 S brewsteb 27990 27987  0  80   0 - 32719 poll_s 08:51 ?        00:00:00 sshd: brewsteb@pts/9
0 S brewsteb 27991 27990  0  80   0 - 30234 wait   08:51 pts/9    00:00:00 -bash
```

# How to Deal With Zombies

Zombies stay in the system until they are waited for

# How to deal with Zombies

Zombies stay in the system until they are waited for

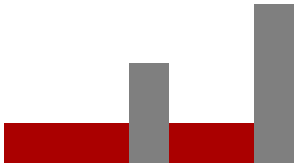

# Orphan Zombies!

- If a parent process terminates *without* cleaning up its zombies, the zombies become orphan zombies
- Orphans are adopted by the `init` process (usually pid = 1) which periodically (in practice, very quickly) waits() for orphans
- Thus eventually, the orphan zombies die

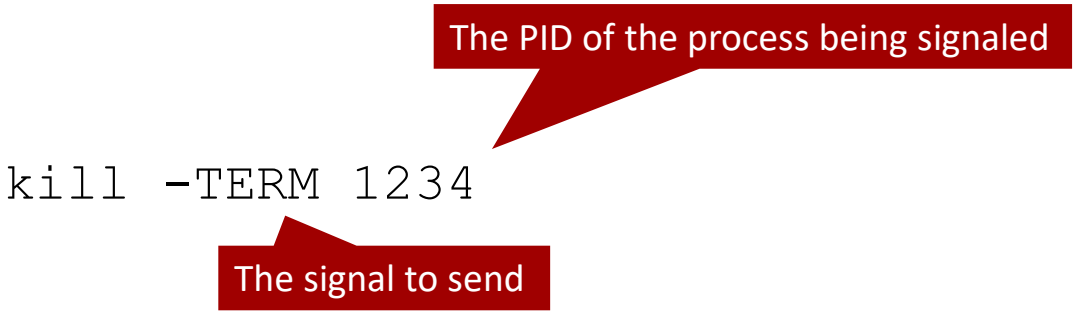
# `kill`

- This UNIX command is used to kill programs
  - another old version is called `kfork`

- "kill" is really a misnomer – it really *just sends signals*



# kill




- The given PID affects who the signal is sent to:
  - If PID > 0, then the signal will be sent to the process PID given
  - If pid == 0, then the signal is sent to all processes in the same process group as the sender (from an interactive command line, this means the foreground process group, i.e. your shell)
  - More trickiness for pid < 0

- We'll discuss more signals later, but you can use the signal `KILL` to tell a process to immediately terminate with no clean-up

# top

- `top` allows you to view the processes running on the machine in real time - one of the few animated built-in programs

```
$ top
top - 14:14:34 up 34 days,  5:15,  9 users,  load average: 0.03, 0.18, 0.22
Tasks: 703 total,   1 running, 697 sleeping,   4 stopped,   1 zombie
Cpu(s):  0.1%us,  0.1%sy,  0.0%ni, 99.8%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  65922540k total,  7876576k used, 58045964k free,   663988k buffers
Swap:  2588668k total,        0k used,  2588668k free,  5258716k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
27609 brewsteb  20   0 27884 1796  996 R  3.4  0.0   0:00.10 top
    1 root      20   0 33656 1624 1292 S  0.0  0.0   1:41.47 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:09.18 kthreadd
    3 root      RT   0     0    0    0 S  0.0  0.0   0:21.23 migration/0
    4 root      20   0     0    0    0 S  0.0  0.0   0:14.47 ksoftirqd/0
    5 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 stopper/0
…
```

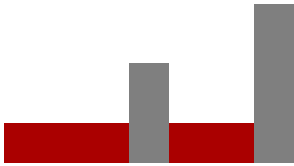
# Diagnosing a Slow CPU

- The *uptime* command shows the average number of runnable processes over several different periods of time (the same info top displays)

```
$ uptime
1:23pm up 25 day(s), 5:59, 72 users, load average: 0.18, 0.19, 0.20
```

- This shows the average number of runnable (the current running process plus the queue of processes waiting to be run) or uninterruptable (waiting for IO) processes over the last 1, 5 and 15 minutes

- If uptime is showing that your runnable queue is consistently *larger than the number of cores*, your CPU is a bottleneck and is causing slow-down



# Diagnosing a Slow CPU - Number of Cores?

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 45
model name      : Intel(R) Xeon(R) CPU E5-2665 0 @ 2.40GHz
stepping        : 7
microcode       : 1808
cpu MHz         : 2399.993
cache size      : 20480 KB
physical id     : 0
siblings        : 16
core id         : 0
cpu cores       : 8
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
…
```

# Diagnosing a Slow CPU - Example - Single Core

```
$ uptime
14:33:04 up 34 days,  5:34, 10 users,  load average: 0.05, 0.15, 0.20
```

This CPU is the champ… or it's not being given anything to do

```
$ uptime
14:33:04 up 34 days,  5:34, 10 users,  load average: 0.88, 1.03, 0.96
```

This CPU is at max - time to upgrade!

```
$ uptime
14:33:04 up 34 days,  5:34, 10 users,  load average: 4.79, 7.23, 6.44
```

It's 3am, and your server is borked; start paging everyone!

# Diagnosing a Slow CPU - Example - Octo Core

```
$ uptime
14:33:04 up 34 days,  5:34, 10 users,  load average: 0.05, 0.15, 0.20
```

Your CPU is bored, and you wasted all the money; but hey, headroom for games!

```
$ uptime
14:33:04 up 34 days,  5:34, 10 users,  load average: 7.99, 8.10, 7.94
```

This CPU is handling processes exactly as fast as it gets them - time to make it more betterer

```
$ uptime
14:33:04 up 34 days,  5:34, 10 users,  load average: 39.90, 41.54, 40.72
```

You're so fired

# Job Control

- How do we start a program, and *still retain access* to the command line for the next program we want to run?
- Can we run multiple processes at once?
- This is called Job Control in UNIX-speak

# Foreground/Background

- There can be only one shell ***foreground*** process – it's the one you're currently interacting with
- If you're at the command prompt, then your foreground program is the shell itself!

- Processes in the ***background*** can still be executing, but they can also be in any number of stopped states:

First State Character:

- `D` Uninterruptible sleep (usually IO)
- `R` Running or runnable (on run queue)
- `S` Interruptible sleep (waiting for an event to complete)
- `T` Stopped, either by a job control signal or because it is being traced
- `Z` Defunct ("zombie") process, terminated but not reaped by its parent

# Foreground/Background in Reality

- There really isn't any difference between processes in these two states; its merely shell nomenclature used to distinguish between them
- When a user enters a command that is intended to run in the foreground (i.e. a normal command), the process started runs to completion *before* the user is prompted again
- When a user enters a command that is intended to run in the background (see later slides), the user is *immediately* prompted again after the process is executed
- In other words, control input to the terminal is not interrupted by a background process

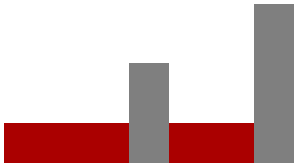
# Start Backgrounded

- Here's how to start a program in the background in the first place:

```
$ ping www.oregonstate.edu &
```

- The ampersand means to start in the background, and must be the last character

- Note that stdout and stderr are still going to the terminal for that process, and stdin might be too if the shell is badly programmed



# Stopping a Process

- Sending the `TSTP` signal stops (not terminates) a process, and puts it into the background
  - Control-z also sends this signal

`ping` is now the foreground process, and I can't enter commands anymore

```
$ ping www.oregonstate.edu
PING www.orst.edu (128.193.4.112) 56(84) bytes of data.
64 bytes from www.orst.edu (128.193.4.112): icmp_seq=1 ttl=250 time=0.362 ms
64 bytes from www.orst.edu (128.193.4.112): icmp_seq=2 ttl=250 time=0.321 ms
64 bytes from www.orst.edu (128.193.4.112): icmp_seq=3 ttl=250 time=0.324 ms
64 bytes from www.orst.edu (128.193.4.112): icmp_seq=4 ttl=250 time=0.328 ms
^Z
[1]+  Stopped                 ping www.oregonstate.edu
$
```

Our shell is once again the foreground process

# jobs

- Use the `jobs` command to see what you're running:

```
$ ping www.oregonstate.edu
PING www.orst.edu (128.193.4.112) 56(84) bytes of data.
64 bytes from www.orst.edu (128.193.4.112): …
64 bytes from www.orst.edu (128.193.4.112): …
^Z
[3]+  Stopped                 ping www.oregonstate.edu
$ jobs -l
[1]   31314 Stopped                 ping www.oregonstate.edu
[2]-  31317 Stopped                 ping www.oregonstate.edu
[3]+  31327 Stopped                 ping www.oregonstate.edu
$ kill -KILL 31327
[3]+  Killed                  ping www.oregonstate.edu
$ kill -KILL %1
[1]-  Killed                  ping www.oregonstate.edu
```

The `-l` switch adds the PID

The – symbol means it was the second-to-last process put in the background

The + symbol means it was the last process put in the background

"Job 1"

# `fg`

- Use the job numbers provided by `jobs` to manipulate processes

- Bring job 1 from the background to the foreground, and start it running again:

  ```
  fg %1
  ```

- Bring most recent backgrounded job to the foreground, and start it running again:

  ```
  fg
  ```

# bg

- Start a specific stopped program that is currently in the background (and keep it in the background):

  ```
  bg %1
  ```

- Start the most recently stopped program in the background (and keep it in the background):

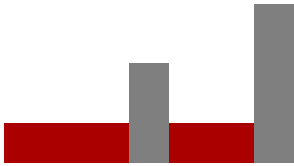
  ```
  bg
  ```

# Who's Got Control of stdout?

- Be advised – background processes can still write to any file including stdout & stderr!

- Jobs demo:

```
1. ping www.oregonstate.edu
2. CTRL-Z
3. jobs
4. fg %1
5. CTRL-Z
6. jobs
7. bg %1
8. ps
9. CTRL-Z CTRL-Z CTRL-Z (doesn't do anything)
10. fg %1
11. CTRL-C
```



# You're Suspended

- Suspend a process that is currently running in the background when you're at the shell

Send stderr and stdout somewhere other than the terminal

Background this command!

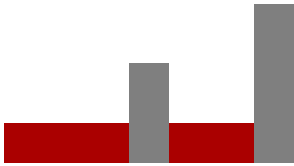
Send the "stop" signal

```
$ ping www.oregonstate.edu 2>/dev/null 1>logfile &
[1] 1660
$ jobs
[1]+  Running                 ping www.oregonstate.edu 2> /dev/null > /dev/null &
$ kill -TSTP %1
[1]+  Stopped                 ping www.oregonstate.edu 2> /dev/null > /dev/null
$ jobs
[1]+  Stopped                 ping www.oregonstate.edu 2> /dev/null > /dev/null
```

# `history` - a Command Visibility Utility

- The history command provides a listing of your previous commands:

```
$ history 5
 1012  jobs
 1013  psme
 1014  top
 1015  jobs
 1016  history 5
```



# Execute a Previous Command

```
$ history 3
 1030  jobs
 1031  psme
 1032  history 3
$ !1030
jobs
$ history 3
 1032  history 3
 1033  jobs
 1034  history 3
$ !-2
jobs
$ !!
jobs
$ history 3
 1034  history 3
 1035  jobs
 1036  history 3
```

Note that no exclamation marks are in the history list - only the actual commands, even when repeated with the ! operator