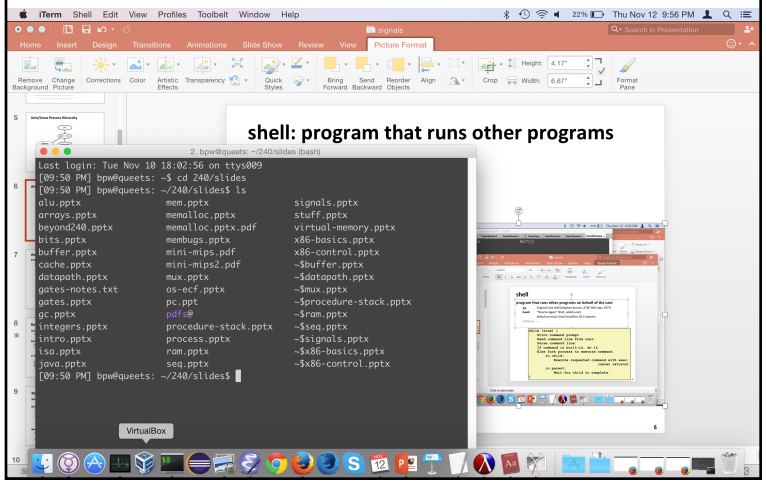
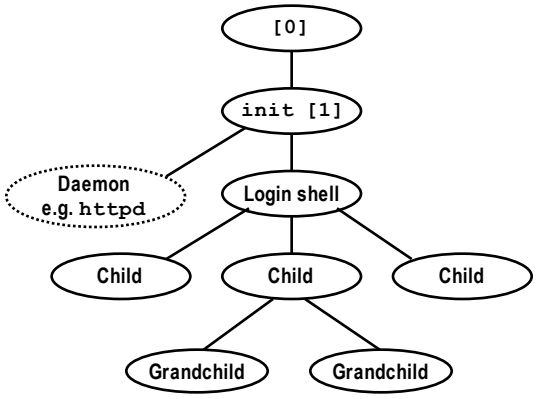
Building a Shell

shell: program that runs other programs



Unix/Linux Process Hierarchy



shell

program that runs other programs on behalf of the user

- **sh** Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
- **bash** "Bourne-Again" Shell, widely used
  default on most Unix/Linux/Mac OSX systems
- others...

```
while (true) {
    Print command prompt.
    Read command line from user.
    Parse command line.
    If command is built-in, do it.
    Else fork process to execute command.
        in child:
            Execute requested command with execv.
                                    (never returns)
        in parent:
            Wait for child to complete.
}
```

terminal ≠ shell

User interface to shell and other programs.

Graphical (GUI) vs. command-line (CLI)

Command-line terminal (emulator):

Input (keyboard)

Output (screen)

8

Background vs. Foreground

Users generally run one command at a time

Type command, read output, type another command

Some programs run "for a long time"

```
$ emacs fizz.txt  # shell stuck until emacs exits.
```

A "background" job is a process we don't want to wait for

```
$ emacs boom.txt &  # emacs runs in background
[1] 9073            # while shell is...
$ gdb ./umbrella    # immediately ready for next command
```

don't do this with emacs unless using X windows version

9

Managing Background Jobs

Shell waits for and reaps foreground jobs.

Background jobs become zombies when they terminate.

Shell might run for a really long time!

Kernel may run out of memory!

fork() returns -1 if per-user process quota exceeded

```
$ ulimit -u            # bash syntax
1024
```

Shell must explicitly reap background jobs.

10

Signals

***Signal*: small message notifying a process of event in system**

like exceptions and interrupts

sent by kernel, sometimes at request of another process

ID is entire message

ID	*Name*	*Corresponding Event*	*Default Action*	*Can Override?*
2	SIGINT	Interrupt (Ctrl-C)	Terminate	Yes
9	SIGKILL	Kill process (immediately)	Terminate	**No**
11	SIGSEGV	Segmentation violation	Terminate & Dump	Yes
14	SIGALRM	Timer signal	Terminate	Yes
15	SIGTERM	Kill process (politely)	Terminate	Yes
17	SIGCHLD	Child stopped or terminated	Ignore	Yes
18	SIGCONT	Continue stopped process	Continue (Resume)	No
19	SIGSTOP	Stop process (immediately)	Stop (Suspend)	**No**
20	SIGTSTP	Stop process (politely)	Stop (Suspend)	Yes

...

12

Sending/Receiving a Signal

Kernel *sends* (delivers) a signal to a *destination process*
by updating state in the context of the destination process.

Reasons:

- **System event**, e.g. segmentation fault (SIGSEGV)
- **Another process used `kill` system call:**
  explicitly request the kernel send a signal to the destination process

Destination process *receives* signal when kernel forces it to react.

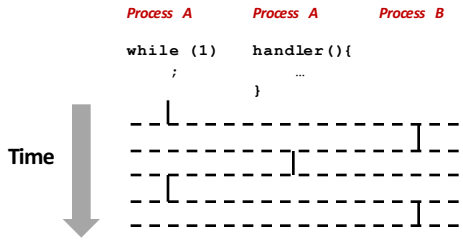
Reactions:

- ***Ignore*** the signal (do nothing)
- ***Terminate*** the process (with optional core dump)
- ***Catch*** the signal by executing a user-level function called *signal handler*
  Like an impoverished Java exception handler

13

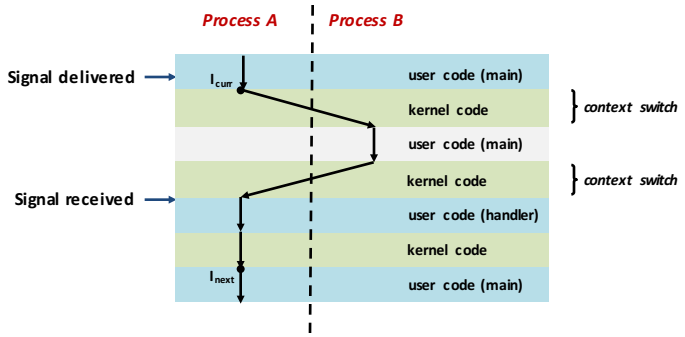
Signals Handlers as Concurrent Flows

Signal handlers run concurrently with main program
(in same process).



15

Another View of Signal Handlers as Concurrent Flows



16

Pending and Blocked Signals

A signal is *pending* if sent but not yet received

- <= 1 pending signal per type per process
- No Queue! Just a bit per signal type.
  - Signals of type S discarded while process has S signal pending.

A process can *block* the receipt of certain signals

- Receipt delayed until the signal is unblocked

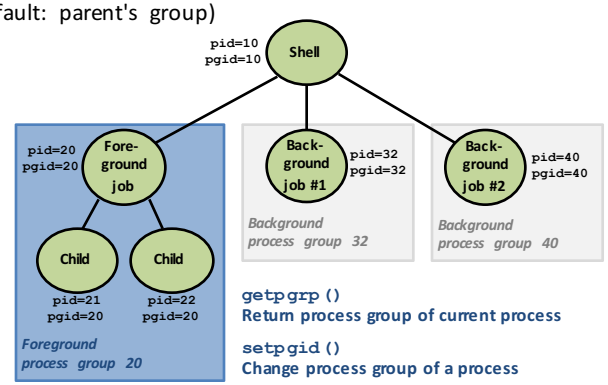
A pending signal is received at most once

Let's draw a picture…

17

Process Groups

Every process belongs to exactly one process group
(default: parent's group)



`getpgrp()`
Return process group of current process

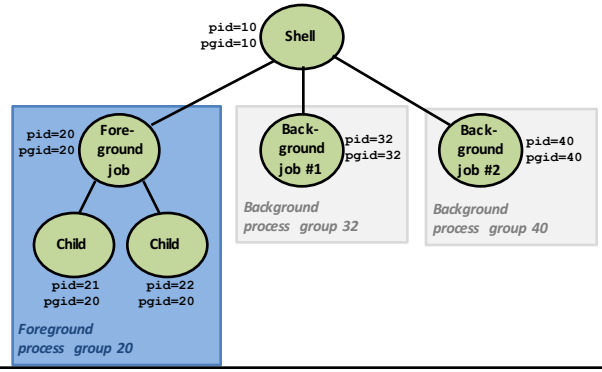
`setpgid()`
Change process group of a process

19

Sending Signals from the Keyboard

Shell: Ctrl-C sends SIGINT (Ctrl-Z sends SIGTSTP) to every job in the foreground process group.

SIGINT – default action is to terminate each process
SIGTSTP – default action is to stop (suspend) each process



20

Signal demos

Ctrl-C

Ctrl-Z

kill

```
kill(pid, SIGINT);
```

21

A Program That Reacts to Externally Generated Events (Ctrl-c)

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void handler(int sig) {
  safe_printf("You think hitting ctrl-c will stop me?\n");
  sleep(2);
  safe_printf("Well...");
  sleep(1);
  printf("OK\n");
  exit(0);
}

main() {
  signal(SIGINT, handler); /* installs ctrl-c handler */
  while(1) {
  }
}
```

```
> ./external
<ctrl-c>
You think hitting ctrl-c will stop me?
Well...OK
>
```

external.c

32

A Program That Reacts to Internally Generated Events

```
#include <stdio.h>
#include <signal.h>

int beeps = 0;

/* SIGALRM handler */
void handler(int sig) {
  safe_printf("BEEP\n");

  if (++beeps < 5)
    alarm(1);
  else {
    safe_printf("DING DING!\n");
    exit(0);
  }
}
```

internal.c

```
main() {
  signal(SIGALRM, handler);
  alarm(1); /* send SIGALRM in
               1 second */

  while (1) {

  }
}
```

```
> ./internal
BEEP
BEEP
BEEP
BEEP
BEEP
DING DING!
>
```

33

Signal summary

Signals provide process-level exception handling

- Can generate from user programs
- Can define effect by declaring signal handler

Some caveats

- Very high overhead
  - >10,000 clock cycles
  - Only use for exceptional conditions
- Not queued
  - Just one bit for each pending signal type
- Many more complicated details we have not discussed.
  - Book goes into too much gory detail.

42