Building a Shell

Shell: program that runs other programs

Unix/Linux Process Hierarchy

Shell

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)

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

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.

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

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

Signals Handlers as Concurrent Flows

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

```
Process A while (1) handler(){
    ...}
Process B
```

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...
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

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

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

```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...\n");
    sleep(1);
    printf("OK\n");
    exit(0);
}

main() {
    signal(SIGINT, handler);
    while(1) {
        
    }
}
```

Signal demos

- Ctrl-C
- Ctrl-Z
- `kill`
- `kill(pid, SIGINT);`
A Program That Reacts to Internally Generated Events

```c
#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);
    }
}

int main() {
    signal(SIGALRM, handler);
    alarm(1); /* send SIGALRM in 1 second */
    while (1) {
    }
    > /internal
    BEEP
    BEEP
    BEEP
    BEEP
    DING DING!
>
```

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.