Shells and Signals

Shells and the process hierarchy

Login shell

init [1]

[0]

Daemon
e.g. httpd

Child

Grandchild

Shell logic

program that runs other programs on behalf of the user

```
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:
        Exec requested command (never returns)
    in parent:
        Wait for child to complete.
}
```

sh
Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
bash
“Bourne-Again” Shell, widely used
default on most Unix/Linux/Mac OS X systems
many others...
**Terminal ≠ shell**

User interface to shell and other programs.
- Graphical (GUI) vs. command-line (CLI)

**Command-line terminal (emulator):**
- Input (keyboard)
- Output (screen, sound)

---

**To wait or not?**

A **foreground** job is a process for which the shell waits.*

```bash
$ emacs fizz.txt  # shell waits until emacs exits.
```

A **background** job is a process for which the shell does not wait*... yet.

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

*Also: foreground jobs get input from (and "own") the terminal. Background jobs do not.

---

**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

---

*Optional*
Signals handlers as concurrent flows

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

```
Process A
while (1) 
  handler();
```

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.

Signal demos

Ctrl-C

Ctrl-Z

kill

kill(pid, SIGINT);

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\n");
    sleep(2);
    safe_printf("Well...\n");
    sleep(1);
    printf("OK\n");
    exit(0);
}

tmain() {
    signal(SIGINT, handler);
    while(1) {
        safe_printf("ctrl-c\n");
        You think hitting ctrl-c will stop me?
        Well...OK
    }
}
```

A program that reacts to internally generated events

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

int beeps = 0;

/main() {
    signal(SIGALRM, handler);
    alarm(1); /* send SIGALRM in 1 second */
    while (1) {
        safe_printf("BEEP\n");
        if (++beeps < 5)
            alarm(1);
        else {
            safe_printf("DING DING!\n");
            exit(0);
        }
    }
}
```

```c
void handler(int sig) {
    safe_printf("BEEP\n");
    if (++beeps < 5)
        alarm(1);
    else {
        safe_printf("DING DING!\n");
        exit(0);
    }
}

InBackground job #1

pid=32
pgid=32

InBackground job #2

pid=40
pgid=40

Foreground process group 20

Background process group 32

Background process group 40

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child

Child
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.