Shells and Signals

shell: program that runs other programs

How many child threads are there at this point for this shell?

1 (foreground)
2 (1 foreground, 1 background)
3 (2 foreground, 1 background)
3 (1 foreground, 2 background)
None of the above
### Shells and the process hierarchy

![Process Hierarchy Diagram]

#### Shell summary

**Program that runs other programs on behalf of the user**

Typically via the “command line interface” (CLI)

**Example shells**

- **sh**: Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
- **bash**: “Bourne-Again” Shell, widely used, default on most Unix/Linux systems
- **zsh**: Pronounced “z shell”, newer, now default on newer Mac systems
- **Terminal**: Default on Windows systems
- **many others…**

### Shell implementation (Concurrency assignment)

**Shell high-level design:**

1. Wait for input from the user. Print the “command prompt” to indicate readiness.
2. Read in a command from the user, parse it (Pointers assignment)
3. Execute the command, either by:
   1. If a built-in command, do it.
   2. Otherwise, create a child process to run the command (fork call)

**Pseudocode:**

```
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.
```

- **cd is built-in**
- **echo is not built-in**
Terminal ≠ shell

**Terminal** is the user interface to shell and other programs.

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

The shell itself does not control pixels, it manipulates strings.

---

To wait or not to wait?

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

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

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

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

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
Signals handlers as concurrent flows

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

```
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)

```bash
getpgrp()  Return process group of current process
setpgid()  Change process group of a process
```

optional
Sending signals from the keyboard

Shell: Ctrl-C sends SIGINT (Ctrl-Z sends SIGTSTP)

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

main() {
    signal(SIGINT, handler); /* installs ctrl-c handler */  
    signal(SIGALRM, handler); /* installs alarm 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);  
        }  
        }  
    exit(0);  
}
```

external.c

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);  
    }  
}

main() {  
    signal(SIGALRM, handler); /* sends SIGALRM in 1 second */  
    alarm(1);  
    while (1) {  
        }  
    exit(0);  
}
```

internal.c

Signal demos

Ctrl-C

Ctrl-Z

kill

kill(pid, SIGINT);
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.

Conclusion of unit: Hardware-Software Interface (ISA)

Lectures
- Programming with Memory
- x86 Basics
- x86 Control Flow
- x86 Procedures, Call Stack
- Representing Data Structures
- Buffer Overflows
- Processes Model
- Shells

Labs
- 6: Pointers in C
- 7: x86 Assembly
- 8: x86 Stack
- 9: Data structures in memory
- 10: Buffer overflows
- 11: Processes

Topics
- C programming: pointers, dereferencing, arrays, structs, cursor-style programming, using malloc
- x86: instruction set architecture, machine code, assembly language, reading/writing x86, basic program translation
- Procedures and the call stack, data layout, security implications
- Processes, shell, fork, wait

Assignments
- Pointers
- x86
- Buffer
- Concurrency

Exam 2: ISA + Process/Shell
April 18
(1 week from today)