CS 240
Foundations of Computer Systems

WELLESLEY

Shells and Signals

Big picture: how do programs run other programs? How does a terminal work?

https://cs.wellesley.edu/~cs240/

shell: program that runs other programs

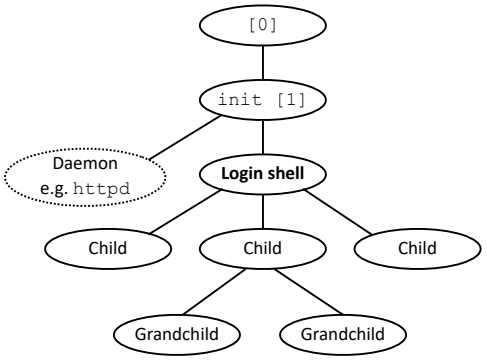
```
avh — -zsh — 80×24
[avh ~ $ pwd
/Users/avh
[avh ~ $ echo "hello cs240"
hello cs240
[avh ~ $ sleep 1
[avh ~ $ sleep 1; echo "hello"
hello
[avh ~ $ (sleep 4; echo "hello")&
[1] 29371
avh ~ $ hello

[1]  + done       ( sleep 4; echo "hello"; )
[avh ~ $ (sleep 5; echo "hello")&
[1] 29577
[avh ~ $ (sleep 10; echo "hello")&
[2] 29648
avh ~ $ hello

[1]  - done       ( sleep 5; echo "hello"; )
avh ~ $ hello

[2]  + done       ( sleep 10; echo "hello"; )
avh ~ $
```


Shells and the process hierarchy



Shell summary

Program that runs other programs on behalf of the user

Typically via the “command line interface” (CLI)

Example: Mac (zsh)

```
avh — -zsh — 32×5
[avh ~ $ pwd
/Users/avh
[avh ~ $ echo $SHELL
/bin/zsh
avh ~ $
```

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
Windows Terminal	Default on Windows systems

many others…

Example: CSLinux (bash)

```
avh — avh@cs:~ — ssh avh@cs.wellesley.edu —...
avh ~ $ ssh avh@cs.wellesley.edu
Last login: Fri Mar  8 22:32:21 2024 from 73.17.106.151
[avh@cs ~] pwd
/home/avh
[avh@cs ~] echo $SHELL
/bin/bash
[avh@cs ~]
```

Shell implementation (Concurrency assignment)

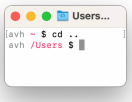
Shell high-level design:

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)

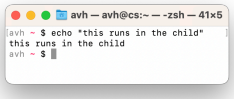
Pseudocode:

```
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



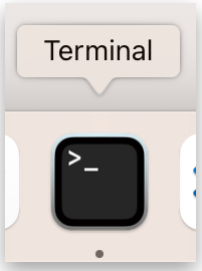
echo is not built-in



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.*

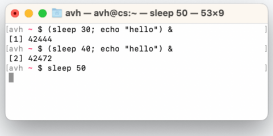
```
$ emacs fizz.txt  # shell waits until emacs exits.
```

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.

How many child processes 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

Concurrency assignment feature: function pointers

```c
void print_each(int *arr, int len) {
  for (int i = 0; i < len; i++) {
    printf("%d\n", arr[i]);
  }
}

void print_squares(int *arr, int len) {
  for (int i = 0; i < len; i++) {
    int v = arr[i];
    printf("%d squared is %d\n", v, v*v);
  }
}
```

Sometimes, we end up writing code that repeats similar patterns for different specific behavior

If we wanted to add dozens more similar functions here (calculator app), how could we use abstraction?

Concurrency assignment feature: function pointers

```c
void do_calculation(int *a, int l, void (*f)(int)) {
    for (int i = 0; i < l; i++) {
        f(a[i]);
    }
}

void print_elem(int x) {
    printf("%d\n", x);
}

void print_square(int x) {
    printf("%d squared is %d\n", x, x*x);
}

// to use
do_calculation(int_arr, 6, print_elem);
do_calculation(int_arr, 6, print_square);
```

Solution: add an additional argument and pass a *function pointer* to specify the specific computation

```
<return type> (*<name>)(<arg types>))
```

Can pass this a function with type signature:

```
<return type> <actual name>(<type> arg, …){
   // … body
}
```



```c
/* In file joblist.h */
// Job was running in foreground last we knew
#define JOB_STATUS_FOREGROUND 1
// Job was stopped last we knew
#define JOB_STATUS_STOPPED    2
// Job was running in background last we knew
#define JOB_STATUS_BACKGROUND 3


typedef struct Job {
  /* Is the process stopped or running? */
  JobStatus status;
  // More field elided ...
  /* Link to next node in list of jobs. */
  struct Job* next;
  struct Job* prev;
} Job;

/* Make a state transition. */
void job_set_status(JobList* jobs, Job* j, JobStatus s);

/* Call function fp once on each job in jobs.  */
void job_iter(JobList* jobs, void (*fp)(JobList*,Job*));

/* Call function fp once on each job in jobs.  */
void job_iter(JobList* jobs, void (*fp)(JobList*,Job*));
```

Exercise:

ex

On the concurrency assignment: we provide a linked list of jobs, with helper functions to iterate over the list

1. Write a function you can pass to `job_iter` that sets a job's status to "stopped
2. Write a line of code using `job_iter` that calls your function from part (1). Assume you have a jobs list in variables `jobs`

Exercise solution:

ex

```c
#include "joblist.h"

/*
Part (1) example: use jobs_iter to set the status of each
job in the job list to stopped */
void stop(JobList *l, Job *j) {
    job_set_status(l, j, JOB_STATUS_STOPPED);
    // Or j->status = JOB_STATUS_STOPPED;
}

void main() {
    JobList* jobs = joblist_create();
    //   Elided: adding jobs...

    // Part (2): call job_iter, passing the function name
    // as an argument
    job_iter(jobs, stop);
}
```

1. Write a function you can pass to `job_iter` that sets a job's status to "stopped
2. Write a line of code using `job_iter` that calls your function from part (1). Assume you have a jobs list in variables `jobs`

Signals

optional

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

...

Sending/receiving a signal

optional

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)

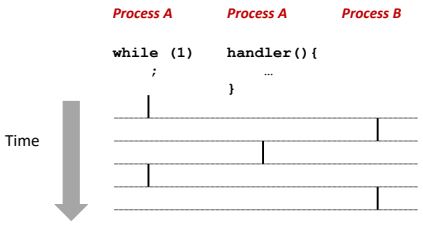
Catch the signal by executing a user-level function called *signal handler*

Like an impoverished Java exception handler

Signals handlers as concurrent flows

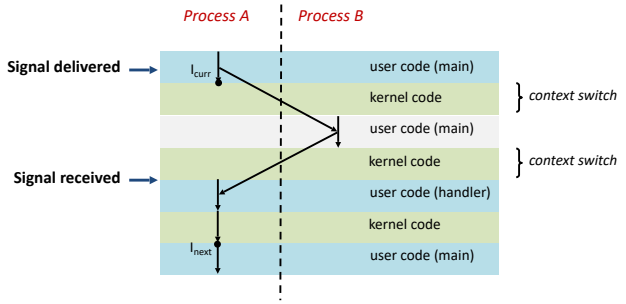
optional

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



Another view of signal handlers as concurrent flows

optional



Pending and blocked signals

optional

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

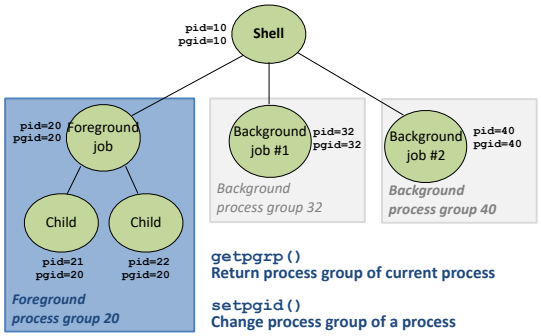
A pending signal is received at most once

Let's draw a picture...

Process Groups

optional

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



`getpgrp()`
Return process group of current process

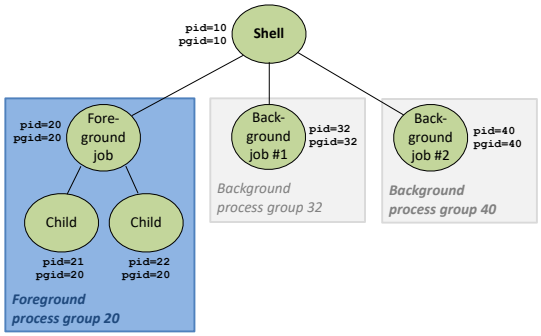
`setpgid()`
Change process group of a process

Sending signals from the keyboard

optional

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

optional

Ctrl-C

Ctrl-Z

kill

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

A program that reacts to externally generated events (Ctrl-c)

optional

```
#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

A program that reacts to internally generated events

optional

```
#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!
>
```


Signal summary

optional

- 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.