18-600: Recitation #7

Oct 10th, 2017

Shell Lab

Today

- **Shell Lab**
  - Exceptional control flow
  - Processes
  - Signals
  - The shell

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor's interrupt pin
  - Handler returns to "next" instruction
- Examples:
  - I/O interrupts
    - hitting Ctrl-C at the keyboard
    - arrival of a packet from a network
    - arrival of data from a disk
  - Hard reset interrupt
    - hitting the reset button
  - Soft reset interrupt
    - hitting Ctrl-Alt-Delete on a PC

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - *Traps*
    - Intentional
    - Examples: ***system calls***, breakpoint traps, special instructions
    - Returns control to "next" instruction
  - *Faults*
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting ("current") instruction or aborts
  - *Aborts*
    - unintentional and unrecoverable
    - Examples: parity error, machine check
    - Aborts current program

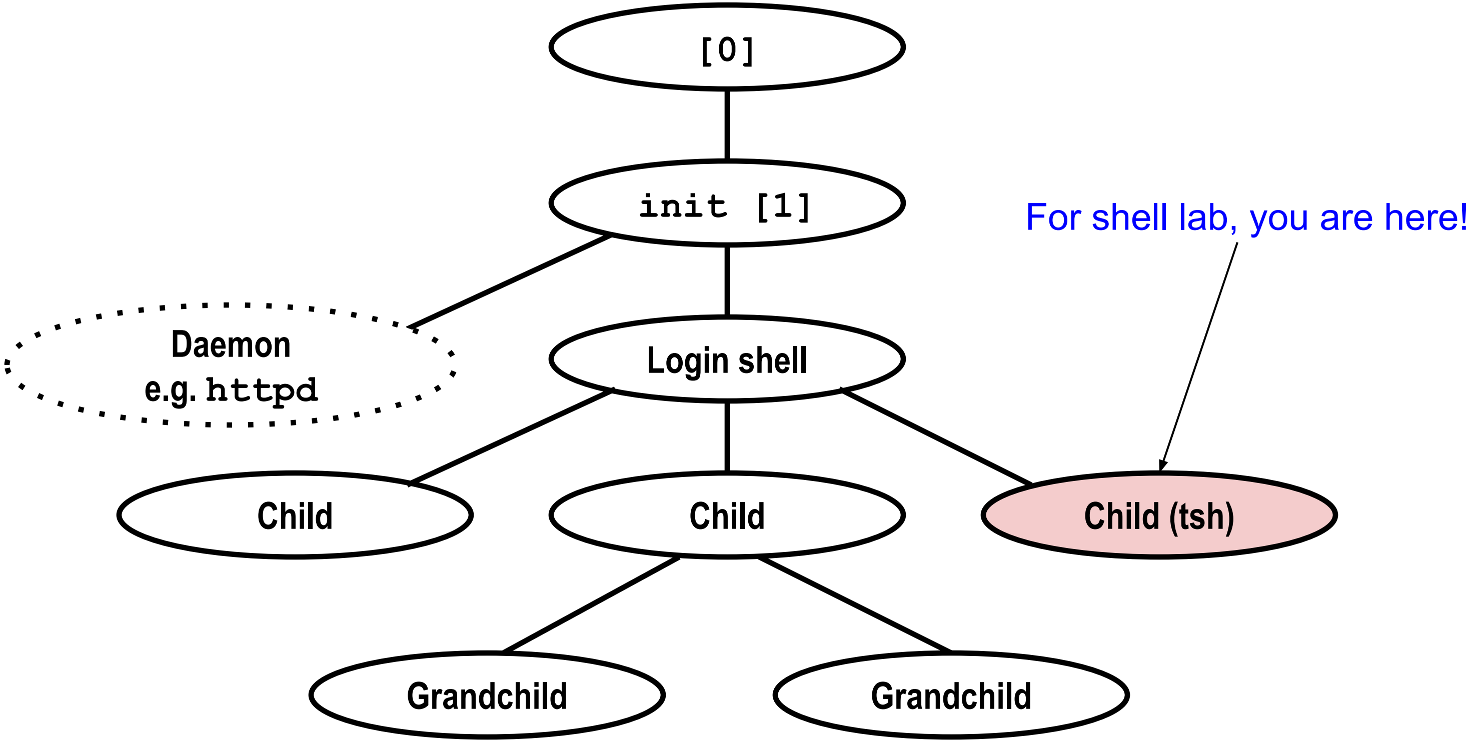
Processes

- What is a *program*?
  - A bunch of data and instructions stored in an executable binary file
- What is a *process?*
  - An instance of a running program
- Process provides each program with two key abstractions:
  - Logical control flow
    - Each program seems to have exclusive use of the CPU
  - Private virtual address space
    - Each program seems to have exclusive use of main memory
    - Gives the running program a ***state***

Processes

- Four basic States
  - Running
    - Executing instructions on the CPU
    - Number bounded by number of CPU cores
  - Runnable
    - Waiting to be run
  - Blocked
    - Waiting for an event, maybe input from STDIN
    - Not runnable
  - Zombie
    - Terminated, not yet reaped

Unix Process Hierarchy



Processes

- Four basic process control function families:
  - fork()
  - exec()
    - And other variants such as execve()
  - exit()
  - wait()
    - And variants like waitpid()
- Standard on all UNIX-based systems
- Don’t be confused:
  Fork(), **E**xit(), **W**ait() are all wrappers provided by CS:APP

Process Examples

```
pid_t child_pid = fork();

if (child_pid == 0){
   /* only child comes here */

   printf(“Child!\n”);

   exit(0);
}
else{

   printf(“Parent!\n”);
}
```

- What are the possible output (assuming fork succeeds) ?
  - `Child!`
    `Parent!`
  - `Parent!`
    `Child!`

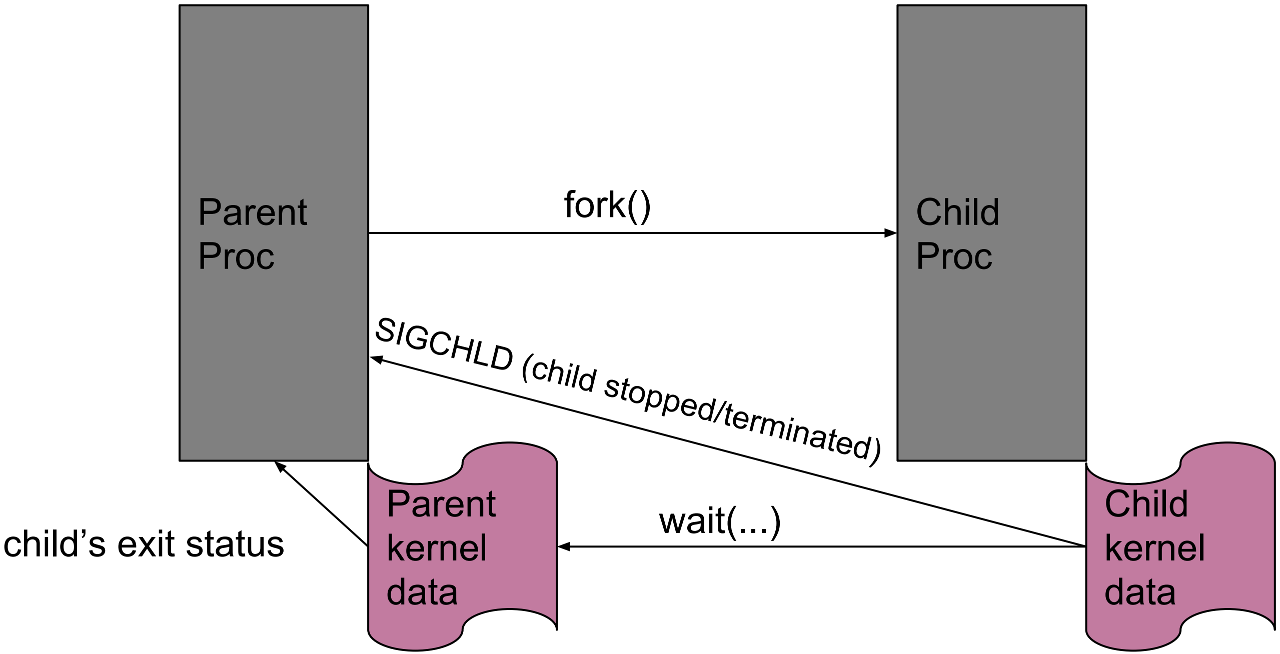
- How to get the child to always print first?

Process Examples

```
int status;
pid_t child_pid = fork();

if (child_pid == 0){
   /* only child comes here */

   printf("Child!\n");

   exit(0);
}
else{
   waitpid(child_pid, &status, 0);

   printf("Parent!\n");
}
```

- Waits till the child has terminated. Parent can inspect exit status of child using 'status'
  - WEXITSTATUS(status)
- Output always:
  Child!
  Parent!

Reaping


Parent
Proc
fork()
Child
Proc
SIGCHLD (child stopped/terminated)
Parent
kernel
data
wait(...)
Child
kernel
data
child's exit status

Process Examples

```
int status;
pid_t child_pid = fork();
char* argv[] = {“/bin/ls”, “-l”, NULL};
char* env[] = {…, NULL};

if (child_pid == 0){
   /* only child comes here */

   execve(“/bin/ls”, argv, env);

   /* will child reach here? */
}
else{
   waitpid(child_pid, &status, 0);

   … parent continues execution…
}
```

- An example of something useful.
- Why is the first arg “/bin/ls”?
- Will child reach here?

How do we get the process tree?

- The operating system launches the **init** process
- **init** then spawns all the other processes (e.g. shell)
- This is done via calls to fork() and exec()
  - fork - create a “duplicate” process, with its own memory and states
  - exec - Hijack the current process’ memory and load an entirely new program
- This keeps repeating

Signals

- A *signal* is a small message that notifies a process that an event of some type has occurred in the system
  - akin to exceptions and interrupts (asynchronous)
  - sent from the kernel (sometimes at the request of another process) to a process
  - signal type is identified by small integer ID’s (1 - 32+)
  - only information in a signal is its **ID** and the fact that it arrived

ID	*Name*	*Default Action*	*Corresponding Event*
2	SIGINT	Terminate	Interrupt (e.g., ctrl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation(Segfault)
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Signals

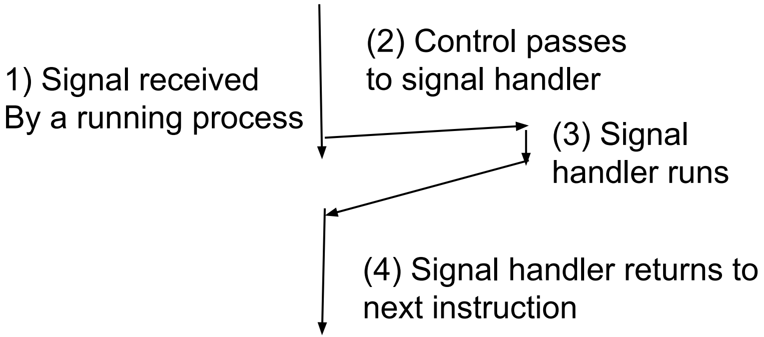
- A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal
- Blocking signals
  - Sometimes code needs to run through a section that can't be interrupted
  - Implemented with `sigprocmask()`
- Waiting for signals
  - Sometimes, we want to pause execution until we get a specific signal
  - Implemented with `sigsuspend()`
- Can't modify behavior of SIGKILL and SIGSTOP
- Think about what could happen when another signal is received within a signal handler!

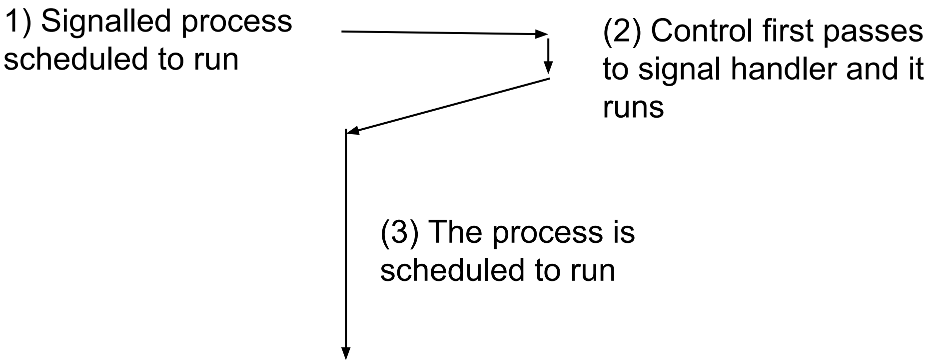
Signal Handling

- Process software
  - Send a signal e.g. via kill(...) [Yes, kill doesn't necessarily kill a process]
- Hardware
  - Raises an exception e.g. segmentation fault
- Kernel coordinates the signal delivery
- The process for which the signal is intended can choose to either receive or ignore the signal (Except SIGKILL and SIGSTOP)
- Standard POSIX signals are not queued (existing pending signal will be overwritten)!

Signal Handling Contd.

- Running Process:
  - Receipt of a signal triggers a control transfer to a signal handler
  - After it finishes processing, the handler returns control to the interrupted program
- Runnable Process:
  - When the process is next scheduled, the control is first transferred to the signal handler
  - After it finishes processing, the handler returns control to the program





Signal Handlers

- Signal handlers
  - Can be installed to run when a signal is received
  - The form is `void handler(int signum){ … }`
  - **Separate** flow of control in the same process
  - Resumes normal flow of control upon returning
  - Can be called **anytime** when the appropriate signal is fired
  - CSAPP provides a Signal(...) API to register handlers, but this is sigaction(...) under the hood! Look up why sigaction(...) has replaced signal(...).

Shell Lab

- Shell Lab will be out on October 12th!
- Read the code we've given you
  - There's a lot of stuff you don't need to write yourself; we gave you quite a few helper functions
  - It's a good example of the code we expect from you!
- Don't be afraid to write your own helper functions; you might find yourself needing them!
- Watch out for all interleaved scenarios to account for race conditions!

Shell Lab

- Read man pages. You may find the following functions helpful:
  - `sigemptyset()`
  - `sigaddset()`
  - `sigprocmask()`
  - `sigsuspend()`
  - `waitpid()`
  - `open()`
  - `dup2()`
  - `setpgid()`
  - `kill()`
- Please read the man pages thoroughly to understand what each function does
- Please **do not** use sleep() to solve synchronization issues.

Shell Lab

- Hazards
  - Race conditions
    - Hard to debug so start early (and think carefully)
  - Reaping zombies
    - Race conditions
    - Handling signals correctly
  - Waiting for foreground job
    - Think carefully about what the right way to do this is

Things to remember while working on Shell Lab

- Run your shell
  - This is the fun part!
- Utilize tshref
  - How should the shell behave?
- Run traces
  - Each trace tests one feature.
- Breathe

Questions?