Recitation 8: Signals & Shells

Andrew Faulring
15213 Section A
28 October 2002

Today’s Plan

• Process IDs & Process Groups
• Process Control
• Signals
• Preemptive Scheduler
  – Race hazards
• Reaping Child Processes

Andrew Faulring

• faulring@cs.cmu.edu
• Office hours:
  – NSH 2504 (lab) / 2507 (conference room)
  – Thursday 5–6

• Lab 5
  – due Thursday, 31 Oct @ 11:59pm
    • Halloween Night ... happy reaping!

Lab 5: Shell

• tshref
  – Use as a guide for output
  – You shell should have same behavior
How Programmers Play with Processes

- Process: executing copy of program
- Basic functions
  - `fork()` spawns new process
  - `exit()` terminates calling process
  - `wait()` and `waitpid()` wait for and reap terminated children
  - `execl()` and `execve()` run a new program in an existing process

Process IDs & Process Groups

- Each process has its own, unique process ID
  - `pid_t getpid();`
- Each process belongs to exactly one process group
  - `pid_t getpgid();`
- To which process group does a new process initially belong?
  - Its parent's process group
- A process can make a process group for itself and its children
  - `setpgid(0, 0);`

Signals

- Section 8.5 in text
  - Read at least twice … really!
- A signal tells our program that some event has occurred
  - For instance, a child process has terminated
- Can we use signals to count events?
  - No
Important Signals

- **SIGINT**
  - Interrupt signal from keyboard (ctrl-c)
- **SIGTSTP**
  - Stop signal from keyboard (ctrl-z)
- **SIGCHLD**
  - A child process has stopped or terminated

Look at Figure 8.23 for a complete list of Linux signals

Sending a Signal

- **Send a signal**
  - Sent by either the kernel
  - Or another process
- **Why is a signal sent?**
  - The kernel detects a system event.
    - Divide-by-zero (SIGFPE)
    - Termination of a child process (SIGCHLD)
  - Another process invokes a system call.
    - `kill(pid_t pid, int SIGINT)`
      - `kill(1500, SIGINT)` → Send SIGINT to process 1500
      - `kill(-1500, SIGINT)` → Send SIGINT to process group 1500
    - `alarm(unsigned int secs)`

Receiving a Signal

- **Default action**
  - The process terminates [and dumps core]
  - The process stops until restarted by a SIGCONT signal
  - The process ignore the signal
- **Can modify the default action with the signal function**
  - Additional action: “Handle the signal”
    - `void sigint_handler(int sig);`
    - `signal(SIGINT, sigint_handler);`
  - Cannot modify action for SIGSTOP and SIGKILL

Receiving a Signal

- **pending**: bit vector: bit $k$ is set when signal type $k$ is delivered, clear when signal received
- **blocked**: bit vector of signals that should not be received
- Only receive non-blocked, pending signals
  - `pending & ~blocked`
Synchronizing Processes

- Preemptive scheduler run multiple programs “concurrently” by time slicing
  - How does time slicing work?
  - The scheduler can stop a program at any point
  - Signal handler code can run at any point, too
- Program behaviors depend on how the scheduler interleaves the execution of processes
- Racing condition between parent and child!
  - Why?

Race Hazard

- Different behaviors of program depending upon how the schedule interleaves the execution of code.

Parent & Child Race Hazard

```c
sigchld_handler() { 
    pid = waitpid(…); 
    deletejob(pid); 
}

eval() { 
    pid = fork(); 
    if(pid == 0) 
        {/* child */
            execve(…); 
        } /* parent */
    /* signal handler might run BEFORE addjob() */
    addjob(…); 
}
```

An Okay Schedule

```

<table>
<thead>
<tr>
<th>time</th>
<th>Shell</th>
<th>Signal Handler</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fork()</td>
<td></td>
<td>execve()</td>
</tr>
<tr>
<td></td>
<td>addjob()</td>
<td></td>
<td>exit()</td>
</tr>
<tr>
<td></td>
<td>sigchld_handler()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>deletejobs()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
A Problematic Schedule

Shell Signal Handler Child

```
time
  fork()
  sigchld_handler()
  execve()
  deletejobs()
  exit()
  addjob()
```

Job added to job list *after* the signal handler tried to delete it!

Solution to Race Hazard

```
sigchld_handler() {
  pid = waitpid(…);
  deletejob(pid);
}
sigprocmask(SIG_BLOCK, …)
fork();
if(pid == 0) {
  /* child */
  sigprocmask(SIG_UNBLOCK, …)
  execve(…);
}
/* parent */
/* signal handler might run BEFORE addjob() */
addjob(…); sigprocmask(SIG_UNBLOCK, …)
```

More details 8.5.6 (page 633)

Reaping Child Process

- Child process becomes zombie when terminates
  - Still consume system resources
  - Parent performs reaping on terminated child
    - Using either `wait` or `waitpid` syscall
- Where to wait children processes to terminate?
  - Two waits
    - `sigchld_handler`
    - `eval` for foreground processes
  - One wait
    - `sigchld_handler`
    - But what about foreground processes?

Busy Wait

```
void eval() {
  …
  /* parent */
  addjob(…);
  while (fg process still alive) {
  ;
  }
}
sigchld_handler() {
  pid = waitpid(…);
  deletejob(pid);
}
```


Pause

```c
void eval() {
    /* parent */
    addjob(...);
    while(fg process still alive) {
        pause();
    }
}

sigchld_handler() {
    pid = waitpid(...);
    deletejob(pid);
}
```

Sleep

```c
void eval() {
    /* parent */
    addjob(...);
    while(fg process still alive) {
        sleep(1);
    }
}

sigchld_handler() {
    pid = waitpid(...);
    deletejob(pid);
}
```

waitpid

- Used for reaping zombied child processes
- `pid_t waitpid(pid_t pid, int *status, int options)`
  - `pid`: wait until child process with pid has terminated
  - `-1`: wait for any child process
  - `status`: tells why child terminated
  - `options`:
    - `WNOHANG`: return immediately if no children have exited (zombied)
    - `waitpid` returns `-1`
    - `WUNTRACED`: report status of stopped children too

waitpid’s status

- `int status;`, `waitpid(pid, &status, NULL)`
- `WIFEXITED(status)`: child exited normally
  - `WEXITSTATUS(status)`: return code when child exits
- `WIFSIGNALED(status)`: child exited because a signal was not caught
  - `WTERMSIG(status)`: gives the number of the terminating signal
- `WIFSTOPPED(status)`: child is stopped
  - `WSTOPSIG(status)`: gives the number of the stop signal
Summary

• Process provides applications with the illusions of:
  – Exclusively use of the processor and the main memory
• At the interface with OS, applications can:
  – Creating child processes
  – Run new programs
  – Catch signals from other processes
• Use man if anything is not clear!