15213 Recitation Section C
Shimin Chen
Oct. 28, 2002

Outline
• Process
• Signals
• Reaping Child Processes
• Race Hazard

Process Concept
• An instance of running program
• Multiple processes run “concurrently” by time slicing
  – What is time slicing?
  – Preemptive scheduler of OS: it can stop a program at any point!

Process IDs & Process Groups
• A process has its own, unique process ID
  – `pid_t getpid();`
• A process belongs to exactly one process group
  – `pid_t getpgrp();`
• A new process belongs to which process group?
  – Its parent’s process group
• A process can make a process group for itself and its children
  – `pid_t pid = getpid();`
  – `setpgid(0, 0);`
  – `getpgrp() -> -pid`

Process Tree for Shell
Signals

- Section 8.5 in text
  - Read at least twice … really!
- A signal tells our program that some event has occurred
- Can we use signals to count events?
  - No

Important Signals (Fig 8.23)

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

Signals: sending

Process 1
kill(pid, SIGINT)

Process 2
OS procedure
other events
OS Kernel

• divide by zero: SIGFPE
• ctrl-c: SIGINT
• child process exit: SIGCHLD

Blocked pending

Signals: receiving

Check when schedule the process to run

Process 2
OS procedure
OS Kernel

Blocked pending
Receiving a Signal

- Default action
  - The process terminates [and dumps core]
  - The process stops until restarted by a SIGCONT signal
  - The process ignores the signal
- Can modify (additional action)
  - “Handle the signal”
    - `void sigint_handler(int sig);
    - signal(SIGINT, sigint_handler);`

Reaping Child Process

- Child process becomes zombie when terminates
  - Still consume system resources
  - Parent performs reaping on terminated child
    - `wait()` `waitpid()`
- Straightforward for reaping a single child
- Tricky for Shell implementation!
  - multiple child processes
  - both foreground and background

Reaping Child Process

- Two waits
  - `sigchld_handler`
  - `eval`: for foreground processes
- One wait
  - `sigchld_handler`
  - But what about foreground processes?

Busy Wait

```c
if(fork() != 0) { /* parent */
    addjob(...);
    while(fg process still alive){
        /* do nothing */
    }
}
```
Pause

```c
if (fork() != 0) { /* parent */
    addjob(...);
    while (fg process still alive){
        pause();
    }
}
```

If signal handled before call to pause, then pause will not return when foreground process sends SIGCHLD

Sleep

```c
if (fork() != 0) { /* parent */
    addjob(...);
    while (fg process still alive){
        sleep(1);
    }
}
```

waitpid()

```c
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 zombied
    - returns -1
  - `WUNTRACED`: report status of stopped children too

- `wait (&status)` equivalent to
  - `waitpid (-1, &status, 0)`

Status in Waitpid

- `int status;
  waitpid(pid, &status, NULL)`
- Macros to evaluate status:
  ```c
  { - WIFEXITED(status): child exited normally
    - WEXITSTATUS(status): return code when child exits
    - WIFSIGNALED(status): child exited because of a signal not caught
    - WTERMSIG(status): gives the terminating signal number
    - WIFSTOPPED(status): child is currently stopped
    - WSTOPSIG(status): gives the stop signal number
  ```
Man page

- Check man page for details of a system call:
  - man waitpid

Race Hazard

- A data structure is shared by two pieces of code that can run concurrently
- Different behaviors of program depending upon how the schedule interleaves the execution of code.

eval & sigchld_handler 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

```
time Shell                      Signal Handler                      Child
--------------- ------------------------------- ---------------
| fork()         | addjob()                           | exit()          |
| addjob(...)    |                                | sigchld_handler()
|                | deletejobs()                      |
```

Shimin Chen
A Problematic Schedule

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

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

Blocking Signals

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

eval() {
    sigprocmask(SIG_BLOCK, ...)
    pid = 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)

Summary

- Process
- Signals
- Reaping Child Processes
- Race Hazard

- Check man page to understand the system calls better
  - man waitpid