Lecture 22: Multitasking and Signals

Some slides adapted (and slightly modified) from:
• Clark Barrett
• Jinyang Li
• Randy Bryant
• Dave O’Hallaron
Multitasking

• OS runs many processes “concurrently”
  – Process: a running program

• Context switches from one process to another
  – Suspend process when it needs to do I/O or timer expires
  – Resume process when I/O available or given scheduling opportunity

• Appears as if all processes executing simultaneously
What really happens

What we think it happens!
OS’ process abstraction

• Basic syscalls for managing processes:
  – **fork** spawns new process
    • Called once, returns twice
  – **exit** terminates own process
    • Puts it into “zombie” status until its parent reaps
  – **wait** and **waitpid** wait for and reap terminated children
  – **execve** runs new program in existing process
    • Called once, never returns
Unix Process Hierarchy

- init [1]
  - Login shell
    - Daemon
      - e.g. httpd
    - Child
      - Child
        - Child
          - Grandchild
          - Grandchild
Shell Programs

- **A shell** is a **user-level** program that runs programs on behalf of the user.
  - **sh**  Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
  - **csh**  BSD Unix C shell (tcsh: enhanced csh)
  - **bash**  “Bourne-Again” Shell

```c
int main() {
    char cmdline[MAXLINE];

    while (1) {
        /* read */
        printf("> ");
        fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* evaluate */
        eval(cmdline);
    }
}
```

**Execution is a sequence of read/evaluate steps**

- Reads a command line from the user.
- Parses the command line and runs programs on behalf of the user.
void eval(char *cmdline) {
    char *argv[MAXARGS]; /* argv for execve() */
    int bg; /* should the job run in bg or fg? */
    pid_t pid; /* process id */

    bg = parseline(cmdline, argv);
    if (!builtin_command(argv)) {
        if ((pid = fork()) == 0) { /* child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }
        else /* otherwise, don't wait for bg job */
        {
            printf("%d %s", pid, cmdline);
        }
    }
}
void eval(char *cmdline) {
    char *argv[MAXARGS]; /* argv for execve() */
    int bg;             /* should the job run in bg or fg? */
    pid_t pid;          /* process id */

    bg = parseline(cmdline, argv);
    if (!builtin_command(argv)) {
        if ((pid = fork()) == 0) {  /* child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }
        if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
}

Correctly reaps foreground jobs
What about background jobs?
What Is a “Background Job”?  

• Some programs run “for a long time”  
  –Example: “delete this file in two hours”  
  
```
shell> sleep 7200; rm /tmp/junk  # shell stuck for 2 hours
```

• A “background” job is a process we don't want to wait for  
  
```
shell> (sleep 7200 ; rm /tmp/junk) &
[1] 907
shell> # ready for next command
```
Problem with Simple Shell Example

- Does not reap background jobs
  - Those jobs will become zombies when they terminate
    - Will never be reaped because shell (typically) will not terminate
  - Will run out of the maxproc quota imposed by OS
    - so that OS does not run out of memory with too many zombies

**Solution:** Signal
- Have OS interrupt the shell process to alert it when a background job completes
  - `unix> ulimit -u 202752`  # bash syntax

The maximum number of processes available to a single user.
Signals

- A **signal** is a small message that notifies a process of an event
  - sent from the kernel (sometimes at the request of another process) to a process (an “upcall”)
  - only information in a signal is its small integer ID (1-30)

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Default Action</th>
<th>Corresponding Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Terminate</td>
<td>Interrupt (e.g., ctl-c from keyboard)</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>Terminate</td>
<td>Kill program (cannot override or ignore)</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>Terminate &amp; Dump</td>
<td>Segmentation violation</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>Terminate</td>
<td>Timer signal</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>Child stopped or terminated</td>
</tr>
</tbody>
</table>

Check Figure 8.25 in the book for more complete list.
Receiving Signals

• When kernel is about to schedule a runnable process, check for pending signals.
  • If none exists
    – Pass control to next instruction in the logical flow for process
  • Else
    – For each pending signal k, trigger corresponding action for process.

• Each signal type has a default action, e.g.
  – Terminate process: terminates and (optionally) dump core
  – Stop process (until restarted by a SIGCONT signal)
  – Ignore the signal
• User process can “catch” a signal by executing a user-defined function.
Unix systems mechanisms used for sending signals rely on the notion of a process group.
Process Groups

- Every process belongs to exactly one process group

**Diagram:**
- **Shell:**
  - PID: 10
  - PGID: 10
- **Foreground job:**
  - PID: 20
  - PGID: 20
  - Child:
    - PID: 21
    - PGID: 20
    - Child:
      - PID: 22
      - PGID: 20
- **Background job #1:**
  - PID: 32
  - PGID: 32
- **Background job #2:**
  - PID: 40
  - PGID: 40

**Functions:**
- `getpgrp()`
  - Return process group of current process
- `setpgid()`
  - Change process group of a process
Process Groups

- A process can get its group ID using: `pid_t getpgrp(void);`
- By default, a child process belongs to the same process group as its parent.
- A process can change the process group of itself or another process by using: `int setpgid(pid_t pid, pid_t pgid);` (returns 0 on success and -1 on error)
  - Two entities involved: the process whose Pid is specified in setpgid; and the process that calls setpgid (i.e. current process). The two processes can be the same one.
  - If `pid` is zero, the PID of the current process is used as the new group ID.
  - If `pgid` is zero, the PID of the process specified by `pid` is used for the process group ID.
- Example: if process 15213 calls `setpgid(0, 0);` the result is to create a new process group whose process group ID is 15213, and adds process 15213 to this new group.
Sending Signals with /bin/kill Program

- /bin/kill program sends arbitrary signal to a process or process group

- Examples
  
  /bin/kill -9 24818
  Send SIGKILL to process 24818

  /bin/kill -9 -24817
  Send SIGKILL to every process in process group 24817
Sending Signals from the Keyboard

- Typing `ctrl-c` (`ctrl-z`) sends a SIGINT (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

What is a job?
A Job

• Unix shells use the abstraction of a job to represent the processes that are created as a result of evaluating a single command line.

• At any point in time, there is at most one foreground job and zero or more background jobs.
Installing Signal Handlers

- User program can alter default signal action using `signal` syscall
  - `handler_t *signal(int signum, handler_t *handler)
  - Returns: ptr to previous handler if OK, SIG_ERR on error

- Different values for `handler`:
  - SIG_IGN: ignore
  - SIG_DFL: revert to the default action
  - The address of a `signal handler` function
    - Called when process receives signal of type `signum`
    - When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal
Example: Reacting to Ctrl-c

```c
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void handler(int sig) {
    printf("You think hitting ctrl-c will stop the bomb?\n");
    sleep(2);
    printf("Well...\n");
    sleep(1);
    printf("OK\n");
    exit(0);
}

main() {
    signal(SIGINT, handler); /* installs ctl-c handler */
    while(1) {
    }
}
```

Linux> ./external
<ctrl-c>
You think hitting ctrl-c will stop the bomb?
Well...OK
Linux>
```
Signal Handler Race

```c
int ccount = 0;
void child_handler(int sig)
{
    int child_status;
    pid_t pid;
    pid = waitpid(-1, &child_status, WNOHANG)
    ccount--;
}
void fork14()
{
    int i;
    signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i++) {
        if (fork() == 0) {
            sleep(1);
            exit(0); /* Child: Exit */
        } else{
            ccount++;
        }
    }
    while (ccount != 0)
        pause(); suspends program execution until a signal arrives
    }

Does this program always terminate?

- Wait for any child process
- Return immediately if no child has exited

- Pending signals are not queued
  - For each signal type, just have single bit indicating whether or not signal is pending
```
Signal Handler Race

```c
int ccount = 0;
void child_handler(int sig)
{
    int child_status;
    pid_t pid;
    while ((pid = waitpid(-1, &child_status, WNOHANG)) > 0)
        ccount--;
}

void fork14()
{
    int i;
    signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i++) {
        if (fork() == 0) {
            sleep(1);
            exit(0); /* Child: Exit */
        }else{
            ccount++;
        }
    }
    while (ccount != 0)
        pause();  // suspends program execution until a signal arrives
}
```

- Wait for any child process
- Return immediately if no child has exited
- Pending signals are not queued
  - For each signal type, just have a single bit indicating whether or not the signal is pending
Conclusions

• 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
  – Don’t have queues
    • Just one bit for each pending signal type