Woo-Yeong Jeong (wooyeong@csl.skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Multitasking (1)

- Programmer’s model of multitasking
  - `fork()` spawns new process
    - Called once, returns twice
  - `exit()` terminates own process
    - Called once, never returns
    - Puts it into “zombie” status
  - `wait()` and `waitpid()` wait for and reap terminated children
  - `execve()` runs new program in existing process
    - Called once, (normally) never returns
Multitasking (2)

- Programming challenge
  - Understanding the nonstandard semantics of the functions.
  - Avoiding improper use of system resources.
    - “Fork bombs”
    - Zombie processes not reaped by parents, etc.
Shell

- **Definition**
  - An application program that runs programs on behalf of the user
    - sh: Original Unix Bourne Shell
    - csh: BSD Unix C Shell
    - tcsh: Enhanced C Shell
    - bash: Bourne-Again Shell

Execution is a sequence of read/evaluate steps

```c
int main()
{
    char cmdline[MAXLINE];
    while (1) {
        /* read */
        printf("> ");
        fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);
        /* evaluate */
        eval(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) { /* parent waits for fg job to terminate */
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        } else         /* otherwise, don’t wait for bg job */
            printf("%d %s", pid, cmdline);
    }
}
Simple Shell Example (2)

- Problem with Simple Shell example
  - Shell correctly waits for and reaps foreground jobs.
  - But what about background jobs?
    - Will become zombies when they terminate.
    - Will never be reaped because shell (typically) will not terminate.
    - Creates a memory leak that will eventually crash the kernel when it runs out of memory.

- Solution
  - Reaping background jobs requires a mechanism called a signal.
Signals

Definition

- A signal is a small message that notifies a process that an event of some type has occurred in the system.
  - Kernel abstraction for exceptions and interrupts.
  - Sent from kernel (sometimes at the request of another process) to a process.
  - Different signals are identified by small integer ID’s.
  - The only information in a signal is its ID and the fact that it arrived.

<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 from keyboard (ctrl-c)</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>
Signal Concepts (1)

- Sending a signal
  - Kernel *sends* (delivers) a signal to a destination process by updating some state in the context of the destination process.
  - Kernel sends a signal for one of the following reasons:
    - Generated internally:
      » Divide-by-zero (*SIGFPE*)
      » Termination of a child process (*SIGCHLD*), ...
    - Generated externally:
      » *kill* system call by another process to request signal to the destination process.
Signal Concepts (2)

- Receiving a signal
  - A destination process receives a signal when it is forced by the kernel to react in some way to the delivery of the signal.
  - Three possible ways to react:
    - Ignore the signal (do nothing)
    - Terminate the process
    - Catch the signal by executing a user-level function called a signal handler.
      - Akin to a hardware exception handler being called in response to an asynchronous interrupt.
**Signal Concepts (3)**

- **Signal semantics**
  - A signal is *pending* if it has been sent but not yet received.
    - There can be at most one pending signal of any particular type.
    - Signals are not queued!
  - A process can *block* the receipt of certain signals.
    - Blocked signals can be delivered, but will not be received until the signal is unblocked.
    - There is one signal that can not be blocked by the process. *(SIGKILL)*
  - A pending signal is received at most once.
    - Kernel uses a bit vector for indicating pending signals.
Signal Concepts (4)

- Implementation

  - Kernel maintains **pending** and **blocked** bit vectors in the context of each process.
    - **pending** – represents the set of pending signals
      » Kernel sets bit k in **pending** whenever a signal of type k is delivered.
      » Kernel clears bit k in **pending** whenever a signal of type k is received.
    - **blocked** – represents the set of blocked signals
      » Can be set and cleared by the application using the **sigprocmask** function.
- Every process belongs to exactly one process group.

- **getpgrp()** – Return process group of current process

- **setpgid()** – Change process group of a process
Sending Signals (1)

- 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.
Sending Signals (2)

- int kill(pid_t pid, int sig)
  - Can be used to send any signal to any process group or process.
    - \( \text{pid} > 0 \), signal \( \text{sig} \) is sent to \( \text{pid} \).
    - \( \text{pid} == 0 \), \( \text{sig} \) is sent to every process in the process group of the current process.
    - \( \text{pid} == -1 \), \( \text{sig} \) is sent to every process except for process 1.
    - \( \text{pid} < -1 \), \( \text{sig} \) is sent to every process in the process group - \( \text{pid} \).
    - \( \text{sig} == 0 \), no signal is sent, but error checking is performed.

- /bin/kill program sends arbitrary signal to a process or process group.
  
  $ \text{kill 10231} \quad // \quad \text{SIGTERM} : \text{default signal}$
  
  $ \text{kill -9 10231} \quad // \quad \text{SIGKILL}$
void fork12() {
    pid_t pid[N];
    int i, child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            while(1); /* Child infinite loop */

    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
Receiving Signals (1)

- Handling signals
  - Suppose kernel is returning from exception handler and is ready to pass control to process p.

  - Kernel computes $\text{pnb} = \text{pending} \& \sim \text{blocked}$
    - The set of pending nonblocked signals for process p
  - if ($\text{pnb} \neq 0$) {
    - Choose least nonzero bit k in $\text{pnb}$ and force process p to receive signal k.
    - The receipt of the signal triggers some action by p.
    - Repeat for all nonzero k in $\text{pnb}$.
  }
  - Pass control to next instruction in the logical flow for p.
Receiving Signals (2)

- **Default actions**
  - Each signal type has a predefined default action, which is one of:
    - The process terminates.
    - The process terminates and dumps core.
    - The process stops until restarted by a `SIGCONT` signal.
    - The process ignores the signal.
Installing Signal Handlers

- **sighandler_t signal (int sig, sighandler_t handler)**
  - typedef void (*sighandler_t)(int);
  - The signal function modifies the default action associated with the receipt of signal `sig`.

- **Different values for handler:**
  - SIG_IGN: ignore signals of type `sig`.
  - SIG_DFL: revert to the default action.
  - Otherwise, handler is the address of a signal handler.
    - Called when process receives signal of type `sig`.
    - Referred to as “installing” the signal handler.
    - Executing handler is called “catching” or “handling” the signal.
    - 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.
Handling Signals (1)

- Things to remember
  - Pending signals are not queued.
    - For each signal type, just have single bit indicating whether or not signal is pending.
    - Even if multiple processes have sent this signal.
  - A newly arrived signal is blocked while the handler of the signal is running.
  - Sometimes system calls such as `read()` are not restarted automatically after they are interrupted by the delivery of a signal.
    - They return prematurely to the calling application with an error condition. (`errno == EINTR`)
Handling Signals (2)

- What is the problem of the following code?

```c
int ccount = 0;

void handler (int sig) {
    pid_t pid = wait(NULL);
    ccount--;
    printf ("Received signal %d from pid %d\n", sig, pid);
}

void fork14() {
    pid_t pid[N];
    int i;
    ccount = N;
    signal (SIGCHLD, handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            /* child */
            exit(0);
    while (ccount > 0)
        sleep (5);
}
```
Handling Signals (3)

- Dealing with nonqueueing signals.

```c
int ccount = 0;
void handler2 (int sig) {
    int child_status;
    pid_t pid;
    while ((pid = wait(&child_status)) > 0) {
        ccount--;
        printf ("Received signal %d from pid %d\n", sig, pid);
    }
}
void fork15() {
    pid_t pid[N];
    int i;
    ccount = N;
    signal (SIGCHLD, handler2);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            /* child */
            exit(0);
    while (ccount > 0)
        sleep (5);
}
```
Handling Signals (4)

- React to externally generated events
  - Example: CTRL-C (SIGINT)

```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...");
    fflush(stdout);
    sleep(1);
    printf("OK\n");
    exit(0);
}

main() {
    signal(SIGINT, handler); /* installs ctrl-c handler */
    while(1) {
    }
}
```
Handling Signals (5)

- React to internally generated events
  - Example: `alarm(int t)` sends `SIGALRM` after `t` seconds.

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

int beeps = 0;
void handler(int sig) {
    printf("BEEP\n");
    fflush(stdout);
    if (++beeps < 5) alarm(1);
    else {
        printf("BOOM!\n");
        exit(0);
    }
}

main() {
    signal(SIGALRM, handler);
    alarm(1); // send SIGALRM in 1 second
    while(1) {
    }
}
```

```bash
linux> a.out
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
linux>
```