Signals

Prof. Jinkyu Jeong (jinkyu@skku.edu)
TA – Seokha Shin (seokha.shin@csl.skku.edu)
TA - Jinhong Kim (jinhong.kim@csl.skku.edu)

Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Multitasking

- 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
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);
    }
}
```
Simple Shell Example (1)

```c
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.
", 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**.
Signal

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>#</th>
<th>Signal Name</th>
<th>Default Action</th>
<th>Comment</th>
<th>POSIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIGHUP</td>
<td>Abort</td>
<td>Hangup of controlling terminal or process</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Abort</td>
<td>Interrupt from keyboard</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>SIGQUIT</td>
<td>Dump</td>
<td>Quit from keyboard</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>SIGILL</td>
<td>Dump</td>
<td>Illegal instruction</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>SIGTRAP</td>
<td>Dump</td>
<td>Breakpoint for debugging</td>
<td>No</td>
</tr>
<tr>
<td>6</td>
<td>SIGABRT</td>
<td>Dump</td>
<td>Abnormal termination</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>SIGIOT</td>
<td>Dump</td>
<td>Equivalent to SIGABRT</td>
<td>No</td>
</tr>
<tr>
<td>8</td>
<td>SIGBUS</td>
<td>Abort</td>
<td>Bus error</td>
<td>No</td>
</tr>
<tr>
<td>9</td>
<td>SIGFPE</td>
<td>Dump</td>
<td>Floating point exception</td>
<td>Yes</td>
</tr>
<tr>
<td>10</td>
<td>SIGHRT</td>
<td>Abort</td>
<td>Forced process termination</td>
<td>Yes</td>
</tr>
<tr>
<td>11</td>
<td>SIGRTM</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>12</td>
<td>SIGRTX</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>13</td>
<td>SIGRTY</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>14</td>
<td>SIGRTY</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>15</td>
<td>SIGRTY</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>16</td>
<td>SIGRTY</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>17</td>
<td>SIGRTY</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>18</td>
<td>SIGRTY</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>19</td>
<td>SIGRTY</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>20</td>
<td>SIGRTY</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>21</td>
<td>SIGRTY</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>22</td>
<td>SIGRTY</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>23</td>
<td>SIGRTY</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>24</td>
<td>SIGRTY</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>25</td>
<td>SIGRTY</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>26</td>
<td>SIGRTY</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>27</td>
<td>SIGRTY</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>28</td>
<td>SIGRTY</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>29</td>
<td>SIGRTY</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>30</td>
<td>SIGRTY</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
<tr>
<td>31</td>
<td>SIGRTY</td>
<td>Abort</td>
<td>Available to processes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 9-1. The First 31 Signals in Linux/i386
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:
  - Explicitly ignore the signal
  - Execute the default action
  - **Catch** the signal by invoking `signal-handler` function
    - Akin to a hardware exception handler being called in response to an asynchronous interrupt.
Signal Concepts (3)

- **Default actions**
  - **Abort**
    - The process is destroyed
  - **Dump**
    - The process is destroyed & core dump
  - **Ignore**
    - The signal is ignored
  - **Stop**
    - The process is stopped
  - **Continue**
    - If the process is stopped, it is put into running state
Signal Concepts (4)

- 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 cannot be blocked by the process. *(SIGKILL)* *(One more... SIGSTOP)*
  - A pending signal is received at most once.
    - Kernel uses a bit vector for indicating pending signals.
Signal Concepts (5)

- **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.
Process Groups

- 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.

![Diagram of process groups and signals]

```
Foreground job
```
```
Background job #1
```
```
Background job #2
```
```
Child
```
```
Child
```

```
Foreground process group 20
```
```
Background process group 32
```
```
Background process group 40
```

 pid=10  pgid=10
 pid=20  pgid=20
 pid=21  pgid=20
 pid=22  pgid=20
 pid=32  pgid=32
 pid=40  pgid=40
```
## Sending Signals (2)

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

- **/bin/kill program sends arbitrary signal to a process or process group.**
  - `$ kill 10231` // SIGTERM : default signal
  - `$ kill -9 10231` // SIGKILL
Sending Signals (3)

```c
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 output of the following program?

```c
pid_t pid;
int counter = 2;

void handler1(int sig) {
    counter = counter - 1;
    printf("%d", counter);
    fflush(stdout);
    exit(0);
}

int main() {
    signal(SIGUSR1, handler1);
    printf("%d", counter);
    fflush(stdout);

    if((pid = fork()) == 0) while(1);
    kill(pid, SIGUSR1);
    waitpid(-1, NULL, 0);
    counter = counter + 1;
    printf("%d", counter);
    exit(0);
}
```
Handling Signals (3)

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);
}
```
Exercise #1

- Deal with non-queueing signals

```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);
}
```
Exercise #2

- React to internally generated events

- Make alarm for every 1 second
  - Print “BEEP” for each second
  - Tip: `alarm(int t)` send SIGALRM after t seconds
Exercise #3

- React to externally generated events

- Make zombie process
  - When the process get ctrl+c signal from keyboard, it just prints “beep” to the monitor 5 times with 1-second interval
  - Print “I’m Alive!” to the monitor after 5-times beep