Exceptions, Processes and Signals

Computer Systems Organization (Spring 2017)
CSCI-UA 201, Section 3

Instructor: Joanna Klukowska

Slides adapted from
Randal E. Bryant and David R. O’Hallaron (CMU)
Mohamed Zahran (NYU)
Shells

See https://en.wikipedia.org/wiki/Shell_(computing)
Linux Process Hierarchy

init [1]

Daemon
e.g. httpd

Login shell

Child

Grandchild

Login shell

Child

Grandchild

…

Note: you can view the hierarchy using the Linux pstree command
Shell Programs

A **shell** is an application program that runs programs on behalf of the user.

- **sh** Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
- **csh/tcsh** BSD Unix C shell
- **bash** “Bourne-Again” Shell (default Linux shell)

```c
int main()
{
    char cmdline[MAXLINE]; /* command line */

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

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

*Execution is a sequence of read/evaluate steps*
Simple Shell `eval` Function

```c
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];    /* Holds modified command line */
    int bg;               /* Should the job run in bg or fg? */
    pid_t pid;            /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv); //return indicator if it was terminated by &
    if (argv[0] == NULL)
        return; /* Ignore empty lines */

    if (!builtin_command(argv)) { //run a program that corresponds to the command
        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);
        }
    }

    /* Parent waits for foreground job to terminate */
    if (!bg) {
        int status;
        if (waitpid(pid, &status, 0) < 0)
            unix_error("waitbg: waitpid error");
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}
```

**Problem:** we never reap the jobs that are run in the background.

**Solution:** Exceptional control flow

- The kernel will interrupt regular processing to alert us when a background process completes
- In Unix, the alert mechanism is called a **signal**
Signals
Signals

A signal is a small message that notifies a process that an event of some type has occurred in the system

- Similar to exceptions and interrupts
- Sent from the kernel (sometimes at the request of another process) to a process
- Signal type is identified by small integer ID’s (1-30)
- 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>User typed 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</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: 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:

- Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
- Another process has invoked the **kill system call** to explicitly request the kernel to send a signal to the destination process

This is not the same as the kill signal. It is a system call used for sending signals (any signals, not just the SIGKILL).
Signal Concepts: 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.

Some possible ways to react:
- Ignore the signal (do nothing)
- Terminate the process (with optional core dump)
- Catch the signal by executing a user-level function called signal handler
Signal Concepts: Pending and Blocked Signals

- **A signal is pending if sent but not yet received**
  - There can be at most one pending signal of any particular type
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded

- **A process can block the receipt of certain signals**
  - Blocked signals can be delivered, but will not be received until the signal is unblocked

- **A pending signal is received at most once**

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 when a signal of type k is delivered
  - Kernel clears bit k in pending when a signal of type k is received

- **blocked**: represents the set of blocked signals
  - Can be set and cleared by using the `sigprocmask` function
  - Also referred to as the signal mask.
Sending Signals: Process Groups

Every process belongs to exactly one process group

- Foreground job
  - Child
    - Child
      - pid=21
      - pgid=20
  - pid=20
  - pgid=20

- Background job #1
  - pid=32
  - pgid=32

- Background job #2
  - pid=40
  - pgid=40

- Foreground process group 20

- Background process group 32

- Background process group 40

getpgrp()
Return process group of current process

setpgid()
Change process group of a process (see text for details)
Sending Signals with `/bin/kill (or just kill) Program`

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

- Examples
  - `kill -9 24818`
    Send SIGKILL to process 24818
  - `kill -9 -24817`
    Send SIGKILL to every process in process group 24817

```
linux> ./forks 16
Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

linux> ps

PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24818 pts/2    00:00:02 forks
24819 pts/2    00:00:02 forks
24820 pts/2    00:00:00 ps

linux> /bin/kill -9 -24817

linux> ps

PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24823 pts/2    00:00:00 ps
```
Sending Signals with \texttt{kill} System Call

```c
void fork12()
{
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Infinite Loop */
            while(1)
                ;
        }

    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    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

Suppose the kernel is returning from an exception handler and is ready to pass control to process p.
Receiving Signals

Suppose kernel is returning from an exception handler and is ready to pass control to process p

Kernel computes \( pnb = \text{pending} \land \neg \text{blocked} \)
- The set of pending nonblocked signals for process p

If \((pnb == 0)\)
- Pass control to next instruction in the logical flow for p

Else
- Choose least nonzero bit \( k \) in \( pnb \) and force process p to receive signal \( k \)
- The receipt of the signal triggers some action by p
- Repeat for all nonzero \( k \) in \( pnb \)
- Pass control to next instruction in logical flow for p
Default Actions

Each signal type has a predefined default action, which is one of:

- The process terminates
- The process stops until restarted by a SIGCONT signal
- The process ignores the signal
Installing Signal Handlers

The signal function modifies the default action associated with the receipt of signal `signum`:

```c
handler_t *signal(int signum, handler_t *handler)
```

Different values for handler:

- `SIG_IGN`: ignore signals of type `signum`
- `SIG_DFL`: revert to the default action on receipt of signals of type `signum`
- Otherwise, handler is the address of a user-level signal handler
  - Called when process receives signal of type `signum`
  - Referred to as “installing” the 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
Signal Handling Example

```c
void sigint_handler(int sig) /* SIGINT handler */
{
    printf("So you think you can stop the bomb with ctrl-c, do you?\n");
    sleep(2);
    printf("Well...");
    fflush(stdout);
    sleep(1);
    printf("OK. :-)\n");
    exit(0);
}

int main()
{
    /* Install the SIGINT handler */
    if (signal(SIGINT, sigint_handler) == SIG_ERR)
        unix_error("signal error");

    /* Wait for the receipt of a signal */
    pause();

    return 0;
}
```
Nested Signal Handlers

- Handlers can be interrupted by other handlers

1. Program catches signal s
2. Control passes to handler S
3. Program catches signal t
4. Control passes to handler T
5. Handler T returns to handler S
6. Handler S returns to main program
7. Main program resumes
Blocking and Unblocking Signals

- Implicit blocking mechanism
  - Kernel blocks any pending signals of type currently being handled.
  - E.g., A SIGINT handler can't be interrupted by another SIGINT (because only one signal of a given type is allowed)

- Explicit blocking and unblocking mechanism
  - `sigprocmask` function

- Supporting functions
  - `sigemptyset` – Create empty set
  - `sigfillset` – Add every signal number to set
  - `sigaddset` – Add signal number to set
  - `sigdelset` – Delete signal number from set

```c
sigset_t mask, prev_mask;

Sigemptyset(&mask); //create empty blocking mask
Sigaddset(&mask, SIGINT); //add SIGINT to the mask

/* Block SIGINT and save previous blocked set */
Sigprocmask(SIG_BLOCK, &mask, &prev_mask);

    /* Code region that will not be interrupted by SIGINT */

/* Restore previous blocked set, unblocking SIGINT */
Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```
Safe Signal Handling

- Handlers are tricky because they are concurrent with main program and share the same global data structures.
  - Shared data structures can become corrupted.
  - Misusing by assuming that signals are queued.

- Read about signals on your Linux system:

```bash
man 7 signal
```

- Some functions do not work well with signals (like `printf`)

- Signal handling is not portable between systems

- Newer version of signal handlers is `sigaction` (see the book for more details)
Pending signals are not queued
• For each signal type, one bit indicates whether or not signal is pending...
• ...thus at most one pending signal of any particular type.

You can’t use signals to count events, such as children terminating.

```c
int ccount = 0;
void child_handler(int sig) {
    int olderrno = errno;
    pid_t pid;
    if ((pid = wait(NULL)) < 0)
        Sio_error("wait error");
    ccount--;
    Sio_puts("Handler reaped child ");
    Sio_putl((long)pid);
    Sio_puts("\n");
    sleep(1);
    errno = olderrno;
}

void fork14() {
    pid_t pid[N];
    int i;
    ccount = N;
    Signal(SIGCHLD, child_handler);

    for (i = 0; i < N; i++) {
        if ((pid[i] = Fork()) == 0) {
            Sleep(1);
            exit(0); /* Child exits */
        }
    }
    while (ccount > 0) /* Parent spins */
    ;
}
```

> ./forks 14
Handler reaped child 23240
Handler reaped child 23241
Correct Signal Handling

Must wait for all terminated child processes
  • Put \texttt{wait} in a loop to reap all terminated children

void child_handler2(int sig)
{
    int olderrno = errno;
    pid_t pid;
    while ((pid = wait(NULL)) > 0) {
        ccount--;
        Sio_puts("Handler reaped child ");
        Sio_putl((long)pid);
        Sio_puts(" \n");
    }
    if (errno != ECHILD)
    
      Sio_error("wait error");
    errno = olderrno;
}