Exceptions, Processes and Signals

Computer Systems Organization (Spring 2016)
CSCI-UA 201, Section 2

Instructor: Joanna Klukowska

Slides adapted from
Randall E. Bryant and David R. O'Hallaron (CMU)
Mohamed Zahran (NYU)

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 */
    if (argc == 2) {
        while (1) {
            /* read */
            printf("\n");
            fgets(cmdline, MAXLINE, stdin);
            if (feof(stdin))
                exit(0);
            /* evaluate */
            eval(cmdline);
        }
    }
}
```

Linux Process Hierarchy

Note: you can view the hierarchy using the Linux `pstr` command

Simple Shell `eval` Function

```c
void eval(char *cmdline)
{
    char arg[256]; /* Argument list string */
    char buf[MAXLINE]; /* Holds modified command line */
    int bg; /* Should the job run in bg or fg? */
    int tmp; /* Process id */
    int *argv = NULL; /* Argument vector */
    int argc = 0; /* Size of argv */
    int status;
    int pid;
    char *argp = cmdline;
    while (*argp != 0) {
        if (*argp == ' ') {
            // set command to null
            *argp = 0;
        } else if (*argp == '-') {
            if (argc > 0) {
                // Child gets new job
                if (fork() == 0) {
                    execvp(argv[0], argv);
                    exit(0);
                }
            }
        }
        argv[argc++] = argp;
        argp += strlen(argp) + 1;
    }
    argv[argc] = NULL;
    execvp(argv[0], argv);
    exit(0);
}
```

Signals

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**

- 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.

**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.

**Sending Signals: Process Groups**

- Every process belongs to exactly one **process group**.

**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
Sending Signals with \texttt{kill} System Call

void \texttt{fork}() {
    \texttt{pid} = \texttt{posix}();
    \texttt{int} \texttt{child}\_\texttt{status}
    \texttt{for} (i = 0; i < M; i++) {
        if (\texttt{pid}[i] = \texttt{fork}() == 0) {
            // Child: "Definite Loop "
            \texttt{while}(1)
        }
    }
    \texttt{for} (i = 0; i < M; i++) {
        \texttt{printf}("parent\_\texttt{pid}[i]: \texttt{pid}[i]\n, \texttt{pid}[i]);
    }
    \texttt{fork}(\texttt{child}\_\texttt{status});
    \texttt{if} (\texttt{WIFEXITED} (\texttt{child}\_\texttt{status}))
        \texttt{printf}("Child \%d terminated with exit status \%d", \texttt{wpid}, \texttt{WEXITSTATUS} (\texttt{child}\_\texttt{status}));
    \texttt{else}
        \texttt{printf}("Child \%d terminated abnormally\", \texttt{wpid});
}

Receiving Signals

- Suppose kernel is returning from an exception handler and is ready to pass control to process \texttt{p}
- Kernel computes \texttt{pnb = pending & ~blocked}
  - The set of pending nonblocked signals for process \texttt{p}
- If (\texttt{pnb} == 0)
  - Pass control to next instruction in the logical flow for \texttt{p}
- Else
  - Choose least nonzero bit k in \texttt{pnb} and force process \texttt{p} to receive signal \texttt{k}
  - The receipt of the signal triggers some action by \texttt{p}
  - Repeat for all nonzero \texttt{k} in \texttt{pnb}
  - Pass control to next instruction in logical flow for \texttt{p}

Default Actions

- Each signal type has a predefined \texttt{default action}, which is one of:
  - The process terminates
  - The process stops until restarted by a \texttt{SIGCONT} signal
  - The process ignores the signal

Installing Signal Handlers

- The signal function modifies the default action associated with the receipt of signal \texttt{signum}:
  \texttt{handler}_\texttt{t} *\texttt{signal} (int \texttt{signum, handler}_\texttt{t} *\texttt{handler})
- Different values for handler:
  - \texttt{SIG_IGN}: ignore signals of type \texttt{signum}
  - \texttt{SIG_DFL}: revert to the default action on receipt of signals of type \texttt{signum}
  - Otherwise, handler is the address of a user-level signal handler
    - Called when process receives signal of type \texttt{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

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

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

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

    return 0;
}
```

sigint.c
Nested Signal Handlers

- Handlers can be interrupted by other handlers

![Diagram of signal handling process]

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.
  - Missing by assuming that signals are queued.
- Read about signals on your Linux system: `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)

Correct Signal Handling

- Must wait for all terminated child processes
  - Put `wait` in a loop to reap all terminated children

```
void child_handler2(int sig)
{
    int olderno = errno;
    pid_t pid;
    while ((pid = wait(NULL)) > 0) {
        count--;
        sio.puts("Handler reaped child ");
        sio.puts((long)pid);
        sio.puts("\n");
    }
    if (errno != ECHILD)
        sio.error("Wait error");
    errno = olderno;
}
```

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

```
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);
```

In-Correct Signal Handling

- 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.

- Jork 14
  - Handler reaped child 23240
  - Handler reaped child 23241

- Jork 15
  - Handler reaped chlc 23246
  - Handler reaped chlc 23247
  - Handler reaped chlc 23248
  - Handler reaped chlc 23249
  - Handler reaped chlc 23250