The image part with relationship ID rId2 was not found in the file.

Department of Computer Science

Lecture #25 – part 2

Signals (Ch. 8-Sec. 8.5)

Prof. Soraya Abad-Mota, PhD

The image part with relationship ID rId2 was not found in the file.

Topics

Exceptional Control Flow (3)

- Process Control
  - system calls for manipulating processes:
    - destruction and reaping
    - loading and running programs
    - process commands and shells (sec. 8.4.6)
- Signals (Ch. 8 - sec. 8.5)

The image part with relationship ID rId13 was not found in the file.

ECF Exists at All Levels of a System

- Exceptions
  - Hardware and operating system kernel software
- Process Context Switch
  - Hardware timer and kernel software
- Signals
  - Kernel software and application software
- Nonlocal jumps
  - Application code

ECF(1) and (2)

ECF(3)

and
Textbook and supplemental slides

The image part with relationship ID rId13 was not found in the file.

(Linux) Signals

- A *signal* is a small message that notifies a process that an event of some type has occurred in the system.
  - Akin 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) fig.8.26 p.757
  - Only information in a signal: its ID and the fact that it arrived

ID	*Name*	*Default Action*	*Corresponding Event*
2	SIGINT	Terminate	User typed ctrl-c
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

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

The image part with relationship ID rId13 was not found in the file.

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*
    - Akin to a hardware exception handler being called in response to an asynchronous interrupt:

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.

Signal Concepts: *Pending/Blocked Bits*

- 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 (explicitly) be set and cleared by using the **`sigprocmask`** function
    - Also referred to as the *signal mask*.

The image part with relationship ID rId13 was not found in the file.

Sending Signals: Process Groups

- Every process belongs to exactly one process group

`getpgrp()`
Return process group of current process

`setpgid()`
Change process group of a process (see text for details)

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 (pgid negative means to send kill to all Ps in the group)

```
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
linux>
```

Topics

- Signals
  - definition
  - sending and receiving
  - pending and blocked + process groups
  - sending signals from the keyboard
  - Signal handlers and Signal Concurrency (Secs. 8.5.5, 8.5.6, 8.5.7)

The image part with relationship ID rId13 was not found in the file.

Sending Signals from the Keyboard

- Typing ctrl-c (ctrl-z) causes the kernel to send 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

Example of `ctrl-c` and `ctrl-z`

```
bluefish> ./forks 17
Child: pid=28108 pgrp=28107
Parent: pid=28107 pgrp=28107
<types ctrl-z>
Suspended
bluefish> ps w
  PID TTY      STAT   TIME COMMAND
27699 pts/8    Ss     0:00 -tcsh
28107 pts/8    T      0:01 ./forks 17
28108 pts/8    T      0:01 ./forks 17
28109 pts/8    R+     0:00 ps w
bluefish> fg
./forks 17
<types ctrl-c>
bluefish> ps w
  PID TTY      STAT   TIME COMMAND
27699 pts/8    Ss     0:00 -tcsh
28110 pts/8    R+     0:00 ps w
```

STAT (process state) Legend:

First letter:

S: sleeping

T: stopped

R: running

Second letter:

s: session leader

+: foreground proc group

See “man ps” for more details

Sending Signals with `kill` Function

```
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);
    }
}
```

forks.c

Receiving Signals

- Suppose kernel is returning from an exception handler and is ready to pass control to a process *B*

The image part with relationship ID rId13 was not found in the file.

Receiving Signals

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

- Kernel computes `pnb = pending & ~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*

The image part with relationship ID rId13 was not found in the file.

Default Actions

- Each signal type has a predefined *default action*, which is one of: (Fig. 8.26 p. 757)
  - The process terminates
  - The process stops until restarted by a SIGCONT signal
  - The process ignores the signal
- Process can modify default action using `signal` function, except for `SIGSTOP` and `SIGKILL` (p. 762)
- What we will see next is that the programmer may modify this default action by creating and calling their own *signal handler.*

The image part with relationship ID rId13 was not found in the file.

Topics

- Signals
  - definition
  - sending and receiving
  - pending and blocked + process groups
  - Signal handlers and Signal Concurrency (Secs. 8.5.5, 8.5.6, 8.5.7)

The image part with relationship ID rId13 was not found in the file.

Installing Signal Handlers

- The `signal` function modifies the default action associated with the receipt of signal `signum`:
  - **`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`**: 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 the instruction in the control flow of the process that was interrupted by the receipt of the signal

Signal Handling Example

```
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;
}
```

sigint.c

Topics

- Signals
  - definition
  - sending and receiving
  - pending and blocked + process groups
  - Signal handlers and
  - Signal Concurrency (Secs. 8.5.5, 8.5.6, 8.5.7)

The image part with relationship ID rId13 was not found in the file.

Signals Handlers as Concurrent Flows

- A signal handler is a separate logical flow (not process) that runs concurrently with the main program

Another View of Signal Handlers as Concurrent Flows

The image part with relationship ID rId13 was not found in the file.

Nested Signal Handlers (fig. 8.31)

- Handlers can be interrupted by other handlers

The image part with relationship ID rId13 was not found in the file.

Blocking and Unblocking Signals (sec. 8.5.4)

- 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
- Explicit blocking and unblocking mechanism
  - `sigprocmask` function (changes blocked bit vector, sec. 8.5.1)
  - a mask contains the set of signals to be blocked
  - Supporting functions (helper functions)
    - `sigemptyset` – Create empty set
    - `sigfillset` – Add every signal number to set
    - `sigaddset` – Add signal number to set
    - `sigdelset` – Delete signal number from set

Temporarily Blocking Signals

fig. 8.32, p. 765

```
sigset_t mask, prev_mask;

Sigemptyset(&mask);
Sigaddset(&mask, SIGINT);

/* 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);
```

The image part with relationship ID rId13 was not found in the file.

Safe Signal Handling (sec. 8.5.5)

- Handlers are tricky because they are concurrent with main program and share the same global data structures.
  - Shared data structures can become corrupted.

- Concurrency issues are further explored in Chap. 12

- For now:
  - some guidelines to help you avoid trouble (p. 766-769)
  - "…conservative guidelines for writing handlers that are safe to run concurrently."

The image part with relationship ID rId13 was not found in the file.

Guidelines for Writing Safe Handlers pp. 766-769

- G0: Keep your handlers as simple as possible
  - e.g., Set a global flag and return + receiving the signal done by main
- G1: Call only async-signal-safe functions in your handlers (next slide)
  - `printf`, `sprintf`, `malloc`, and `exit` are not safe!
  - re-entrant (access only local vars) or can't be interrupted by signal handler
- G2: Save and restore `errno` on entry and exit
  - So that other handlers don't overwrite your value of `errno`
- G3: Protect accesses to shared data structures by temporarily blocking all signals. To prevent possible corruption.
- G4: Declare global variables as `volatile`
  - To prevent compiler from storing them in a register or cache
- G5: Declare global flags as volatile sig_atomic_t (type for which reads and writes guaranteed to be atomic)
  - *flag*: var. that is only read or written (e.g. flag = 1, not flag++)
  - declared this way does not need to be protected like other globals

Async-Signal-Safety

- Function is *async-signal-safe* if either reentrant (e.g., all variables stored on stack frame, CS:APP3e 12.7.2) or non-interruptible by signals.
- Posix guarantees 117 functions to be async-signal-safe
  - Source: "`man 7 signal`" (from the Linux Foundation)
  - Popular functions on the list: (list on p. 767)
    - `_exit, write, wait, waitpid, sleep, kill`
  - Popular functions that are **not** on the list:
    - `printf, sprintf, malloc, exit`
    - Unfortunate fact: `write` is the only async-signal-safe output function

The image part with relationship ID rId13 was not found in the file.

Examples of signal handlers

- In addition to the guidelines presented here
- The textbook presents specific examples of bad situations with solutions for the programmer of signal handlers to avoid them:
  - Safely generating formatted output (use async-signal-safe funcs.)
  - Cannot use signals to count events
  - Must wait for all terminated child processes using wait (races)
  - Portable signal handling (diff versions of Linux) (pp. 774-775)
  - Synchronizing Flows to Avoid Races (Sec. 8.5.6 p. 776)
  - Explicitly Waiting for Signals (Sec. 8.5.7 p. 778)

Signal handler situations

- Twelve (12) slides illustrate the situations described above
- When you study on your own: read the slides in parallel to the textbook sections
- Show 32 and 34-35
- Then jump to slide 46

The image part with relationship ID rId13 was not found in the file.

Safely Generating Formatted Output

- Use the reentrant SIO (Safe I/O library) from `csapp.c` in your handlers. (instead of printf on slide 20, bomblab)
  - `ssize_t sio_puts(char s[]) /* Put string */`
  - `ssize_t sio_putl(long v) /* Put long */`
  - `void sio_error(char s[]) /* Put msg & exit */`

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

sigintsafe.c

Correct signal handling example

- This is a realistic example because shells and Web servers are implemented in a similar way.
- Can find full explanation for a similar example in the textbook, pp.770-773

The image part with relationship ID rId13 was not found in the file.

Correct Signal Handling (p.770)

```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.c

- 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 in other processes, such as children terminating.

```
whaleshark> ./forks 14
Handler reaped child 23240
Handler reaped child 23241
```

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 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;
}
```

```
whaleshark> ./forks 15
Handler reaped child 23246
Handler reaped child 23247
Handler reaped child 23248
Handler reaped child 23249
Handler reaped child 23250
whaleshark>
```

Portable Signal Handling (pp. 774-775)

- Ugh! Different versions of Unix can have different signal handling semantics
  - Some older systems restore action to default after catching signal
  - Some interrupted system calls can return with errno == EINTR
  - Some systems don't block signals of the type being handled
- Solution: `sigaction` to clearly specify the semantics
- Study on your own: textbook pages and this slide

```
handler_t *Signal(int signum, handler_t *handler)
{
    struct sigaction action, old_action;

    action.sa_handler = handler;
    sigemptyset(&action.sa_mask); /* Block sigs of type being handled */
    action.sa_flags = SA_RESTART; /* Restart syscalls if possible */

    if (sigaction(signum, &action, &old_action) < 0)
        unix_error("Signal error");
    return (old_action.sa_handler);
}
```

csapp.c

Synchronizing Flows to Avoid Races (8.5.6)

- Simple shell with a subtle synchronization error because it assumes parent runs before child.

```
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;

    Sigfillset(&mask_all);
    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        if ((pid = Fork()) == 0) { /* Child */
            Execve("/bin/date", argv, NULL);
        }
        Sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /* Parent */
        addjob(pid);  /* Add the child to the job list */
        Sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    exit(0);
}
```

procmask1.c

Synchronizing Flows to Avoid Races

- SIGCHLD handler for a simple shell

```
void handler(int sig)
{
    int olderrno = errno;
    sigset_t mask_all, prev_all;
    pid_t pid;

    Sigfillset(&mask_all);
    while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap child */
        Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        deletejob(pid); /* Delete the child from the job list */
        Sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    if (errno != ECHILD)
        Sio_error("waitpid error");
    errno = olderrno;
}
```

procmask1.c

Corrected Shell Program without Race

```
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, mask_one, prev_one;

    Sigfillset(&mask_all);
    Sigemptyset(&mask_one);
    Sigaddset(&mask_one, SIGCHLD);
    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
        if ((pid = Fork()) == 0) { /* Child process */
            Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
            Execve("/bin/date", argv, NULL);
        }
        Sigprocmask(SIG_BLOCK, &mask_all, NULL); /* Parent process */
        addjob(pid);  /* Add the child to the job list */
        Sigprocmask(SIG_SETMASK, &prev_one, NULL);  /* Unblock SIGCHLD */
    }
    exit(0);
}
```

procmask2.c

Last example about signals

- Section 8.5.7 + next 5 slides to study on your own.

The image part with relationship ID rId13 was not found in the file.

Explicitly Waiting for Signals (sec. 8.5.7)

- Handlers for program explicitly waiting for SIGCHLD to arrive.

```
volatile sig_atomic_t pid;

void sigchld_handler(int s)
{
    int olderrno = errno;
    pid = Waitpid(-1, NULL, 0); /* Main is waiting for nonzero pid */
    errno = olderrno;
}

void sigint_handler(int s)
{
}
```

waitforsignal.c

Explicitly Waiting for Signals

Similar to a shell waiting for a foreground job to terminate.

```
int main(int argc, char **argv) {
    sigset_t mask, prev;
    Signal(SIGCHLD, sigchld_handler);
    Signal(SIGINT, sigint_handler);
    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);

    while (1) {
    Sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
    if (Fork() == 0) /* Child */
            exit(0);
    /* Parent */
    pid = 0;
    Sigprocmask(SIG_SETMASK, &prev, NULL); /* Unblock SIGCHLD */

    /* Wait for SIGCHLD to be received (wasteful!) */
    while (!pid)
            ;
    /* Do some work after receiving SIGCHLD */
        printf(".");
    }
    exit(0);
}
```

waitforsignal.c

Explicitly Waiting for Signals

- Program is correct, but very wasteful
- Other options:

```
while (!pid)  /* Race! */
    pause();
```

```
while (!pid) /* Too slow! */
    sleep(1);
```

- Solution: `sigsuspend`

The image part with relationship ID rId13 was not found in the file.

Waiting for Signals with `sigsuspend`

- `int sigsuspend(const sigset_t *mask)`

- Equivalent to atomic (uninterruptable) version of:

```
sigprocmask(SIG_BLOCK, &mask, &prev);
pause();
sigprocmask(SIG_SETMASK, &prev, NULL);
```

The image part with relationship ID rId13 was not found in the file.

Waiting for Signals with `sigsuspend`

```
int main(int argc, char **argv) {
    sigset_t mask, prev;
    Signal(SIGCHLD, sigchld_handler);
    Signal(SIGINT, sigint_handler);
    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);

    while (1) {
        Sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
        if (Fork() == 0) /* Child */
            exit(0);

       /* Wait for SIGCHLD to be received */
        pid = 0;
        while (!pid)
            Sigsuspend(&prev);

       /* Optionally unblock SIGCHLD */
        Sigprocmask(SIG_SETMASK, &prev, NULL);
    /* Do some work after receiving SIGCHLD */
        printf(".");
    }
    exit(0);
}
```

sigsuspend.c

Remaining

- Finish Ch. 8 ECF with non-local jumps (.25)
- Virtual Memory (1.75 lectures)
- System-level I/O (.75)
- Concurrency (.75)

- Final Exam: Wednesday December 11, 2024
  - 7:45 to 9:45 pm is the schedule, but I am working on changing it to 5pm, I need to have your agreement to change it. If you all agree, I will ask permission from the Dean of School of Engineering.

The image part with relationship ID rId13 was not found in the file.