CS 201

Signals

Gerson Robboy
Portland State University

Signals

A *signal* is a message that notifies a process that an event of some type has occurred.

Signals are the operating system abstraction for exceptions and interrupts.

- Asynchronous
- Interrupts the process like an interrupt, but via software

Signal Concepts

Sending a signal

- Kernel *sends* (delivers) a signal to a *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 request the kernel to send a signal to the destination process.

Let's pause for a concrete example

A process dereferences a null pointer.

What happens at the hardware level?

You see something like: "Segmentation fault. Core dumped." Your program terminates.

The operating system lingo for how the kernel propagates an exception to the process is, it sends the process a signal.

The process "receives" the signal.

- Usually this means the signal kills the process.

Signal Concepts (cont)

Receiving a signal

- A destination process *receives* a signal sent by the kernel.
- By default, most signals cause the process to terminate.
- But, there's a way a process can handle most signals. We'll see how.
- 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.
    - » Analogous to a kernel exception handler
    - » Asynchronous

A signal is *pending* if it has been sent but not yet received.

Signals

Sent by the kernel 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.

A few frequently seen signals:

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt from keyboard (ctl-c)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

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.

Putting it all together

Remember *exceptions*? An exception is an event that alters the flow of control at the hardware level.

- Control goes to the kernel, via an interrupt vector.
- The kernel has a handler for each kind of event.
- Usually this detour of control is invisible to the user process.

For some events, particularly faults, the kernel handles the event by sending a *signal* to the process.

Signals are the higher level, software abstraction of exceptions.

- Alters the flow of control of a process.
- Can also have a handler (analogous to an exception handler)

Putting it all together (continued)

Signals typically alter the flow of control at the user level.

- By default, most signals (but not all) terminate the process.
- Can send control to a *signal handler* in the user program.

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 for signals of type `signum`.
  - Yes, this is weird, but you can assign these integer values to a pointer.
- Otherwise, handler is the address of a *signal handler*

Signal Handling Example

```
void sigint_handler(int sig)
{
    printf("Process %d received signal %d\n",
            getpid(), sig);
    exit(0);
}

main()
{
    . . .   // Do stuff
    signal(SIGINT, sigint_handler);
    . . .   // Do more stuff
}
```

What exactly happens when signal is called from main?
What will happen when the user hits Control-C?

Signal Handlers

A signal handler is a function you write, to handle a signal.

- **Called when process receives signal of type `signum`**
- **Referred to as "*installing*" the handler.**
- **Executing handler is called "*catching*" or "*handling*" the signal.**

When a signal is received, control is diverted to the handler.

When the handler returns, control passes back to:

- **In some cases, the next instruction.**
- **In some cases, the instruction that was interrupted by an exception.**

`wait`: Synchronizing with children

```
int wait(int *child_status)
```

- What does the `wait` system call actually do?
- What is the default action of SIGCHLD?
- What does a SIGCHLD signal do if you have called wait?
- What does it do if you haven't called wait?
- Can you catch a SIGCHLD signal with a handler?
- What happens if you fork a child, the child exits, you don't have a SIGCHLD handler, and you never call wait?

Exercise

Write a small program that installs a handler for the SIGSEGV signal, and then accesses an illegal memory address in order to execute the handler.

Sending Signals with `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
linux> 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> kill -9 -24817
linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24823 pts/2    00:00:00 ps
linux>
```

Other useful system calls

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

- Sends signal *sig* to process *pid*
- Although the name is 'kill,' it can be used for communication
  - The recipient process can catch the signal if not SIGKILL
  - Synchronization between processes

Other useful system calls

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status,
   int options);
```

- A newer, more versatile form of *wait*
- Can wait on a particular pid, a process group, or all child processes
- Options:
  - WNOHANG: Return immediately if no child has exited (and if a child has exited, return the pid)
  - WUNTRACED: Also return for children which are stopped but not traced.

Other useful system calls

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

- Sends signal *SIGALRM* to this process after *seconds*
- What is the default action for SIGALRM?

Sending Signals with `kill` Function

```
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);
    }
}
```

What does this program do?

```
#include <unistd.h>
main()
{
        int i;
        int j = alarm(6);

        while(1);
        printf("exiting\n");
        exit(0);
}
```

A Program That Reacts to Internally Generated Events

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

int beeps = 0;

/* SIGALRM handler */
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) {
    /* handler returns here */
  }
}
```

What does this program do?

Signals do not interrupt a handler for the same signal

While handling a signal, that signal is *blocked*.

That is one way that signals get to be pending.

When the handler returns, then the blocked signal can be received.

Signals do not have queues

Just one bit for each pending signal type

What happens if many signals arrive at once?

- **The process is handling the first one**
- **The second one is *pending***
- **Other signals may be lost**

```
int ccount = 0;
void child_handler(int sig)
{
    int child_status;
    pid_t pid = wait(&child_status);
    ccount--;
    printf("Received signal %d from process %d\n", sig, pid);
}

int main()
{
    pid_t pid[N];
    int i, child_status;
    ccount = N;
    signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            exit(0);
        }
    while (ccount > 0)
        pause();/* Suspend until signal occurs */
}
```

If a process has many children and wants to reliably know when each one terminates, how do you do it?

Sigaction()

The newer version of *signal* with a zillion options.

Problem: different flavors of Unix have subtle variations in how they handle signals (see sec. 8.5)

- Really a problem to standardize

The POSIX solution : sigaction can specify in detail how signal handling should behave

Use sigaction to write portable signal-handling code.

On POSIX-compliant systems, signal is implemented using sigaction.

- but POSIX doesn't specify in detail how signal should behave.

Summary

Signals provide process-level exception handling

- Can generate signals from user programs
- Can handle them with signal handlers

Some caveats

- Very high overhead
  - >10,000 clock cycles
  - Only use for exceptional conditions
- They don't have queues
  - Just one bit for each pending signal type