Signals

COMP 321

Dave Johnson

1

Review: Interrupts, Exceptions, and Traps

***Interrupt*: An asynchronous signal from a device outside the CPU**

- Examples: someone typed a key on the keyboard, a disk hardware I/O request has completed, or packet arrived over the network

***Exception*: An error executing a specific CPU instruction**

- Privileged instruction, memory protection, divide by zero, etc.
- The error depends on the instruction operands or circumstances

***Trap*: A special CPU instruction used to call kernel services (e.g., x86-64 syscall)**

- Often referred to as a ***kernel call*** or ***system call***
- Write to a file on disk, create a new process, allocate more memory, etc.
- Unlike an exception, this instruction ***always*** causes exception-like behavior

2

A "Signal" is a Software Abstraction of These

Created entirely in software by operating system kernel, not by hardware

- Triggered by the kernel in response to some hardware events, such as
  - Bad attempted memory access by this process (SIGSEGV, SIGBUS)
  - Floating point exception by this process (SIGFPE)
  - Attempt by this process to execute illegal instruction (SIGILL)
  - Interrupt (control-C) typed on the keyboard (SIGINT)
- Triggered by the kernel in response to some software-only events, such as
  - A child of this process has terminated or stopped (SIGCHLD)
  - A software timer set by this process has expired (SIGALRM)
  - Attempted I/O with terminal while in the background (SIGTTIN, SIGTTOU)
- See "man 7 signal" for a list of signals (each identified by a small integer)

A Simple Example: alarm() and SIGALRM

```
unsigned int  alarm(unsigned int seconds);
```

A kernel call to set an alarm clock timer for delivery of a signal

- Arranges for a SIGALRM signal to this process after "seconds" time
- If the alarm clock is already set for this process, it is reset to "seconds"
- If "seconds" is zero, cancels the alarm clock
- Returns the number of seconds that had been left on the alarm

An abstraction of the hardware's clock interrupt hardware mechanism

- Timer and calling interrupt handler procedure are done by hardware
- ***Signals, instead, are an abstraction, created by OS kernel software***

Default Behavior When a Process Receives a Signal

Each type of signal has a "default" behavior – which is one of

- Terminate the process and dump its "core" (i.e., its address space) to a file on disk (e.g., for debugging), or
- Just terminate the process (no core dump), or
- Stop (basically, freeze) the process's execution, or
- Continue (i.e., resume) the process's execution if currently stopped from a signal, or
- Entirely ignore the signal (no effect on the process)

A Signal Handler Procedure Can Override Behavior

A process may have some procedure called when certain signals received by it

- Sometimes referred to as "catching" the signal
- Conceptually similar to an interrupt/exception handler procedure
- But, again, signals are entirely all made by kernel software, not hardware
- Many types are in response to some hardware event, but it is the operating system kernel, in software, that causes some signal to be sent to a process

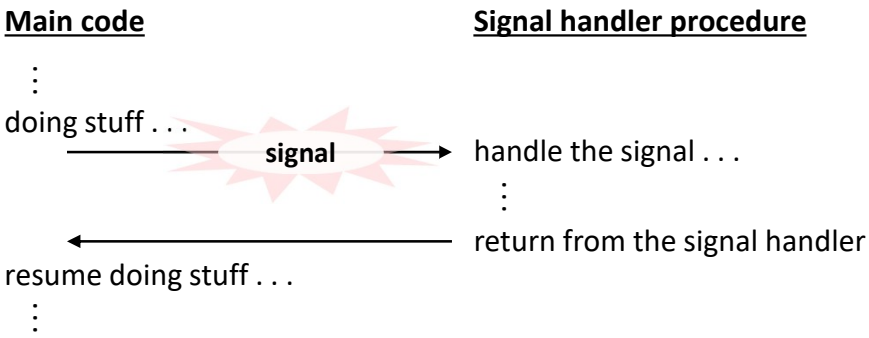
Signal handler procedure can essentially do whatever it wants to, including

- Choose to terminate the process, or
- Do something to "handle" the signal and then return from the signal handler, causing the process to return to where it was when the signal occurred

Calling the Signal Handler

Basically, an asynchronous procedure call to the signal handler

- There is no procedure call instruction there in the main code
- But the procedure call to the signal handler procedure still happens, ***invisible to the main code***



But This Concurrency Can Create Problems

A signal can occur "when you least expect it"

- For example

Main program

```
int  X = 0
             movl   X, %eax
X = X + 1    addl   $1, %eax
             movl   %eax, X
```

signal handler

```
             movl   X, %ebx
X = X + 2    addl   $2, %ebx
             movl   %ebx, X
```

- Possible final results: X = 1 or 3
- Even worse if this is two different signal handlers, instead of main program
  - Possible final results: X = 1 or 2 or 3

Some Tips on Writing Safe Signal Handlers

Do not create this kind of concurrency problem for yourself

- Use only async-signal-safe functions in your signal handler
- Functions like printf, fprintf, sprintf, malloc, free, and exit are ***not*** safe, ***do not use them in a signal handler***
- The "write" kernel call (such as to file descriptor 2 = standard error) is safe
- The textbook also gives you csapp sio_puts, sio_putl, sio_error, etc., functions

Be careful of the global variable errno

- The signal handler call is supposed to be invisible to the main code
- But many things you might do in a signal handler can change errno
- To protect for this, save errno at the top of your handler, restore at the bottom

Variables Shared Between Main Code and Signal

Suppose the main code uses some variables shared with a signal handler

- The signal handler execution is supposed to be invisible to the main code
- But the signal handler could "unexpectedly" change the value of such a shared variable used in the main code
- The compiler doesn't generally understand that this is possible, but it is
- To have the compiler correctly generate code for this case
  - Such shared variables should be declared with "***volatile***" keyword, e.g.,

    volatile int my_var;
  - Lets the compiler know that this variable may change value unexpectedly

Changing or Checking the Behavior for Some Signal

```
int  sigaction(int  signum,
      const struct sigaction * _Nullable restrict act,
      struct sigaction * _Nullable restrict oldact);
```

A process can change the behavior and/or check current behavior of a signal

- "signum" specifies which signal
- "act" points to a struct defining the desired new bahavior when that type of signal is received by this process
  - If NULL, no change to signal's behavior is made
- "oldact" points to a struct that returns current setting of this signal's behavior
  - If NULL, the current setting for this signal is not returned

Dealing with a struct sig_action

A struct sig_action has several fields, some of which overlap or conflict

- Best practice is to
  - Zero out the entire struct sig_action contents (e.g., with memset)
  - Then set just the fields in it that you need
- The most important fields are sa_handler and sa_action
  - Both define handling for that type of signal, but use only one, not both
- Handler may be specified as one of
  - SIG_DFL – Sets handling for this type of signal back to its default
  - SIG_IGN – Requests to ignore this type of signal in the future
  - Address of a function – address of the handler procedure for this signal

Setting Up a Signal Handler for Some Signal

```
void  MySignalHandler(int  signum)
{
      /* … */
}
```

```
struct sigaction  newact;

memset(&newact,  0,  sizeof(newact));
newact.sa_handler = MySignalHandler;
newact.sa_flags = SA_RESTART;

if (sigaction(signum,  &newact,  NULL) < 0)       /* example: signum = SIGALRM */
      perror("sigaction");
```

If the signal interrupts some kernel call, (generally) restart the kernel call after the signal handler returns

Signal Synchronization: Blocking Signals

We need tools to control when certain signals are allowed (or not allowed)

- Setting up signal handler for signal "signum" ***also*** causes that signal to be ***blocked*** during executing inside that signal handler when called
- Prevents the signal handler from interfering with itself in its own execution (and prevents possible infinite recursion if signal occurs frequently)
- This signal is blocked upon calling the signal handler, restored on return

```
struct sigaction  newact;

memset(&newact,  0,  sizeof(newact));
newact.sa_handler = MySignalHandler;
newact.sa_flags = SA_RESTART;

sigaction(signum,  &newact,  NULL);
```

How Are Pending Signals Handled by the Kernel?

For each process, the kernel remembers (e.g., in the process's PCB)

- pending = bitmap with ***1 bit*** for each different ***type*** of signal
  - A bit in the pending bitmap is ***set*** if signal of that type is ***pending***
- blocked = bitmap with ***1 bit*** for each ***type*** of signal
  - A bit in the blocked bitmap is ***set*** if signals of that type are ***blocked***

The signals that may be delivered to the process = (pending & ~blocked)

- When a signal is delivered, the corresponding bit in pending is cleared
- When a signal is unblocked, the corresponding bit in blocked is cleared

Some signals sent to a process may thus get "lost"

- While that signal is blocked, or even if they just occur very close together (setting a pending bit when it's already set has no effect)

Automatically Blocking Additional Signals

```
struct sigaction  newact;
sigset_t  block_sigs;

sigemptyset(&block_sigs);
sigaddset(&block_sigs,  SIGCHLD);
sigaddset(&block_sigs,  SIGINT);

memset(&newact, 0, sizeof(newact));
newact.sa_handler = MySignalHandler;
newact.sa_flags = SA_RESTART;
newact.sa_mask = block_sigs;

if (sigaction(SIGALRM,  &newact,  NULL) < 0)
      err(1,  "sigaction");
```

These additional types of signals will be automatically blocked on calling the signal handler and restored on its return

Example: SIGALRM ***and*** SIGCHLD and SIGINT get blocked when the handler is called

Manually Blocking or Unblocking Signals

```
int  sigprocmask(int  how,  const sigset_t *_Nullable restrict  set,
        sigset_t *_Nullable restrict  oldset);
```

Can be used when needed (e.g., outside of a signal handler)

- "how" defines the kind of change to make (ignored if set == NULL)
  - SIG_BLOCK: block additional signals given by "set"
  - SIG_UNBLOCK: unblock signals given by "set" (OK if already not blocked)
  - SIG_SETMASK: change entire set of blocked signals to just "set"
- The set of blocked signals before the change can be returned in "oldset"
  - Can be used to, e.g., restore the blocked signals back to what they were

Related Topic: Process Groups

```
int  setpgid(pid_t pid,  pid_t pgid);
pid_t  getpgrp(void);
```

Note the strangely different names set**pgid** vs. get**pgrp**

The target of a signal may be a single process or a "process group"

- Each process is always in exactly one process group, identified by a pid_t
- Generally, a process group is a process started directly by the shell, plus that process's children (and grandchildren, etc.), but this depends on the shell
- A process can use setpgid(pid, pgid) to change its own process group anytime
  - If pid == 0, sets the calling process's own process group id
  - if pgid == 0, the process's process group id is set equal to its own pid
- ***A process's process group id is inherited across fork() and execve()***

Process Group Example

A simplified version of how the shell runs commands as process groups

- The code used to run each command (in a loop, once for each command)
- Runs each command (and thus also that child process's own child processes if any) as its own process group

```
pid_t pid = fork();
if (pid == 0) {
        setpgid(0,  0);
        execv(pathname, args);
} else {
        wait(NULL);
}
```

While still running the shell, the child sets own process group equal to its own pid

Another library version of exec

Manually Sending a Signal

```
int  kill(pid_t  pid,  int  sig);
```

Used, e.g., by the "kill" command

Note "/bin/kill" vs. shell's built-in

Manually sends any signal "sig" to a process or a process group

- The target depends on the value of pid
  - If pid > 0, target is that specific process
  - If pid == 0, target is every process in this process's own process group
  - If pid == -1, target is every process this process has permission to signal
  - if pid < -1, target is every process in process group number -pid
- If sig == 0, no signal is sent, but other checks are still performed
  - Can be used, e.g., to check the existence of a process with a given pid

Foreground and Background Process Groups

The real power of process groups

- Suppose you run "cmd1 &" and then "cmd2" (with no &) from the shell
  - What happens when you type control-C on the terminal?
  - Which one of cmd1 and/or cmd2 gets terminated?
- Suppose you run "cmd1 &", "cmd2 &", and "cmd3 &" from the shell
  - What happens when one of them tries to read from standard input?
  - When you type something, which one gets it?
- These kinds of problems are resolved by the ***foreground process group*** vs. ***background process groups***
  - Defined by the process group of the ***terminal*** on which these are running

Terminal Process Groups

```
pid_t tcgetpgrp(int fd);
int tcsetpgrp(int fd, pid_t pgrp);
```

Defining the process group of a terminal

- A ***terminal*** is always controlled by only a ***single*** process group
- Normally, this is the process group running in the ***foreground*** on that terminal
- For the shell, running a new process (i.e., command) and/or "fg" or "bg" cause the shell to change the terminal's process group
  - ***Always set to the process group id of the current foreground group***
- These are library functions, really implemented by calling the kernel calls "int ioctl(int fd, TIOC**G**PGRP, pid_t *argp)" and "int ioctl(int fd, TIOC**S**PGRP, pid_t *argp)"

Signals Sent by Typing on the Terminal

There are 3 different characters you can type to cause terminal to send a signal

- control-C – The terminal sends a SIGINT (interrupt) signal
  - Default behavior is to terminate the process
- control-\ – The terminal sends a SIGQUIT (quit) signal
  - Default behavior is to terminate the process and dump core for it
- control-Z – The terminal sends a SIGTSTP (terminal stop) signal
  - Default behavior is to stop the process (until SIGCONT)

In all cases, the signal is sent (only) to the process group matching the terminal's process group (the foreground process group)

Reading and Writing the Terminal

Reading from the terminal

- Can only be done by the foreground process group
- If the process is not in the foreground group, the terminal sends a SIGTTIN (terminal input) signal to the process group attempting the read
- Causes all processes in that process group to stop (until SIGCONT)

Writing to the terminal

- Can normally be done by foreground or background processes
- But if TOSTOP output mode is set on the terminal and the process is not in the foreground group, the terminal sends a SIGTTOU (terminal output) signal to the process group attempting the write
- Causes all processes in that process group to stop (until SIGCONT)

Expanded Use of the waitpid() Kernel Call

```
pid_t  waitpid(pid_t pid,  int *_Nullable wstatus,  int options);
```

Can wait for child events other than just child process termination

- (Again, pid == -1 means to wait for any process without regard to process ids)
- "options" should be the logical OR of needed options
  - WUNTRACED – Return info also for child processes that have just been stopped, e.g., due to terminal process groups
    - "traced" refers to a process being debugged by gdb (i.e., thus a child of gdb), which always returns status when stopped at a breakpoint)
  - WNOHANG – If nothing to report, return immediately rather than "hanging" (returns 0, which is never a "real" process id)

Example: Reaping All New Child Status Changes

Need to reap all <u>new</u> child status changes in a <u>single</u> SIGCHLD handler call

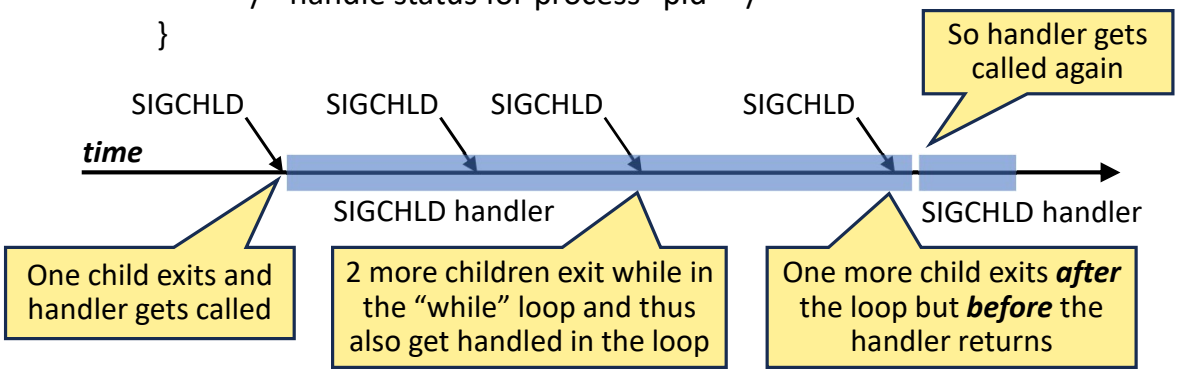
- There is only a ***single*** "pending" bit for each ***type*** of signal
- Additional pending signals of the same type get OR'd together in that single bit
  - And thus get (sort of) "lost"
  - You might get only one SIGCHLD call despite many child processes having exited that you haven't yet reaped
- Thus, a ***single*** SIGCHLD handler call must be able to reap ***all new*** changes

```
while ((pid = waitpid(-1, &wstatus, WUNTRACED|WNOHANG)) > 0) {
        /* handle status for process "pid" */
}
```

Reaping All New Child Status Changes

Need to reap all <u>new</u> child status changes in a <u>single</u> SIGCHLD handler call

```
while ((pid = waitpid(-1, &wstatus, WUNTRACED|WNOHANG)) > 0) {
        /* handle status for process "pid" */
}
```



Summary Definition of Some Signals

- SIGKILL – Terminate the process (***can't*** be caught by a signal handler)
- SIGTERM – Terminate the process (default for the "kill" command)
- SIGCHLD – Report exited child (or other status changes of a child process)
- SIGALRM – A software timer set by this process has expired
- SIGINT – Control-C typed, sent to the foreground process group
- SIGQUIT – Control-\ typed, sent to the foreground process group
- SIGCONT – Restart a process stopped from an earlier signal
- SIGSTOP – Stop the process until restarted with SIGCONT
- SIGTSTP – Control-Z typed, sent to the foreground process group
- SIGTTIN – Attempted background terminal input, sent to that process group
- SIGTTOU – Attempted background terminal output (with TOSTOP enabled)