Operating System Concepts

Lecture 7: Signals

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

Today's class

- Signals
  - Generation
  - Disposition
  - Blocking
- Interprocess communication (IPC)

Signals are software interrupts

- signals provide a way of handling **asynchronous events**
  - they are very short messages, hence can be viewed as a form of IPC

Signals are software interrupts

- signals provide a way of handling **asynchronous events**
  - they are very short messages, hence can be viewed as a form of IPC
- Mac OS X 10.6.8 and Linux 3.2.0 each support 31 different signals

Signals are software interrupts

- signals provide a way of handling **asynchronous events**
  - they are very short messages, hence can be viewed as a form of IPC
- Mac OS X 10.6.8 and Linux 3.2.0 each support 31 different signals
- every signal has a name which begins with SIG
  - names are defined by positive integer constants in the header `<signal.h>`
  - 9 or `SIGKILL`: kill; 11 or `SIGSEGV`: segmentation fault
  - use `kill -l` to get a list of signals (this is architecture dependent)

Signals are software interrupts

- signals provide a way of handling **asynchronous events**
  - they are very short messages, hence can be viewed as a form of IPC
- Mac OS X 10.6.8 and Linux 3.2.0 each support 31 different signals
- every signal has a name which begins with SIG
  - names are defined by positive integer constants in the header `<signal.h>`
  - 9 or `SIGKILL`: kill; 11 or `SIGSEGV`: segmentation fault
  - use `kill -l` to get a list of signals (this is architecture dependent)
- every signal has a default action associated with it
  - (a) terminate, (b) terminate with a **core dump**, (c) ignore, and (d) stop
    - definition of core dump: a memory image of the process is left in a file named core in the current working directory of the corresponding process

Terminology

- a signal is
  - posted/generated/sent if the event that causes the signal has occurred
  - delivered/caught if the associated action is taken
    - this action is referred to as **signal disposition**
  - pending if it has been posted but not yet delivered
    - the intermediate state between generated and delivered
    - signals will be pending if the target process blocks them
  - blocked if the target process does not want it delivered
    - the target process asked (using the signal mask) the kernel to block that signal

Terminology

- a signal is
  - posted/generated/sent if the event that causes the signal has occurred
  - delivered/caught if the associated action is taken
    - this action is referred to as **signal disposition**
  - pending if it has been posted but not yet delivered
    - the intermediate state between generated and delivered
    - signals will be pending if the target process blocks them
  - blocked if the target process does not want it delivered
    - the target process asked (using the signal mask) the kernel to block that signal
- a signal can be process-directed or thread-directed (such as `SIGSEGV` and `SIGFPE`)
  - a process-directed signal may be delivered to any one of the threads that does not currently have the signal blocked

Signal generation

- user presses certain terminal keys
  - Ctrl-C (or DEL) causes `SIGINT` to be generated and sent to foreground processes
  - Ctrl-Z causes `SIGTSTP` to be generated (**Note** `SIGSTOP`≠`SIGTSTP`)
  - Ctrl-\ causes `SIGQUIT` to be generated

Signal generation

- user presses certain terminal keys
  - Ctrl-C (or DEL) causes `SIGINT` to be generated and sent to foreground processes
  - Ctrl-Z causes `SIGTSTP` to be generated (**Note** `SIGSTOP`≠`SIGTSTP`)
  - Ctrl-\ causes `SIGQUIT` to be generated
- the kernel wants to notify a process that its execution has led to a hardware exception (e.g., divide by zero, floating-point overflow, segmentation fault)

Signal generation

- user presses certain terminal keys
  - Ctrl-C (or DEL) causes `SIGINT` to be generated and sent to foreground processes
  - Ctrl-Z causes `SIGTSTP` to be generated (**Note** `SIGSTOP`≠`SIGTSTP`)
  - Ctrl-\ causes `SIGQUIT` to be generated
- the kernel wants to notify a process that its execution has led to a hardware exception (e.g., divide by zero, floating-point overflow, segmentation fault)
- the `kill` command or the `kill(pid_t pid, int sig)` system call is used

Signal generation

- user presses certain terminal keys
  - Ctrl-C (or DEL) causes `SIGINT` to be generated and sent to foreground processes
  - Ctrl-Z causes `SIGTSTP` to be generated (**Note** `SIGSTOP`≠`SIGTSTP`)
  - Ctrl-\ causes `SIGQUIT` to be generated
- the kernel wants to notify a process that its execution has led to a hardware exception (e.g., divide by zero, floating-point overflow, segmentation fault)
- the `kill` command or the `kill(pid_t pid, int sig)` system call is used
- a software condition occurs
  - `SIGURG` (generated when out-of-band data arrives over a network connection)
  - `SIGPIPE` (generated when a process writes to a pipe that has no reader)
  - `SIGALRM` (generated when a timer set by the `alarm` function expires)

POSIX.1 reliable-signal

- sending a signal
  - a process can send a signal to another process or a group of processes
    - using the `kill( )` system call
    - **only if it has permission:** the real or effective user ID of the receiver is the same as that of the sender
  - a process can send a signal with accompanying data to another process (just like IPC)
    - use the `sigqueue( )` function
  - a process can send itself a signal using the `raise( )` function
    - similar to `kill(getpid(), sig)`

POSIX.1 reliable-signal

- sending a signal
  - a process can send a signal to another process or a group of processes
    - using the `kill( )` system call
    - **only if it has permission:** the real or effective user ID of the receiver is the same as that of the sender
  - a process can send a signal with accompanying data to another process (just like IPC)
    - use the `sigqueue( )` function
  - a process can send itself a signal using the `raise( )` function
    - similar to `kill(getpid(), sig)`
- waiting for a signal
  - the `pause( )` system call puts a process to sleep until any signal is caught; this signal can be generated by the `alarm` function
    - it returns only when a signal was caught and the signal-catching function returned

POSIX.1 reliable-signal

- signal dispositions

  1. `SIG_IGN`: ignore the signal (except `SIGKILL` and `SIGSTOP` which can never be ignored as they are surefire ways of stopping a process)

  2. `SIG_DFL`: let the default action happens (in most cases terminate process or terminate process with a core dump; in some cases ignore)

  3. address of the user-defined handler: the Kernel catches it by invoking this function

  - `SIGKILL` and `SIGSTOP` cannot be caught

  - a signal handler can return or call exit; if it returns the normal sequence of instructions continues executing

POSIX.1 reliable-signal

- signal dispositions
  1. `SIG_IGN`: ignore the signal (except `SIGKILL` and `SIGSTOP` which can never be ignored as they are surefire ways of stopping a process)
  2. `SIG_DFL`: let the default action happens (in most cases terminate process or terminate process with a core dump; in some cases ignore)
  3. address of the user-defined handler: the Kernel catches it by invoking this function
     - `SIGKILL` and `SIGSTOP` cannot be caught
     - a signal handler can return or call exit; if it returns the normal sequence of instructions continues executing
- system calls can be interrupted by a signal
  - some of them are automatically restarted, e.g., `ioctl`, `read`, `readv`, `write`, `writev`, `wait`, and `waitpid`

Installing a handler for a signal

```
sighandler_t signal(int signum, sighandler_t handler);
```

sets the disposition of the specified signal

Installing a handler for a signal

```
sighandler_t signal(int signum, sighandler_t handler);
```

sets the disposition of the specified signal

- a signal handler is a function that takes a single integer argument and returns nothing
- we can pass SIG_IGN or SIG_DFL instead of a handler
- `signal` returns a pointer to the previous disposition of signal or `SIG_ERR` on error

Installing a handler for a signal

```
sighandler_t signal(int signum, sighandler_t handler);
```

sets the disposition of the specified signal

- a signal handler is a function that takes a single integer argument and returns nothing
- we can pass SIG_IGN or SIG_DFL instead of a handler
- `signal` returns a pointer to the previous disposition of signal or `SIG_ERR` on error
- **portability problem**: behaviour varies across UNIX versions (use `sigaction` instead)

Installing a handler for a signal

```
sighandler_t signal(int signum, sighandler_t handler);
```

sets the disposition of the specified signal

- a signal handler is a function that takes a single integer argument and returns nothing
- we can pass SIG_IGN or SIG_DFL instead of a handler
- `signal` returns a pointer to the previous disposition of signal or `SIG_ERR` on error
- **portability problem**: behaviour varies across UNIX versions (use `sigaction` instead)

```
#include <stdio.h>, <unistd.h>, <signal.h>
int  i;
void quit(int code) {
        fprintf(stderr, "\nInterrupt (code= %d, i= %d)\n", code, i);
}
int main (void) {
       if(signal(SIGQUIT, quit) == SIG_ERR)
              perror(“can’t catch SIGQUIT”);
       for (i= 0; i < 9e7; i++)
              if (i % 10000 == 0) putc('.', stderr);
       return(0);
}
```

POSIX signal environment

- examining or modifying the action associated with a particular signal (except for `SIGKILL` and `SIGSTOP`)
  - `sigaction( )`
  - this function supersedes the `signal( )` function from earlier releases of the UNIX System

POSIX signal environment

- examining or modifying the action associated with a particular signal (except for `SIGKILL` and `SIGSTOP`)
  - `sigaction( )`
  - this function supersedes the `signal( )` function from earlier releases of the UNIX System
- blocking signals in a signal set from delivery to a process
  - `sigprocmask( )`

POSIX signal environment

- examining or modifying the action associated with a particular signal (except for `SIGKILL` and `SIGSTOP`)
  - `sigaction( )`
  - this function supersedes the `signal( )` function from earlier releases of the UNIX System
- blocking signals in a signal set from delivery to a process
  - `sigprocmask( )`
- manipulating the signal set (a bit vector)
  - `sigemptyset( ), sigfillset( ) , sigaddset( ) , sigdelset( ), sigismember( )`
  - when a signal is caught and the handler is entered, the current signal is automatically added to the signal mask of the process; this is to prevent subsequent occurrences of that signal from interrupting the signal handler

POSIX signal environment

- examining or modifying the action associated with a particular signal (except for `SIGKILL` and `SIGSTOP`)
  - `sigaction( )`
  - this function supersedes the `signal( )` function from earlier releases of the UNIX System
- blocking signals in a signal set from delivery to a process
  - `sigprocmask( )`
- manipulating the signal set (a bit vector)
  - `sigemptyset( ), sigfillset( ) , sigaddset( ) , sigdelset( ), sigismember( )`
  - when a signal is caught and the handler is entered, the current signal is automatically added to the signal mask of the process; this is to prevent subsequent occurrences of that signal from interrupting the signal handler
- returning the set of signals that are blocked from delivery and currently pending for the calling process
  - `sigpending( )`

Example of the sigaction( ) system call

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

void signal_callback_handler(int signum) {
   printf("Caught signal!\n");

   // uncomment the next line to break the loop when signal is received
   // exit(1);
}

int main() {
   struct sigaction sa;
   sa.sa_flags = 0;
   sigemptyset(&sa.sa_mask);
   sa.sa_handler = signal_callback_handler;
   sigaction(SIGINT, &sa, NULL);  // we are not interested in the old disposition

   // sigaction(SIGTSTP, &sa, NULL);
   while (1) {}
}
```

Example of the sigaction( ) system call

```
struct sigaction {
                void     (*sa_handler)(int);
                void     (*sa_sigaction)(int, siginfo_t *, void *);
                sigset_t   sa_mask;
                int        sa_flags;
                void     (*sa_restorer)(void);
};
```

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

void signal_callback_handler(int signum) {
   printf("Caught signal!\n");

   // uncomment the next line to break the loop when signal is received
   // exit(1);
}

int main() {
   struct sigaction sa;
   sa.sa_flags = 0;
   sigemptyset(&sa.sa_mask);
   sa.sa_handler = signal_callback_handler;
   sigaction(SIGINT, &sa, NULL);  // we are not interested in the old disposition

   // sigaction(SIGTSTP, &sa, NULL);
   while (1) {}
}
```

Process creation and signals

- when a process calls `fork`
  - the child inherits the parent's signal dispositions and signal mask
  - the child starts off with a copy of the parent's memory image, so the signal-handler is defined in the child

Process creation and signals

- when a process calls `fork`
  - the child inherits the parent's signal dispositions and signal mask
  - the child starts off with a copy of the parent's memory image, so the signal-handler is defined in the child
- when a process calls `exec`
  - the disposition of any signals being caught (not ignored) are changed to their default action in the new program
  - the status of all other signals is left alone
  - the signal mask is preserved across the exec system call

Interprocess Communication

Cooperating processes

- any two processes are either independent or cooperating

Cooperating processes

- any two processes are either independent or cooperating
- cooperating processes work with each other to accomplish a single task
  - improve performance
    - overlapping activities or performing work in parallel
  - improve program structure
    - each cooperating process is smaller than a single monolithic program

Cooperating processes

- any two processes are either independent or cooperating
- cooperating processes work with each other to accomplish a single task
  - improve performance
    - overlapping activities or performing work in parallel
  - improve program structure
    - each cooperating process is smaller than a single monolithic program
- they may need to share information
  - OS makes this happen!

Homework

- See the examples posted on eClass