Unix Signals (CSE 422S)

Ken Wong
Washington University

kenw@wustl.edu
www.arl.wustl.edu/~kenw

The Signal Concept

- Signals are asynchronous event notifications
  » Like software interrupt but sent by OS to process

- Every signal has a name and a positive integer
  » See signal(7)
  » <signal.h>: Mapping between names and numbers
    - See /usr/include/bits/signum.h (e.g., SIGKILL → 9)

Generating Signals

- Command Line: kill -KILL 8371
  - Unconditionally kill process 8371
  - Equivalent to 'kill -9 8371'

- Keyboard: ctrl-c
  - Interactive interrupt key
  - Equivalent to sending SIGINT signal to shell (and its proc group)

- Program: if (kill(8371, SIGTERM) == -1) . . .

Terminology

- A signal is generated by certain events
  » Hardware exception (divide by 0)
  » A software condition becomes true (alarm timer expires)
  » A terminal generates a signal (kill command)
  » The kill(2) system call
  » OS Kernel sets a flag in the process table for the signal

- Signal is delivered to a process when signal action is taken

- A signal is pending if it is generated, but not delivered

- A process can:
  » Block the delivery of a signal or
  » Ignore the signal (throw it away)

- If a signal is generated more than once while blocked, the user can have:
  » One delivery (typical case) or
  » Many deliveries (i.e., queue the signals)

- The process signal mask indicates the blocked signals

Some Signals and Default Actions

Abnormal Termination

- SIGINT: User presses interrupt (usually ctrl-c) key
- SIGPIPE: Process writes to a pipe after reader has quit
- SIGALRM: Alarm clock expires
- SIGTERM: Terminate (kill) process
- SIGHUP: Terminate (kill) process (Can’t be caught/ignored)

Stop Process

- SIGTSTP: User presses suspend (ctrl-z) key
- SIGSTOP: Stop process (Can’t be caught/ignored)
  » SIGCONT continues process
- SIGTTIN: Background process attempts to read from controlling terminal

Implementation Dependent

- SIGCHLD: Child process terminates
- SIGQUIT: User presses quit (ctrl-1) key ➔ produces core dump

- See signal(7) for list of signals and their values
Handling A Signal

- Semantics: When signal X occurs, do Y.
  - Y is called the disposition or action

- Action Choices
  - **Ignore** the signal
    - Works for most signals
    - SIGKILL and SIGSTOP cannot be ignored
  - **Catch** the signal
    - Call a signal handler (user-written function)
    - Let the default action occur
      - Every signal has a default action
      - In most cases, terminate the process

Concepts

- **Signal Mask**
  - Indicates the set of signals which should be blocked
    - Blocking means hold for later delivery (different from ignore)
  - Type is sigset_t
  - Manipulated by five functions
    - sigaddset, sigdelset, sigemptyset, sigfillset, sigismember
  - sigprocmask used to read/write a process’s signal mask

- **Signal handler**
  - The function that is called when a signal is caught

- **Signal action**
  - Action associated with a signal
  - sigaction is used to examine/specify action for a signal
  - Do NOT use signal (unreliable signals)

Code Example 1 (1)

```c
static void handle_alarm(int signo) {
    ... code to handle alarm ...
    return; // or exit
}
```

```c
int main(int argc, char *argv[]) {
    struct sigaction action, oaction;
    sigemptyset(&action.sa_mask); // additional signals
    action.sa_flags = 0; // no special options
    action.sa_handler = handle_alarm; // the handler code
    if (sigaction(SIGALRM, &action, &oaction) < 0) Error...
    for (int i=0; i<5; i++) {
        alarm(2); // send SIGALRM in 2 sec
        pause(); // suspend until get signal
        printf("I woke up!\n");
    }
    return 0;
}
```

Code Example 1 (2)

```c
int sigaction(int sig, const struct sigaction *act, struct sigaction *oact)
```

- Changes the action taken by a process for a signal
  - sig: Any signal except SIGKILL and SIGSTOP
  - act: SIG_IGN or SIG_DFL or ptr to handler function
  - oact: Where to save old action if non-null
  - Return 0 if successful; -1 otherwise

- sigaction structure

```c
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask; // additional signals to block
    int sa_flags;
    void (*sa_restorer)(void); // OBSOLETE
}
```
**Signal Sets**

- A **signal set** is a data type (sigset_t) used to represent multiple signals
  - Each signal is represented by a single bit
  - Solaris uses 128 bits
  - Linux uses 1024 bits
- Initialize signal set by first turning all bits either off or on (use `sigemptyset` or `sigfillset`)
- Then add and delete specific signals in the set
  - Use `sigaddset` or `sigdelset`

**Code Example 1 (3)**

```c
struct sigaction {
    void (*sa_handler)(...);
    void (*sa_sigaction)(...);
    sigset_t sa_mask;
    int sa_flags;
    ... }
```

**Signal Related Functions**

- **kill**: Send a signal to a process
- **raise**: Send a signal to itself
- **alarm**: Set a timer that will expire in the future
  - Generate SIGALRM when timer expires
- **pause**: Suspend calling process until a signal is delivered to process
- **sigemptyset**: Exclude all signals in signal set
- **sigismember**: Return 1 if a signal is in a signal set
- **sigprocmask**: Examine/modify which signals are blocked from deliver to a process
- **sigpending**: Return blocked/pending signal set
- **sigaction**: Examine/modify action for a signal

**Difficulties With Handling Signals**

- Whether POSIX functions that are interrupted by signals should be restarted
  - What happens when a process catches a signal while it is executing a library function?
  - Slow versus fast I/O
- Signal handlers calling **non-reentrant functions**
  - Any function that changes the value of a static variable, use malloc/free, or use a global data structure
- Handling errors that use the system global variable errno
- **Async-Safe Function**
  - Problem is that a function may have to wait for a signal handler to complete before it completes
  - Predictable results when called from within a signal handler
**Signal Interrupts System Call**

- Signal handling depends on how fast the call is
  - Terminal I/O is slow (may block for a long time)
  - Disk I/O is fast (may block for a short time)
  - Some calls (e.g., getpid) do not block at all
- Slow POSIX calls are interrupted by signals
  - They return after the signal handler returns
  - Return code is -1 with \( \text{errno} = \text{EINTR} \)
  - No clear way to determine which functions can get interrupted except by looking for \( \text{errno} = \text{EINTR} \) in man page of the function !!!
  - Be careful when calling functions from inside signal handlers !!!
    - fprintf is NOT async-safe (but usually OK for simple testing)

**Handling errno**

- A function sets \( \text{errno} \) because of an error but a signal handler is called before the error message is printed
  - The signal handler could change \( \text{errno} \) because of an error
  - It should restore the proper value of \( \text{errno} \) before returning

```c
void myhandler(int signum) {
  int old_errno;
  old_errno = errno;
  ... Do Something ...
  errno = old_errno;
}
```

**Avoiding Common Mistakes**

- Use \texttt{sigaction}, NOT \texttt{signal}, system call
  - But OK if OS redefines \texttt{signal} as \texttt{sigaction}
- Avoid serious race conditions
  - A signal handler is an asynchronous thread of control
  - Use signal mask to block unwanted interruptions of a handler
- Make signal handlers do little work
  - If event requires a lot of processing
    - Raise a flag in the handler
    - Have the normal code path check the flag and do the work
- Avoid calling functions that are not \texttt{async-signal safe} inside signal handlers if it can be interrupted by a signal

**Siglongjmp and Sigsetjmp Example**

```c
static sigjmp_buf jmpbuf;
static volatile sig_atomic_t jumpok = 0;

static void handler(int signo) {
  if (jumpok == 0) return;
  siglongjmp(jmpbuf, 1);
}

int main(void) {
  struct sigaction act;
  act.sa_flags = 0;
  act.sa_handler = handler;
  if ( (sigemptyset(&act.sa_mask) == -1) ||
       sigsetjmp(jmpbuf, 1) ) ... Error/Exit ...
  sigsetjmp(jmpbuf, 1); // return here from handler
  jumpok = 1;
  for ( ; ; ) { ... Main Loop ... }
}
```
Siglongjmp and Sigsetjmp

- **sigsetjmp** is analogous to a statement label
  - Like placing a marker at the current location
  - `int sigsetjmp(sigjmp_buf env, int savemask);`
    - `env` is initialized with information needed to jump back to the current location
    - Save the current signal mask in the `env` buffer if `savemask` is nonzero
    - Return value is 0 when directly called
    - Return value is `val` argument of `siglongjmp(env, val)` otherwise

- **siglongjmp** is analogous to a goto statement
  - `void siglongjmp(sigjmp_buf env, int val);`