UNIX Network Programming

Asynchronous Programming with Signals

Douglas C. Schmidt

Overview of Signals

- Signals are "software interrupts"
  - *i.e.*, they work similarly to hardware interrupts
- Signals provide a means for asynchronous programming in single-threaded applications
  - Therefore, they introduce issues related to "concurrency control"
- Signals are generally used as latches, *not* queues
  - Moreover, there are no multi-level signal priorities

Signals as Virtual Machine

- Signals can provide a "software virtual machine"

Hardware Machine	Software Virtual Machine
instruction set	set of system calls
restartable instructions	restartable system calls
interrupts/traps	signals
interrupt/trap handlers	signal handlers
blocking interrupts	masking signals
interrupt stack	signal stack

Two Types of Signals

- Signals may be used:

1. *Synchronously*
   - To signal errors
     - ▷ *e.g.*, SIGFPE, SIGSEGV
   - Always delivered to the thread that incurs the error
2. *Asynchronously*
   - To notify a process that an event has occurred
     - ▷ *e.g.*, SIGINT, SIGIO, SIGALRM
   - Can be delivered to any thread willing to handle the signal

- Signal 0 is used to determine if a specified process is "alive"

. *i.e.*, kill (pid, 0);

Example of kill() System Call

- Example: check whether a process exists using the `kill` system call:

```
#include <stdio.h>
#include <errno.h>
int main (int argc, char *argv[]) {
    if (argc != 2)
        fprintf (stderr, "usage: %s pid\n", argv[0]),
        exit (1);
    else {
        pid_t pid = atoi (argv[1]);
        if (pid <= 0)
            fprintf (stderr, "warning, broadcasting...\n");
        printf ("process %d does%s exist\n",
            pid, kill (pid, 0) == -1 &&
            errno == ESRCH ? " not" : "");
    }
    return 0;
}
```

- This program checks if a process exists, even if it is not the owner of the process

Reliable Signals

- Signals can be delivered *reliably*, *e.g.*,
  1. By one process to another (or to itself)
  2. By the kernel to a process
- Beware of portability problems, however...
  1. Older versions of UNIX have "unreliable" signal semantics
     - *e.g.*, signals might get lost or cause improper failures or race conditions
  2. Newer versions of UNIX have a wide variety of reliable and unreliable signal facilities
     - Diversity is necessary to support backwards compatibility

Conditions That Generate Signals

- (1) Typing certain characters at the terminal:
  - ^C – interrupt character (SIGINT)
  - ^\ – quit character (SIGQUIT)
- (2) Occurrence of certain hardware conditions:
  - SIGFPE – for floating point arithmetic errors
  - SIGSEGV – referencing an illegal or invalid address
- (3) Occurrence of certain software conditions:
  - SIGCLD – indicates death of a child process
  - SIGALRM – software alarm clock

Conditions That Generate Signals (cont'd)

- (4) Using the `kill` command at the shell level, *e.g.*,

```
% kill -9 12345
# -9 (SIGKILL) cannot be caught or ignored
```

- (5) Using the `kill` or `killpg` system calls to send a signal to a process or process group
  - *e.g.*,

    ```
    kill (int pid, int signo);
    killpg (int pgrp, int signo);
    ```

  - Note, the sending and receiving processes must have the same effective userid
    - ▹ Unless the process is the superuser

Standard Signals

- The following are a list of standard UNIX signals, divided up into categories (see <signal.h> for more details)

  – Hardware exceptions (handled as "traps")

```
SIGFPE    8   // floating point exception
SIGILL    4   // illegal instruction (not reset when caught)
SIGBUS    10  // bus error
SIGSEGV   11  // segmentation violation
SIGSYS    12  // bad argument to system call
```

  – Timed events

```
SIGALRM   14  // alarm clock
SIGVTALRM 26  // virtual time alarm
SIGPROF   27  // profiling time alarm
```


Signals (cont'd)

- Signal categories (cont'd)

  – User requests and job control

```
SIGINT    2   // interrupt
SIGQUIT   3   // quit
SIGHUP    1   // hangup
SIGTERM   15  // software termination signal from kill
SIGKILL   9   // kill (cannot be caught or ignored)
SIGTSTP   18  // stop signal from tty
SIGSTOP   17  // sendable stop signal not from tty
SIGCONT   19  // continue a stopped process
```

  – Input/Output after user request

```
SIGIO     23  // Asynchronous input/output possible signal
SIGPOLL   23  // System V version of "SIGIO"
SIGURG    16  // urgent condition on IO channel
SIGPIPE   13  // write on a pipe with no one to read it
SIGTTIN   21  // to readers pgrp upon background tty read
SIGTTOU   22  // like TTIN for output if (tp->t_local&LTOSTOP)
```


Signals (cont'd)

- Signal categories (cont'd)

  – Exceeded limits

```
SIGXCPU  24   // exceeded CPU time limit
SIGXFSZ  25   // exceeded file size limit
```

  – Window changed or data lost

```
SIGWINCH 28   // window changed
SIGLOST  29   // resource lost (e.g., record-lock lost)
```

  – User defined signals

```
SIGUSR1  30   // user defined signal 1
SIGUSR2  31   // user defined signal 2
```


Signals Handling Actions

- Actions taken when a process receives a signal:

  – *Kernel Actions*

    ▹ The signal is blocked from further occurrence (other signals are not, however)

      · *i.e.*, signals do not queue by default...

    ▹ The current process state is saved

    ▹ A new "context" is built to handle the signal

  – *User Actions* depend on signal disposition

  1. SIG_IGN: ignore it
  2. SIG_DFL: take the "default action"
     ▹ *e.g.*, usually terminate the process and dump core
  3. Call a user-supplied function (a.k.a. "signal handler")
     ▹ Be careful of re-entrancy problems, race conditions, and deadlock

Signal Handling APIs

- Depending on which version of UNIX is being used, signal handlers may be set via

1. `signal`
   - Original API
2. `sigset`
   - SVR2
3. `sigvec`
   - BSD 4.3
4. `sigaction`
   - SVR4

- In general, use of `sigaction` is preferred if backwards compability is not an issue...

Example of signal()

- `signal` sets the handler for `sig` to `func` and returns old handler

void (*signal (**int** sig, **void** (*func)(**int**)))(**int**);

- *e.g.*,

```
int global = 0;
void catcher1 (int) { global = global + 10; }
void catcher2 (int) { global = global − 12; }
int main (void) {
      signal (SIGINT, catcher1);
      signal (SIGQUIT, catcher2);
      ...
      for (;;) {
            sigpause (0);
            if (global != 0) // signal occurred
            ...
      }
}
```


The signal() Interface (cont'd)

- *Problems*
  - No way to protect critical sections of data shared between `catcher1`, `catcher2`, and `main`
  - The original semantics of catching a signal set the disposition back to system default
    - ▷ *i.e.*, SIG_DFL or SIG_IGN
  - Therefore, if a second signal occurred whilst in the handler the process might terminate...
- To solve these problems, a series of extensions were made to different variants of UNIX

The sigvec() Interface (cont'd)

- `sigvec` is a more general interface
  - It solves reliability problems by allowing the specification of a set of signals that are atomically deferred upon a catcher's invocation
    - ▷ By default, current signal is deferred automatically
  - These signals are atomically released upon return from the catcher
  - *e.g.*,

```
struct sigvec {
      void (*sv_handler)(); // signal handler
      int sv_mask; // signal mask to apply
      int sv_flags; // see signal options
};
int sigvec (int sig, struct sigvec *vec,
      struct sigvec *old_vec);
```

- *Problem*
  - Number of signals is limited to 32 bits...

Example of sigvec()

- *e.g.*,

```
int global = 0;
void catcher1 (int) { global = global + 10; }
void catcher2 (int) { global = global - 12; }
int main (void) {
      struct sigvec vec;
      int omask;

      vec.sv_handler = catcher1;
      vec.sv_mask = sigmask (SIGQUIT);
      vec.sv_flags = 0;
      sigvec (SIGINT, &vec, 0);
      vec.sv_handler = catcher2;
      vec.sv_mask = sigmask (SIGINT);
      vec.sv_flags = 0;
      sigvec (SIGQUIT, &vec, 0);
      ...
      omask = sigblock (sigmask (SIGINT)
                  | sigmask (SIGQUIT));
      // examine global data atomically
      sigsetmask (omask);
}
```


The sigaction() Interface

- The POSIX `sigaction` interface was adopted by SVR4 UNIX
  - It utilizes an ADT `sigset_t` to overcome the 32 bit limit of the BSD `sigvec` interface
  - In addition, the `sigaction` API also supports backwards compatibility with old-style signal handlers
  - *e.g.*,

```
struct sigaction {
      void (*sa_handler)(); // signal handler
      sigset_t sa_mask; // signal mask to apply
      int sa_flags; // SA_RESTART
                  // SA_SIGINFO
};
```


The sigaction() Interface (cont'd)

- **int** sigaction (**int** sig, **struct** sigaction *vec, **struct** sigaction *old_vec);
  - Set and/or get disposition of a signal
  - Subsumes `signal`, `sigset`, `sigvec` calls
- **int** sigsuspend (sigset_t *set);
  - Automatically replace the set of deferred signals and pause until a signal is received
  - Returns only after signal-catching function returns
  - Subsumes `sigpause`
- **int** sigpending (sigset_t *set);
  - Returns the set of deferred signals

The sigaction() Interface (cont'd)

- **int** sigprocmask (int flag, sigset_t *set, sigset_t *oset);
  - Set and/or get the current deferred signal set
  - Subsumes earlier APIs
  - `flag` enables
    1. replacing (SIG_SETMASK)
    2. or'ing (SIG_BLOCK)
    3. and'ing with complement of `set` (SIG_UNBLOCK)

The sigset_t Operations

- **typedef struct** { **unsigned long** sigbits[/* impl dependent */]; } sigset_t;
- **int** sigemptyset (sigset_t *set);
  - Create an empty set of signals
- **int** sigfillset (sigset_t *set);
  - Create the set containing every signal the implementation supports
- **int** sigaddset (sigset_t *set, **int** sigio);
  - Add a signal to a set
- **int** sigdelset (sigset_t *set, **int** signo);
  - Delete a signal from a set
- **int** sigismember (sigset_t *set, **int** signo);
  - Test for membership of a signal in a set

Example of sigaction()

- *e.g.*,

```
int global = 0;
void catcher1 (int) { global = global + 10; }
void catcher2 (int) { global = global - 12; }
int main (void) {
	struct sigaction vec;
	sigset_t mask, omask;

	vec.sa_handler = catcher1;
	vec.sa_flags = 0;
	sigemptyset (&vec.sa_mask);
	sigaddset (&vec.sa_mask, SIGQUIT);
	sigaction (SIGINT, &vec, 0);
	vec.sa_handler = catcher2;
	vec.sa_flags = 0;
	sigemptyset (&vec.sa_mask);
	sigaddset (&vec.sa_mask, SIGINT);
	sigaction (SIGQUIT, &vec, 0);
	...
	sigemptyset (&mask);
	sigaddset (&mask, SIGINT);
	sigaddset (&mask, SIGQUIT);
	sigprocmask (SIG_BLOCK, &mask, &omask);
	// examine global data atomically
	sigprocmask (SIG_SETMASK, &omask, 0);
}
```


Complex Signal Example

- The following example illustrates signals, `fork`, `setjmp`, `longjmp`, and `kill`
- Here's the output:

```
% a.out
in child process 23727
in parent process 23726
I am [schmidt]
^Cdo you wish to quit? [y/n]..: n
coming back from longjump!
I am [schmidt]
^\received SIGQUIT in process 23727
^\received SIGQUIT in process 23727
^Cdo you wish to quit? [y/n]..: n
coming back from longjump!
I am [schmidt]
^Cdo you wish to quit? [y/n]..:
timed out, received SIGTERM in process 23727, exiting
```


Complex Signal Example (cont'd)

- signal-test.c

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

const int TIME_OUT_SECS = 10;
jmp_buf begin;
pid_t childpid; // Must be global.
pid_t pid;

// Forward decls.
void SIGQUIT_handler (int), SIGINT_handler (int);
void SIGTERM_handler (int), SIGALRM_handler (int);
void do_child (void);
int do_parent (void);

int main (int argc, char *argv[]) {
	switch (childpid = fork ()) {
	case -1: perror ("main::fork"), exit (1);
	case 0:
		do_child ();
		// NOTREACHED
	default:
		if (do_parent () < 0)
			perror (argv[0]), exit (1);
	}
	return 0;
}
```

Complex Signal Example (cont'd)

- signal-test.c

```
void do_child (void) {
        pid = getpid ();
        signal (SIGQUIT, SIGQUIT_handler);
        signal (SIGINT, SIGINT_handler);
        signal (SIGTERM, SIGTERM_handler);
        fprintf (stderr, "in child proc %d\n", pid);

        if (setjmp (begin) != 0)
                fprintf (stderr, "back from longjump!\n");

        fprintf (stderr, "I am [%s]\n", getlogin ());

        for (;;)
                sigpause (0);
                // try hitting ^C or ^\ here!
}

int do_parent (void) {
        pid = getpid ();
        signal (SIGINT, SIG_IGN);
        signal (SIGQUIT, SIG_IGN);
        signal (SIGALRM, SIGALRM_handler);
        fprintf (stderr, "in parent proc %d\n", pid);
        alarm (TIME_OUT_SECS);
        return wait ((int *) 0);
}
```


Complex Signal Example (cont'd)

- signal-test.c

```
#if defined (DEBUG)
#define D(X) do { X } while (0)
#else
#define D(X)
#endif

void SIGALRM_handler (int sig) {
        D (psignal (sig, "signal occurring was"););
        if (kill (childpid, SIGTERM) == −1)
                perror ("signal handler");
}
void SIGTERM_handler (int sig) {
        D (psignal (sig, "signal occurring was"););
        fprintf (stderr, "time out, in proc %d\n", pid);
        exit (1);
}
void SIGINT_handler (int sig) {
        char b[3];
        D (psignal (sig, "signal occurring was"););
        do
                fputs ("do you wish to quit? [y/n]..: ", stderr);
        while (*fgets (b, sizeof b, stdin) != 'y' && *b != 'n');
        if (*b == 'y')
                exit (1);
        else
                longjmp (begin, 1);
}
void SIGQUIT_handler (int sig) {
        D (psignal (sig, "signal occurring was"););
        fprintf (stderr, "quit proc %d\n", pid);
}
```


Signal Chaining

- The various `signal` APIs all return the disposition that currently exists
- This is useful to implement "signal chaining", *e.g.*,

```
typedef void (*SignalHandler) (...);
SignalHandler f;
f = signal (SIGINT, (SignalHandler) foo);
// ... Critical Section ...
signal (SIGINT, (SignalHandler) f);
```

- In general, don't use `signal`, use `sigaction`, however!

Slow System Calls and Signals

- "Slow" system calls are usually those that can block while reading from devices that have long latency
  - *e.g.*, terminals or network devices
    - ▷ But not disk drives!
- The slow system calls include `flock`, `ioctl`, `read`, `readv`, `wait`, `write`, `writev`, `accept`, `connect`
- Slow system calls are normally interrupted when a signal occurs
  - The system call returns −1
  - `errno` is set to EINTR
    - ▷ See /usr/include/sys/errno.h

Slow System Calls and Signals (cont'd)

- A process that handles signals (or spawns children) should test whether an error return == −1 and `errno` == EINTR, and then reissue the slow system call, if desired, *e.g.*,

```
while ((nfd = accept (fd, 0, 0)) == -1 &&
       errno == EINTR)
       continue;
```

- In SVR4 all system calls may be flagged as restarted via the `sigvec` or `sigaction` system calls
  - *i.e.*, add the SA_RESTART flag to `sa_flags`

Summary

- Signals provide a means for asynchronous programming in single-threaded applications

- Signals are not particularly useful as a general-purpose IPC mechanism, *e.g.*,

1. They are not commonly used to deliver data
2. The identity of the sender is not generally available
3. Signal semantics are subtle and error-prone

- It is often better to build applications using higher-level programming models such as processes and threads