Signals and signal processing (2)

Suntae Hwang

Kookmin University

Unreliable signals (1/2)

- Signals were unreliable in earlier version of UNIX (V7)
  - Signals could get lost
    - a signal could occur and the process would never know about it
  - The action for the signal was reset to its default each time the signal occurred
  - A process had a little control over a signal
    - can catch or ignore the signal, but can't *block* it
    - unable to turn a signal off when it didn't want the signal to occur (all it can do was ignore the signal)
    - can't "prevent the following signals from occurring, but only *remember* if they do occur".

Unreliable signals(2/2)

```
int     sig_int();    /*my signal handling function*/
…
signal(SIGINT, sig_int); /*establish handler*/
…
sig_int()
{
   signal(SIGINT, sig_int);
        /*reestablish handler for next occurrence*/
    …          /*process the signal*/
}
```

```
/* set the flag to remember that an interrupt occurs */
int      sig_int_flag;
main() {
    int  sig_int();
    …
    signal(SIGINT,sig_int);
    …
    while(sig_int_flag==0)
          pause();  /* go to sleep, wating for signal */
     …
}
sig_int() {
    signal(SIGINT,sig_int);
    sig_int_flag=1; /*set flag for main loop to
 examine*/
}
```

Q: What happens if the interrupt signal occurs again before singal() in sig_int() executes?

Q: What happens if the interrupt signal occurs again before pause() and after while statement?

Reliable Signal Terminology and Semantics

- A signal is *generated* for a process (or sent to a process) when the event that causes the signal occurs
  - when signal is generated the kernel usually sets a flag of some form in the process table
- A signal is *delivered* to a process when the action for a signal is taken
- During the time between the generation of signal and its delivery, the signal is said to be *pending.*
- Each process has a *signal mask* that defines the set of signals currently *blocked* from delivery to that process

Reliable Signal Terminology and Semantics

- *blocking* the delivery of a signal
  - A blocked signal (with default or catch signal action) remains *pending* until
    - unblocks the signal
    - changes the action to ignore the signal
  - What to do with a blocked signal is determined when the signal is *delivered*, not when it is generated
    - This allows the process to change the action for the signal before it is delivered
  - Most Unix does not queue blocked signals generated more than once; the Unix kernel just delivers the signal once.

Signal sets

- A data type to represent multiple signals (sigset_t)
- Do not assume global/static variable initialization in C for sigset_t
- Functions to manipulate signal sets

```
#include <signal.h>
int sigemptyset (sigset_t *set);
int sigfillset (sigset_t *set);
int sigaddset (sigset_t *set, int signo);
int sigdelset (sigset_t *set, int signo);
                                    all four return: 0 if OK, -1 on error
int sigismember ( const sigset_t *set, int signo);
                                    Returns:1 if true, 0 if false
```

Signal Functions

- Superset of the functionality of signal()
- sigprocmask(int how, const sigset_t * setp, sigset_t *osetp);
  - examine or change signal masks
- sigpending(sigset_t* setp);
  - Return the set of signals that are blocked and pending
- sigaction(int signo, const struct sigaction* act, struct sigaction oact);
  - examine or modify the action associated with a particular signal
- sigsuspend(const sigset_t* sigmask);
  - atomic operation (reset the signal mask and pause)
- sigsetjmp(sigjmp_buf env, int savemask)/siglongjmp(sigjmp_buf env, init val)
  - nonlocal branching from a signal handler

Sigprocmask function

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```
Returns: 0 if OK, -1 on error

- Examine or change the signal mask *that is the set of signals currently blocked from delivery to the process*
  - First, if *oset* is nonnull pointer, the current signal mask for the process is returned through oset
  - Second, if *set* is a nonnull pointer, then the *how* argument indicates how the current signal mask is modified
  - if *set* is NULL, *how* is not significant
  - *how*
    - SIG_BLOCK : redefine the new signal mask (*set* + *oset*)
    - SIG_UNBLOCK : unblock the signals in *set*
    - SIG_SETMASK : the new signal mask = *set*

- if there are any pending, unblocked signals after `sigprocmask` at least one of these signal is delivered to the process before `sigprocmask` returns

Sigprocmask Function Example

```
#include <errno.h>
#include <signal.h>
#include "ourhdr.h"

void
pr_mask(const char *str)
{
        sigset_t sigset;
        int      errno_save;

        errno_save = errno;       /* we can be called by
signal handlers */
        if (sigprocmask(0, NULL, &sigset) < 0)
                err_sys("sigprocmask error");

        printf("%s", str);
        if (sigismember(&sigset, SIGINT)) printf("SIGINT ");
        if (sigismember(&sigset, SIGQUIT))printf("SIGQUIT ");
        if (sigismember(&sigset, SIGUSR1))printf("SIGUSR1 ");
        if (sigismember(&sigset, SIGALRM))printf("SIGALRM ");
                /* remaining signals can go here */
        printf("\n");
        errno = errno_save;
}
```

Sigpending Function

```
#include <signal.h>
int sigpending (sigset_t *set);
                                Returns: 0 if OK, -1 on error
```

- Returns the set of signals that are blocked from delivery and currently pending for the calling process
  - The set of signals is returned through the *set* argument

Sigpending Function

```
static void      sig_quit(int);
int main(void) {
        sigset_t newmask, oldmask, pendmask;

        if (signal(SIGQUIT, sig_quit) == SIG_ERR)
                err_sys("can't catch SIGQUIT");
        sigemptyset(&newmask); sigaddset(&newmask, SIGQUIT);

        /* block SIGQUIT and save current signal mask */
        if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
                err_sys("SIG_BLOCK error");
        sleep(5);                  /* SIGQUIT here will remain pending */

        if (sigpending(&pendmask) < 0)
                err_sys("sigpending error");
        if (sigismember(&pendmask, SIGQUIT))
                printf("\nSIGQUIT pending\n");

        /* reset signal mask which unblocks SIGQUIT */
        if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
                err_sys("SIG_SETMASK error");
        printf("SIGQUIT unblocked\n");
        sleep(5);          /* SIGQUIT here will terminate with core file */
        exit(0);
}
```

Sigpending Function

```
static void
sig_quit(int signo)
{
        printf("caught SIGQUIT\n");

        if (signal(SIGQUIT, SIG_DFL) == SIG_ERR)
                err_sys("can't reset SIGQUIT");
        return;
}
```

```
$ a.out
^\^\^\^\                generate signal several times(before
                         5 seconds are up)
SIGQUIT pending          after return from sleep(5)
caught SIGQUIT           in signal handler (signal is generated once!)
SIGQUIT unblocked        after return from sigprocmask
^\Quit(coredump)         generate signal again (message by shell)
```

Sigaction Function (1/3)

```
#include <signal.h>
int sigaction (int signo, const struct sigaction *act,
                          struct sigaction *oact);
                          Returns:0 if OK, -1 on error
```

- Examine or modify the action associated with a particular signal - supersedes the signal function from earlier UNIX
  - If act pointer is nonnull, we are modifying the action.
  - If oact pointer is nonnull, the system returns the previous action for the signal.
- Specify a set of signals added to the signal mask before the signal handler is called - *includes the signal being delivered*.
  - This way we are able to *block* certain signals whenever a signal handler invoked
  - When the signal-catching function returns, the signal mask of process is *reset* to its previous value
- The action remains installed until we explicitly change it by calling sigaction().

Sigaction Function (2/3)

```
struct sigaction {
     void       (*sa_handler)(int);          /* address of signal handler */
     sigset_t    sa_mask;                    /* additional signals to block */
     int         sa_flags;                   /* signal options */
     void (*sa_sigaction)(int, siginfo_t *, void *);
};
```

- *sa_handler*: points to the signal hanlder or SIG_IGN or SIG_DFL
- sa_mask: additional signals to block when the signal handler (as opposed to SIG_IGN or SIG_DFL) is called
- sa_flags : specifies various options for the handling of the signal
  - SA_RESTART - system calls interrupted are automatically restarted
  - SA_NODEFER - when this signal is caught, the signal is not automatically blocked by the system while the signal handler executes (unreliable signal)
  - SA_RESETHAND - the disposition for the signal is reset to SIG_DFL on entry to the signal handler (unreliable signal)
  - SA_SIGINFO – provides additional information to a signal handler. Final `sig_action` field is used

Sigaction Function (3/3)

```
/* An implementation of reliable signal() using sigaction */
/* for unreliable signal(), use SA_RESETHAND and SA_NODEFER */

#include <signal.h>

Sigfunc *
signal(int signo, Sigfunc *func) {
  struct sigaction    act, oact;

  act.sa_handler = func;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;

  if (signo != SIGALRM) {
        act.sa_flags |=SA_RESTART;  /*SVR4, 4.3+BSD*/
  }

  if (sigaction(signo, &act, &oact) <0)
        return(SIG_ERR);

  return (oact.sa_handler);
}
```

Sigsetjmp and siglongjmp Functions

```
#include <set jmp.h>
int sigsetjmp (sigjmp_buf env, int savemask) ;
        Returns: 0 if called directly, nonzero if returning from a call to siglongjmp

int siglongjmp (sigjmp_buf env, int val) ;
```

- What happens to the signal mask for the process if we longjmp out of the signal handler?
- sigsetjmp and siglongjmp saves and restores the signal mask - use these functions for nonlocal branching from a signal handler
  - If *savemask* is nonzero then sigsetjmp also saves the current signal mask of the process in *env*
  - When siglongjmp is called, if the *env* argument was saved with nonzero *savemask*, then siglongjmp restores the saved signal mask

Sigsetjmp and siglongjmp Functions

<Example of sigsetjmp and siglongjmp>

```
static sigjmp_bufjmpbuf;
static volatile  sig_atomic_t   canjump;
int main(void) {
        …
        if(sigsetjmp(jmpbuf, 1)) exit(0);
        canjump=1;                  /* now sigsetjmp() is OK */
        …
}
static void sig_usr1(int signo) {
        ...
        if (canjump==0) return;    /* unexpected signal,         */
                                   /* not jmpbuf ready, ignored */
        …
        canjump = 0;
        siglongjmp(jmpbuf, 1);
}
```

- Use of *canjump* variable
  - protection against the signal handler being called when the jump buffer isn't initialized by sigsetjmp
  - sig_atomic_t : variable can be written without being interrupted (ex. No page boundary crossing)

Sigsuspend Function (1/4)

```
#include <signal.h>
int sigsuspend (const sigset_t *sigmask) ;
                                Returns: -1 with errno set to EINTR
```

- Performs resetting the signal mask and put the process to sleep in a single atomic operation
  - Signal mask of the process is set to the value pointed to by *sigmask.*
  - The process is also suspended until a signal is caught or until a signal occurs that terminates the process
  - If a signal is caught and if the signal handler returns, then `sigsuspend` returns and the signal mask of the process is set to its *old value*

Sigsuspend Function (2/4)

```
/* protect critical regions of code from interrupt signals : a wrong way */
sigset_t newmask, oldmask;

sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
        /* block SIGINT and save current signal mask */
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

        /* critical region of code */

        /* reset signal mask which unblocks SIGINT */
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");

pause(); /* wait for signal to occur */

        /* and continue processing ... */
```

- Problem: any signal between the second sigprocmask() and pause() gets lost.

Sigsuspend Function (3/4)

```
/* protect critical regions of code from interrupt signals : a right way */
sigset_t  newmask, oldmask, zeromask;
if (signal(SIGINT, sig_int) == SIG_ERR) err_sys("signal(SIGINT) error");

sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
          /* block SIGINT and save current signal mask */
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) err_sys("SIG_BLOCK error");

          /* critical region of code */

          /* allow all signals and pause */
if (sigsuspend(&zeromask) != -1) err_sys("sigsuspend error");

          /* reset signal mask which unblocks SIGINT */
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) err_sys("SIG_SETMASK error");

          /* and continue processing ... */
```

- Eliminated the previous problems. (unblock and pause)
- Note: need the second `sigprocmask`() to unblock SIGINT because the return of `sigsuspend`() set the signal mask to its value before the call.

Sigsuspend Function (4/4)

```
volatile sig_atomic_t   quitflag;       /* set nonzero by signal handler */
int main(void) {
      sigset_t       newmask, oldmask, zeromask;
       if (signal(SIGINT, sig_int) == SIG_ERR) err_sys("signal(SIGINT) error");
       if (signal(SIGQUIT, sig_int) == SIG_ERR) err_sys("signal(SIGQUIT) error");

       sigemptyset(&zeromask);     sigemptyset(&newmask);
       sigaddset(&newmask, SIGQUIT);  /* block SIGQUIT and save current signal mask */

       if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)  err_sys("SIG_BLOCK error");

       while (quitflag == 0)  sigsuspend(&zeromask);

       /* SIGQUIT has been caught and is now blocked; do whatever */
       quitflag = 0;
       if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)  err_sys("SIG_SETMASK error");
       exit(0);
}
void sig_int(int signo)    {  /* one signal handler for SIGINT and SIGQUIT */
       if (signo == SIGINT) printf("\ninterrupt\n");
       else if (signo == SIGQUIT) quitflag = 1;   /* set flag for main loop */
       return;
}
```

- `sigsuspend` to wait for a global variable to be set.

Sleep(1/2)

```
static void sig_alrm(void) {
      return; /* nothing to do, just returning wakes up sigsuspend() */
}

unsigned int sleep(usigned int nsecs) {
     struct sigaction   newact, oldact;
     sigset_t              newmask, oldmask, suspmask;
     unsigned int       unslept;

     newact.sa_handler = sig_alrm;
     sigemptyset(&newact.sa_mask);
     newact.sa-flags = 0;
     sigaction(SIGALRM, &newact, &oldact);
                        /* set our handler, save previous information */

     sigemptyset(&newmask);
     sigaddset(&newmask, SIGALRM);
                        /* block SIGALRM and save current signal mask */
     sigprocmask(SIG_BLOCK, &newmask, &oldmask);
```

Sleep(2/2)

```
    alarm(nsecs);

    suspmask = oldmask;
    sigdelset(&suspmask, SIGALRM);  /* make sure SIGALRM isn't blocked */

    sigsuspend(&suspmask);                  /* wait for any signal to be caught */

    /* some signal has been caught, SIGALR is now blocked */

    unslept = alarm(0);
    sigaction(SIGALRM, &oldact, NULL);   /* reset previous action */

                            /* reset signal mask, which unblocks SIGALRM */
    sigprocmask(SIG_SETMASK, &oldmaks, NULL);

    return(unslept);
}
```

- Handles signals reliably
- Avoiding the race condition
- Do not handle any interactions with previously set alarms

Job-Control Signals

SIGCHLD	: Child process has stopped or terminated
SIGCONT	: Continue process, if stopped
SIGSTOP	: Stop signal (can't be caught or ignored)
SIGTSTP	: Interactive stop signal
SIGTTIN	: Read from controlling terminal by a background process group
SIGTTOUT	: Write to controlling terminal by member of a background process group

- Most applications don't handle these signals - interactive shell do all the work required to handle these signals.
  - A program that manages the terminal needs to handle job-control signals (ex: vi editor – suspended when control-Z and redraws the screen when fg)
- When type Control-Z (suspend Character), SIGTSTP sent to all processes in the foreground process group.

Job-Control Signals

```
/* How to handle SIGTSTP (handling a job-control signal) */
int main(void) {
int n;char          buf[BUFFSIZE];
/* only catch SIGTSTP if we're running with a job-control shell */
if (signal(SIGTSTP, SIG_IGN) == SIG_DFL)
          signal(SIGTSTP, sig_tstp);
while ( (n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
          if (write(STDOUT_FILENO, buf, n) != n) err_sys("write error");
          if (n < 0) err_sys("read error");
exit(0);
}
static void sig_tstp(int signo) {       /* signal handler for SIGTSTP */
sigset_t  mask;
/* ... move cursor to lower left corner, reset tty mode ... */

/* unblock SIGTSTP, since it's blocked while we're handling it */
sigemptyset(&mask);
sigaddset(&mask, SIGTSTP);
sigprocmask(SIG_UNBLOCK, &mask, NULL);

signal(SIGTSTP, SIG_DFL);     /* reset disposition to default */

kill(getpid(), SIGTSTP);      /* and send the signal to ourself */

          /* we won't return from the kill until we're continued */

signal(SIGTSTP, sig_tstp);    /* reestablish signal handler */

/* ... reset tty mode, redraw screen ... */
return;
}
```