Signals

Introduction

- Signals
  - Software interrupts
  - provides a way of handling asynchronous events.
    - E.g. A user types the interrupt key to stop a program.

- Signal name
  - Begins with ‘SIG’.
    - E.g. SIGABRT, SIGTERM, SIGALRM, …
  - Is defined by positive integer constants in <signal.h>
    - E.g. #define SIGHUP 1
    - Depends on architecture and OS.

Introduction

- Examples of signal generation
  - When user press ‘Ctrl-C’ on the terminal.
    - Generates SIGINT signal.
  - When executes an invalid memory references.
    - Generates SIGSEGV signal. (SEGmentation Violation)
  - When superuser want to kill a process.
    - Generates SIGKILL signal.
  - When a process writes to a pipe after the reader has terminated.
    - Generates SIGPIPE signal.

Introduction

- Disposition of the signal(called action).
  - Ignore the signal
    - SIGKILL and SIGSTOP cannot be ignored.
  - Catch the signal
    - We should tell the kernel to call a signal handler function whenever the signal occurs.
  - Execute the default action
    - The default action for most signals is to terminate.

Signals

- Signals for terminating processes
  - SIGHUP
    - This signal is sent to the controlling process(session leader) associated with a controlling terminal if a disconnection is detected.
    - termination

Signals

- Signals for terminating processes(cont.)
  - SIGINT
    - It is often used to terminate a runaway program.
    - It is sent to all foreground processes.
    - [CTRL-C]
    - termination
  - SIGQUIT
    - Is similar to SIGINT, but generates a core file.
    - [CTRL-\]
    - termination with core

"core" means that a memory image of the process is left in the file named core of the current working directory.
It can be used for debugging.

Signals

- Signals for terminating processes(cont.)
  - SIGABRT
    - abnormal termination (abort()).
    - terminate
  - SIGKILL
    - irrevocable termination signal.
    - It provides the superuser with a sure way to kill process.
    - cannot be caught or ignored.
    - terminate
  - SIGTERM
    - default signal sent out by the kill command.
    - terminate

Signals

- Signals for terminating processes(cont.)
  - SIGCHLD(or SIGCLD)
    - When a process terminates, it is sent to parent.
    - Ignore
    - The parent must catch using wait().

Signals

- Signals for suspending or resuming.
  - SIGCONT
    - Continue a stopped process.
    - resume
  - SIGSTOP
    - Stop a process.
    - Cannot be caught or ignored.
    - suspend

Signals

- Signals for suspending or resuming(cont).
  - SIGTSTP
    - When we type the terminal suspend key.
    - [CTRL-Z]
    - suspend
  - SIGTTIN
    - When a background process tries to read from terminal.
    - suspend
  - SIGTTOU
    - When a background process tries to write to terminal.
    - suspend

Signals

- Signals triggered by a physical circumstance
  - SIGILL
    - illegal hardware instruction
    - terminate
  - SIGTRAP
    - An implementation-defined hardware fault.
    - use this signal to transfer control to a debugger when a breakpoint instruction is executed.
    - terminate with core
  - SIGBUS
    - bus error
    - terminate

Signals

- Signals triggered by a physical circumstance(cont.)
  - SIGFPE
    - arithmetic error (floating point exception)
    - terminate
  - SIGSEGV
    - Invalid memory reference
    - terminate with core

Signals

- Signals available for use by the programmer
  - SIGUSR1, SIGUSR2
    - User-defined signal, for use in application programs
    - terminate
- Signal generated when a pipe is closed
  - SIGPIPE
    - pipe without reader
    - terminate
- Refer the textbook for entire list of signals!

signal()

```
#include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);
                    Returns: previous disposition of signal if OK, SIG_ERR on error
```

- installs a signal handler for the signal with *signo*.
  - *signo* is the name of the signal.
  - *func* is one of the followings.
    - SIG_IGN: Ignore the signal.
    - SIG_DFL: set the action of the signal to its default value.
    - a user-specified function(signal handler).
  - It is possible to use one signal handler for several signals.
  - Return value is the previous signal handler.

signal()

- Example

```
#include        <signal.h>

void myhandler(int signo)
{
        switch (signo) {
        case SIGQUIT : printf("SIGQUIT(%d) is caught\n",SIGQUIT);
                break;
        case SIGTSTP : printf("SIGTSTP(%d) is caught\n",SIGTSTP);
                break;
        case SIGTERM : printf("SIGTERM(%d) is caught\n",SIGTERM);
                break;
        case SIGUSR1 : printf("SIGUSR1(%d) is caught\n",SIGUSR1);
                break;
        default: printf("other signal\n");
        }
        return;
}
```

signal()

- Example(cont.)

```
int main(void)
{
        signal(SIGQUIT, myhandler);
        signal(SIGTSTP, SIG_DFL);                    //use default handler
        signal(SIGTERM, myhandler);
        signal(SIGUSR1, myhandler);

        for (;;)
                pause();
}
```

Stop until it receive a signal.

signal()

- Execution

```
$ ./a.out
^\
SIGQUIT(3) is caught
^Z
[1]+  Stopped                 ./a.out
$ ps
  PID TTY          TIME CMD
15554 pts/2    00:00:00 bash
15587 pts/2    00:00:00 a.out
15588 pts/2    00:00:00 ps
$ kill 15587
SIGTERM(15) is caught
$ kill -USR1 15587
SIGUSR1(10) is caught
$
```

kill()

```
#include <signal.h>

int kill(pid_t pid, int signo);
                                        Both return: 0 if OK, -1 on error
```

- Sends a signal to a process or a group of processes.

kill()

- *pid* argument
  - pid > 0
    - The signal is sent to process with *pid*.
  - pid == 0
    - The signal is sent to all processes in the process group of the current process.
  - pid == -1
    - The signal is sent to all processes on the system for which the sender has permission to send the signal.
  - pid < -1
    - The signal is sent to all processes whose process group ID equals the absolute value of *pid*.

raise()

```
#include <signal.h>

int raise(int signo);
                                   Both return: 0 if OK, -1 on error
```

- Sends a signal to itself.
  - raise(signo); is equivalent to kill(getpid(), signo);

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, -1 on error
```

- Signal sets
  - # of different signals can exceed # of bits in an integer, so we need a data type to represent multiple signals.
- Why signal set?
  - We'll use this with such functions as sigprocmask (in the next section) to tell the kernel not to allow any of the signals in the set to occur.

Signal sets

- sigemptyset()
  - initializes the signal set to empty.
- sigfillset()
  - initializes set to full, including all signals.
- sigaddset() and sigdelset()
  - add and delete respectively signal *signo* from *set*.
- sigismember( )
  - tests whether *signo* is a member of *set*

sigprocmask()

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
                                        Returns: 0 if OK, -1 on error
```

- signal mask of a process is the set of signals currently blocked from delivery to that process.
- change the list of currently blocked signals.
  - *how* argument
    - SIG_BLOCK
      - Union of the current set and the *set* argument.
      - The signals in *set* are added into the current set.

sigprocmask()

- SIG_UNBLOCK
  - Intersection of the current set and the complement of the *set* argument.
  - The signals in *set* are removed from the current set.
- SIG_SETMASK
  - Replace the current set with the *set* argument.

- *oset* argument
  - if non-null, the previous value of the signal mask is stored in *oset*.

sigprocmask()

- Example

```
#include "apue.h"
#include <errno.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()

```
#include <signal.h>

int sigpending(sigset_t *set);
                                        Returns: 0 if OK, -1 on error
```

- Returns the set of signals that are currently pending.
  - The signal mask of pending signals is stored in *set*.

sigpending()

- Example

```
#include "apue.h"
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");

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

Example(cont.)

```
static void sig_quit(int signo) {
   printf("caught SIGQUIT\n");
   if (signal(SIGQUIT, SIG_DFL) == SIG_ERR)
   err_sys("can't reset SIGQUIT");
}
```

Execution

```
$ ./a.out
^\                        generate signal once (before 5 seconds are up)
SIGQUIT pending           after return from sleep
caught SIGQUIT            in signal handler
SIGQUIT unblocked         after return from sigprocmask
^\Quit(coredump)          generate signal again
$ ./a.out
^\^\^\^\^\^\^\^\^\^\      generate signal 10 times (before 5 seconds are up)
SIGQUIT pending
caught SIGQUIT            signal is generated only once
SIGQUIT unblocked
^\Quit(coredump)          generate signal again
```

sigaction()

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *act, struct sigaction *oact);
                                                  Returns: 0 if OK, -1 on error
```

- Examine or change or both the action associated with a specific signal.
  - *signo* argument
    - The signal number whose action we are changing.
  - If *act* is non-null, the new action for *signo* signal is installed.
  - If *oact* is non-null, the previous action is saved in *oact*.
- It supersedes signal().

sigaction()

```
struct sigaction {
        void        (*sa_handler)(int);                  /* addr of signal handler, */
                                                         /* or SIG_IGN, or SIG_DFL */
        sigset_t    sa_mask;                             /* additional signals to block */
        int         sa_flags;                            /* signal options */

        /* alternate handler */
        void        (*sa_sigaction)(int, siginfo_t *, void *);
};
```

- sa_handler
  - specifies the action to be associated with *signo*.
  - SIG_DFL, SIG_IGN, or a pointer to a signal handler.

sigaction()

- sa_mask
  - a mask of signals which should be blocked during execution of the signal handler.
- sa_flags
  - specifies a set of flags which modify the behavior of the signal handling process.
  - SA_INTERRUPT, SA_NOCLDSTOP, SA_NOCLDWAIT, SA_NODEFER, SA_ONSTACK, SA_RESETHAND, SA_RESTART, SA_SIGINFO
- sa_sigaction
  - is obsolete and should not be used.

sigaction()

- Example

```
#include    <signal.h>

void catchint(int signo)
{
            printf(“\nCATCHINT: signo=%d\n”, signo);
            printf(“CATCHINT: returning \n”);
}

int main()
{
            static struct sigaction act;
            act.sa_handler = catchint;
            sigfillset(&(act.sa_mask));
            sigaction(SIGINT, &act, NULL);
```

sigaction()

- Example(cont.)

```
            printf(“sleep call #1\n”);
            sleep(1);
            printf(“sleep call #2\n”);
            sleep(1);
            printf(“sleep call #3\n”);
            sleep(1);
            printf(“sleep call #4\n”);
            sleep(1);
            printf(“Existing \n”);
            exit(0);
}
```

sigaction()

- Execution

```
$ ./a.out
sleep call #1
sleep call #2
sleep call #3
^C
CATCHINT: signo=2
CATCHINT: returning
sleep call #4
Existing
$
```

alarm()

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
                              Returns: 0 or number of seconds until previously set alarm
```

- Set a timer that will expire at a specified time in the future.
  - When the timer expires, SIGALRM is generated.
  - Default action is to terminate the process, but most processes catch this signal.
  - There is only one alarm clock per process.
    - If, when we call alarm, a previously registered alarm clock for the process has not yet expired, the number of seconds left is returned. The previously registered alarm clock is replaced by the new one.

alarm()

- Example

```
/* header file */

int alarm_flag = FALSE;

/* signal handler */
void setflag (int sig){
        alarm_flag = TRUE;
}

int main (int argc, char **argv) {
        int nsecs, j;
        pid_t pid;
        static struct sigaction act;

        if (argc <=2){
                fprintf (stderr, "Usage: ./a.out #seconds message\n");
                exit (1);
        }
```

alarm()

- Example(cont.)

```
if ((nsecs = atoi(argv[1])) <= 0){
        fprintf(stderr, “invalid time\n");
        exit (2);
}

switch (pid = fork()){
        case 0:                 /* child */
                break;
        default:                /* parent */
                printf ("child process id %d\n", pid);
                exit (0);
}
act.sa_handler = setflag;
sigaction(SIGALRM, &act, NULL);

alarm(nsecs);
pause();
```

alarm()

- Example(cont.)
-
```
        if (alarm_flag == TRUE){
                printf ("Alarmed!\t");
                for (j = 2; j < argc; j++)
                        printf ("%s", argv[j]);
                printf ("\n");
        }
        exit (0);
}
```

- Execution

```
$ ./a.out 3 hello world
child process id 15017
$ Alarmed!      helloworld
```

pause()

```
#include <unistd.h>

int pause(void);
                                   Returns: -1 with errno set to EINTR
```

- Suspends the calling process until a signal is caught.

abort()

```
#include <stdlib.h>

void abort(void);
                                        This function never returns
```

- Sends the SIGABRT to the caller.

sleep()

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
                                        Returns: 0 or number of unslept seconds
```

- Causes the calling process to be suspended until
  - the amount of time specified by seconds has elapsed, or a signal is caught by the process.
  - return value
    - 0 if the requested time has elapsed, or the number of seconds left to sleep.