Lecture 7 - Signals (Chapter 10)

- Signal concepts

  Signals are software interrupts in the form of a number.
  Signals are asynchronous (i.e. that can appear at any time).

  Every signal has a name that begins with "SIG".
  For example, SIGABRT is the abort signal that is generated where a process calls abort.
  The signal names are representative of positive constants as defined in <signal.h>.

  Early versions of UNIX had 15 different signals, and some of the later ones 31.

  Conditions that generate signals:

  1. The terminal generated signals occur when users press certain keys.

  2. Hardware exceptions generate signals: (i.e. divide by 0, invalid memory references, etc).

  3. The kill(2) function allows a process to send any signal to any process or process group.

  4. The kill(1) command allows us to send signals to other processes.

  5. Software conditions can generate signals (i.e. SIGURG, SIGPIPE, SIGALRM, etc).

  Signal dispositions (i.e. Actions associated with signals):

  1. **Ignore** the signal. Some signals can be ignored while others cannot. SIGKILL and SIGSTOP are examples of signals that cannot be ignored.

  2. **Catch** the signal. We can register a function to be called when a signal occurs, and perform whatever actions we like.

  3. Allow the **default** action to apply. Every signal has a default action (p. 292). Note the default action in most cases is to terminate the process.

Portion of Figure 10.1 (p. 292)

Signal Name	Description	Default Action
SIGABRT	Abnormal termination (abort)	Terminate with core
SIGALRM	Time out (alarm)	Terminate
SIGCHLD	Change in status of child	Ignore
SIGCONT	Continue stopped process	Continue / ignore
SIGFPE	Floating point exception	Terminate with core
SIGHUP	Hangup	Terminate
SIGINT	Terminal interrupt	Terminate
SIGKILL	Termination	Terminate (not catchable)
SIGQUIT	Terminal quit	Terminate with core
SIGSEGV	Invalid memory reference	Terminate
SIGSTOP	Stop process	Stop process (not catchable)
SIGTERM	Terminatation	Terminate
SIGUSR1	User-defined signal	Terminate
SIGUSR2	User-defined signal	Terminate

Note: See discussions of individual signals pages 293-298.

- Signal function

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

  OR

  ```
  typedef void   Sigfunc(int);
  sigfunc        *signal(int, Sigfunc *);
  ```

  Returns: previous disposition of signal if OK, SIG_ERR on error

  The **signo** argument is the signal number.
  The **func** argument is one of: SIG_IGN, SIG_DFL, the address of a function to be called.

  Example:

```
static void sig_usr(int);

int main(void)
{
        if (signal(SIGUSR1, sig_usr) == SIG_ERR)
                perror("Installation of SIGUSR1 handler failed");

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

- Unreliable signals

  In early versions of UNIX, signals were unreliable (i.e. not guaranteed for delivery). We could catch a signal or ignore it, but could not block its delivery.

  One problem with these early functions is that the action for the signal was reset to default each time the signal occurred.

  Example:

```
int     sig_int();

…
signal(SIGINT, sig_int);
…

sig_int()
{
        signal(SIGINT, sig_int);
        …
}
```

  This means there is a window of time in which a second signal could get executed with the default action (i.e. terminate the process). This is another form of a race condition.

- Interrupted system calls

  If a process received a signal while the process was blocked in a "slow" system call, the system call was interrupted (i.e. returned with an error, and errno set to EINTR).

  "Slow" system calls are defined as those that can block forever:

  1. reads calls for files that can block the caller forever if data is not present (i.e. pipes, terminals, network devices, etc).
  2. write call to those same files if data cannot be accepted immediately
  3. open calls for files that block until some condition occurs (i.e. open of a terminal device that depends on modem answering).
  4. certain ioctl operations
  5. some of the inter-process communications functions

Example:

```
again:
        if ((n = read(fd, buff, BUFFSIZE)) < 0)
        {
                if (errno == EINTR)
                        goto again;
        }
```

To save from the mess of handling this situation, some versions of UNIX automatically restarted some system calls (i.e. ioctl, read, readv, write, writev, wait, and waitpid).

Note: See figure 10.3 (page 305) for a list of functions and if they restart system calls automatically or not.

- Reentrant functions

  If a signal handler returns (i.e. does not call exit), then we need to resume our code where we left off.

  It is possible that our program was in the middle of some function call (i.e. malloc, etc.) that should be completed on the return of the handler.

  It is also possible that the function we are in the middle of executing relies on some static data structure that can be corrupted during the execution of the handler if we call it again.

  POSIX.1 specifies the functions that are guaranteed to be reentrant (page 306), and some versions of UNIX have defined more.

- SIGCLD semantics

  Two signals that often generate confusion are SIGCLD and SIGCHLD.

  SIGCHLD behaves like most other signals.

  SIGCLD (on System V systems):

  - If the process specifically sets it disposition to SIG_IGN, children of the calling process will not generate zombie processes.
  - If we set the disposition to SIGCLD to be caught, the kernel immediately checks if there are any child processes ready to be waited for, and calls the SIGCLD handler.

Example (Broken signal handler for System V):

```
int main()
{
        pid_t   pid;

        if (signal(SIGCLD, sig_cld) == SIG_ERR)
                perror("signal error");

        if ((pid = fork()) < 0)
                perror("fork error");
        else if (pid == 0)
        {
                sleep(2);
                _exit(0);
        }

        pause();
        return(0);
}

static void sig_cld()
{
        pid_t   pid;
        int     status;

        printf("SIGCLD received\n");

        if (signal(SIGCLD, sig_cld) == SIG_ERR)
                perror("signal error");

        if ((pid = wait(&status)) < 0)
                perror("wait error");

        printf("pid = %d\n", pid);

        return;
}
```

- Reliable signal terminology and semantics

  A signal is **generated** for a process when the event that causes the signal occurs.
  A signal is **delivered** to a process when the action for a signal is taken.
  The time between generation of signal and delivery, the signal is **pending**.

  Note: It is possible to **block** a signal and leave it pending for some time.

If a blocked signal is generated more than once before being delivered, POSIX allows the system to deliver the signal either once or more than once.

Each process has a **signal mask** that defines the set of signals currently blocked from delivery to that process.

- Kill and raise

The "kill" function sends a signal to a process or a group of processes.
Thse "raise" function allows a process to send a signal to itself.

```
int     kill(pid_t pid, int signo)
int     raise(int signo)
```

Various possible values for pid:

o	pid > 0	Sent to the process whose process id is "pid"
o	pid == 0	Sent to all processes whose PGID equals PGID of sender.
o	Pid < 0	Sent to all processes whose PGID equals |pid|.
o	Pid == -1	Unspecified

Note: We need permission to send a signal to a process. Our EUID must match that of the process, or we must be root.

- Alarm and pause

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

The alarm function sets an alarm clock to send a SIGALRM signal after "seconds".

It is possible due to system overhead for the signal to be delivered slightly off time.

There is only one of these alarm clocks per process. If we call alarm more than once, we are resetting the alarm clock to a new value.

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

Pause causes the process to suspend until a signal is caught.

Example: (imcomplete version of sleep):

```
static void sig_alrm(int signo)
{
        return;
}

unsigned int sleep1(unsigned int nsecs)
{
        if (signal(SIGALRM, sig_alrm) == SIG_ERR)
                return(nsecs);

        alarm(nsecs);
        pause();
        return(alarm(0));
}
```

Side effects of "sleep1":

1. If caller is already using "alarm", we have reset his value.
2. We have modified the disposition of SIGALRM.
3. There is a race condition between first call to alarm and the call to pause.

- Signal sets

A **signal set** is set of zero or more signals to be operated on.

```
int     sigemptyset(sigset_t    *set);
int     sigfillset(sigset_t *set);
int     sigaddset(sigset_t *set, int signo);
int     sigdelset(sigset_t *set, int signo);

int     sigismember(const sigset_t *set, int signo);
```

Note: You must call either sigemptyset or sigfillset before using a signal set.

Note: See pages 320 for possible implementation of signal sets.

- Sigprocmask

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);

If "oset" is a nonnull pointer, the current signal mask for the process is returned.

If "set" is a nonnull pointer, then the "how" argument indicates how the current signal mask is modified:

SIG_BLOCK	The new signal mask is union of current and "set"
SIG_UNBLOCK	The new signal mask is intersection of current and complement of "set".
SIG_SETMASK	The new signal mask is "set"

Example:

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

        if (signal(SIGQUIT, sig_quit) == SIG_ERR)
                perror("signal error");

        sigemptyset(&newmask);
        sigaddset(&newmask, SIGQUIT);

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

        sleep(5);

        if (sigpending(&pendmask) < 0)
                perror("sigpending");

        if (sigismember(&pendmask, SIGQUIT))
                printf("SIGQUIT pending\n");

        if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
                perror("sig_setmask");

        printf("SIGQUIT unblocked\n");

        sleep(5);    exit(0);
}
```

- Sigpending

  Int sigpending(sigset_t *set);

  Sigpending returns the set of signals that are blocked from delivery and currently pending for the calling process.

- Sigaction

  Sigaction allows us to examine or modify the action associated with a particular signal.

  int sigaction(int signo, const struct sigaction *act, struct sigaction *oact);

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

  If "act" is non-null, we are modifying the action for the signal.
  If "oact" is non-null, the system returns the previous action for the signal.

  Note: the handler installed by sigaction remains installed until we change it!

  Possible values for sa_flags (see page 326):

SA_NOCLDSTOP	don't send SIGCHLD if child stops
SA_RESTART	system calls automatically restarted
SA_NOCLDWAIT	don't create zombie processes of children

  Example:

```
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_INTERRUPT;
        else
                act.sa_flags |= SA_RESTART;
```

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

        return(oact.sa_handler);
}
```

- Sigsetjmp and Siglongjmp

```
int     sigsetjmp(sigjmp_buf env, int savemask);
void    siglongjmp(sigjmp_buf env, int val);
```

Same as setjmp and longjmp execpt allows restoration of signal mask.

- Sigsuspend

We might want to 1) block a signal 2) perform some task 3) unblock 4) wait for signal.

Example (incorrect way):

```
sigset_t        new, old;

sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);

if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        Perror("sig_block sigint");

<<< critical code section >>>

if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        perror("sig_setmask");

pause();
```

Sigsuspend was created to provide an atomic way to reset a signal mask, and pause.

```
int     sigsuspend(const sigset_t *sigmask);
```

Returns: -1 with errno set to EINTR

Note: see code page 337-338 for child and parent synchronization routines.

- Abort

  void abort(void);

  As mentioned, abort causes program to terminate with core.
  Abort sends the SIGABRT signal to the process.

  Note: See code page 341 for a sample implementation of abort

- System function

```
int system(const char *cmdstring)
{
        pid_t                   pid;
        int                     status;
        struct sigaction        ignore, saveintr, savequit;
        sigset_t                childmask, savemask;

        if (cmdstring == NULL)
                return(1);

        ignore.sa_handler = SIG_IGN;
        sigemptyset(&ignore.sa_mask);
        ignore.sa_flags = 0;

        if (sigaction(SIGINT, &ignore, &saveintr) < 0)
                return(-1);
        if (sigaction(SIGQUIT, &ignore, &savequit) < 0)
                return(-1);

        sigemptyset(&childmask);
        sigaddset(&childmask, SIGCHLD);

        if (sigprocmask(SIG_BLOCK, &childmask, &savemask) < 0)
                return(-1);

        if ((pid = fork()) < 0)
                status = -1;
        else if (pid == 0)              /* child */
        {
                sigaction(SIGINT, &saveintr, NULL);
                sigaction(SIGQUIT, &savequit, NULL);
                sigprocmask(SIG_SETMASK, &savemask, NULL);
                execl("/bin/sh", "sh", "-c", cmdstring, (char *) 0);
                _exit(127);
        }
```

```
        else                            /* parent */
        {
                while (waitpid(pid, &status, 0) < 0)
                        if (errno != EINTR)
                        {
                                status = -1;
                                break;
                        }
        }

        if (sigaction(SIGINT, &saveintr, NULL) < 0)
                return(-1);

        if (sigaction(SIGQUIT, &savequit, NULL) < 0)
                return(-1);

        if (sigprocmask(SIG_SETMASK, &savemask, NULL) < 0)
                return(-1);

        return(status);
}
```

- Sleep

unsigned int sleep(unsigned int seconds);

Returns: 0 or number of unslept seconds

This function causes the process to suspend until:

- "seconds" has elapsed
- a signal is caught and the handler returned

Example:

```
static void sig_alrm(int signo)
{
        return;
}

unsigned int sleep2(unsigned int nsecs)
{
        struct sigaction        newact, oldact;
        sigset_t                newmask, oldmask, suspmask;
        unsigned int            unslept;
```

```
        /*  setup SIGALRM handler */

        newact.sa_handler = sig_alrm;
        sigemptyset(&newact.sa_mask);
        newact.sa_flags = 0;

        sigaction(SIGALRM, &newact, &oldact);

        /*  Block SIGALRM */

        sigemptyset(&newmask);
        sigaddset(&newmask, SIGALRM);
        sigprocmask(SIG_BLOCK, &newmask, &oldmask);

        alarm(nsecs);

        /*  Make sure SIGALRM is unblocked and suspend */

        suspmask = oldmask;
        sigdelset(&suspmask, SIGALRM);

        sigsuspend(&suspmask);

        /*  We caught signal, SIGALRM is now blocked */

        unslept = alarm(0);

        /*  Reset things and return */

        sigaction(SIGALRM, &oldact, NULL);
        sigprocmask(SIG_SETMASK, &oldmask, NULL);

        return(unslept);
}
```

- Job control signals

SIGCHLD	child has stopped or terminated
SIGCONT	continue process if stopped
SIGSTOP	stop process
SIGTSTP	Interactive stop signal
SIGTTIN	Read from controlling terminal by background process
SIGTTOU	Write to controlling terminal by background process