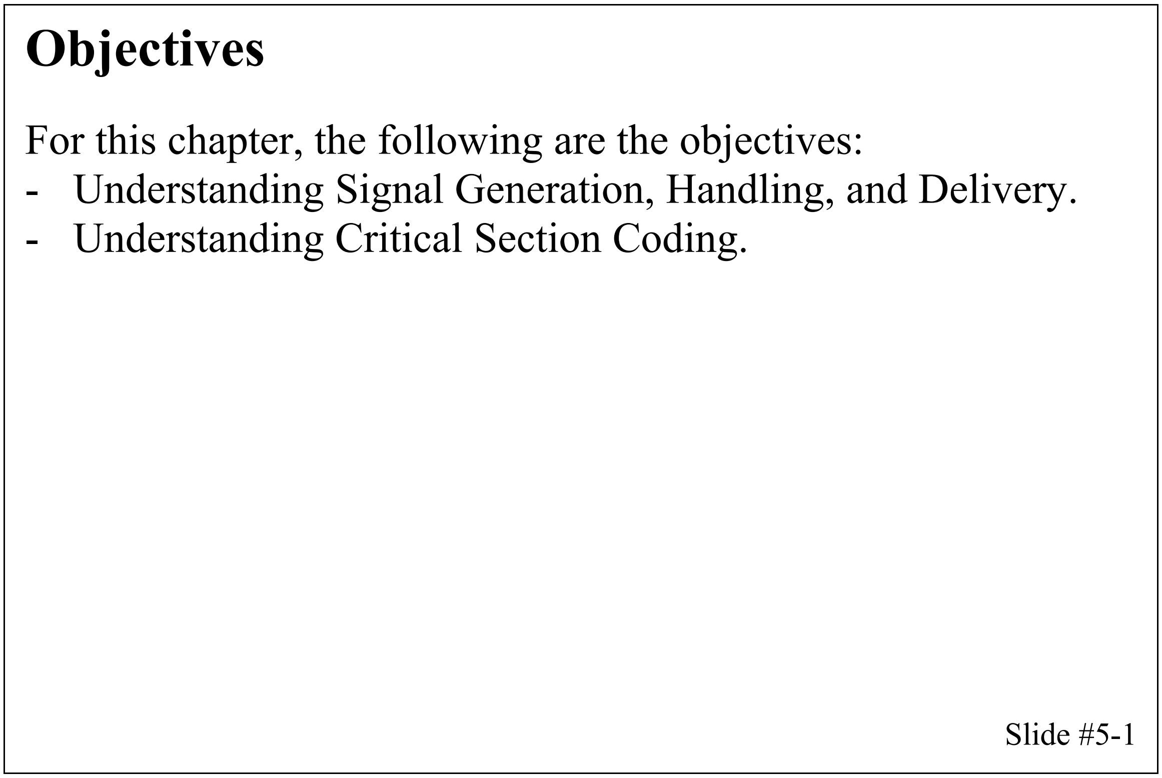
Chapter 5

Signals

Chapter Objectives

To understand concepts of Signals used over the duration of the course.



Objectives

For this chapter, the following are the objectives:

- Understanding Signal Generation, Handling, and Delivery.
- Understanding Critical Section Coding.

Slide #5-1

Notes

In this chapter, we examine asynchronous events and signals.

The objective of this chapter is to provide an understanding of concepts on signals that are used over the duration of the course.

Chapter Organization

1. **Objective**: Introduction to
   - Asynchronous Events, and
   - Signals.

2. **Description**: A signal is the notification of an asynchronous event.
   This chapter provides an introduction to the concepts used in course.

3. **Concepts Covered in Chapter**:
   - Introduction to Signal Generation, Delivery and Handling.
   - The design and architecture of reliable UNIX Signals.
   - LINUX Notes

4. **Prior Knowledge**:
   same as Chapter #1

5. **Teaching & Learning Strategy**:
   Discussion questions are,
   - What are signals?
   - re-startable reads?,

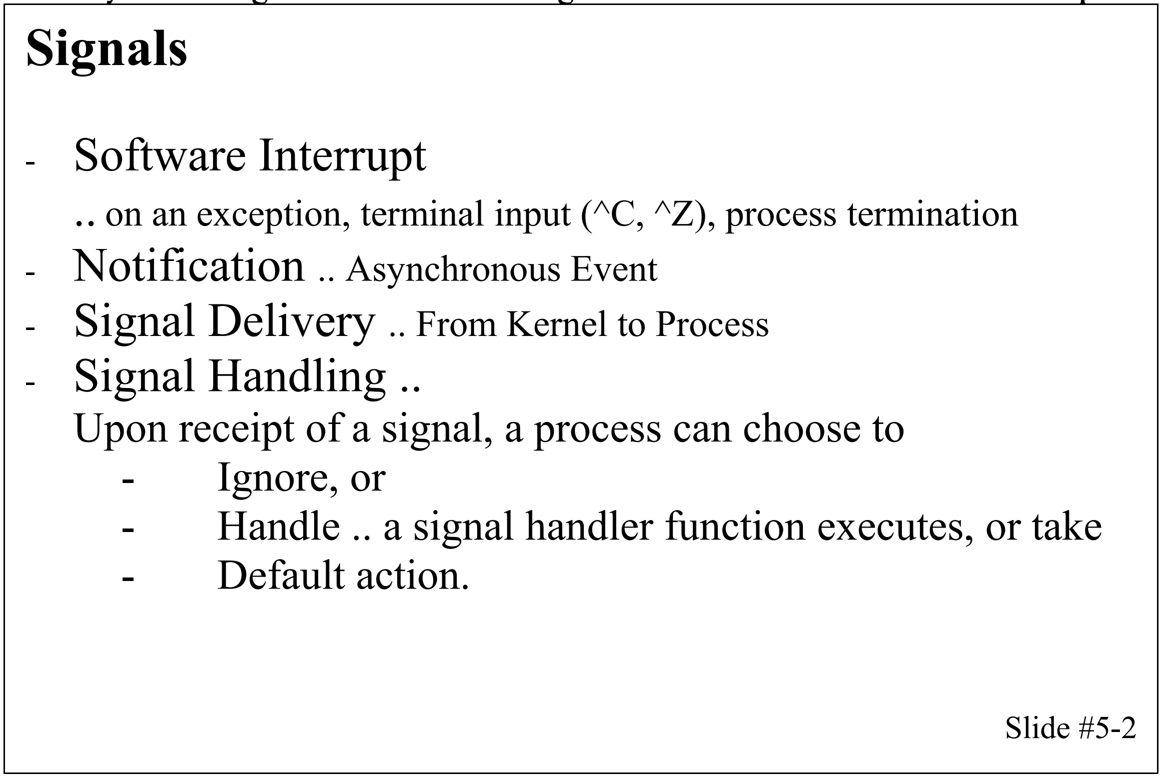
6. **Teaching Format**:
   Theory + Homework Assignments

7. **Study Time**: 120 Minutes (Lecture & Theory)
   + ~45 minutes (Homework Assignments)

8. **Assessment**: Group Homework Assignments

9. **Homework Eval**: Group

10. **Chapter References**:
    Stevens; APUE:
    Vahalia; UI:
    Robbins & Robbins; USP: Ch #8



Signals

- Software Interrupt

  .. on an exception, terminal input (^C, ^Z), process termination
- Notification .. Asynchronous Event
- Signal Delivery .. From Kernel to Process
- Signal Handling ..

  Upon receipt of a signal, a process can choose to
  - Ignore, or
  - Handle .. a signal handler function executes, or take
  - Default action.

Slide #5-2

Notes

1. Signals – Basic Concepts

1.1. Signals can occur at any time during a programs execution.
(“Asynchronous” vs “Synchronous”).

1.2. Signals are generated by several types of events:

	Event	Cause
1.	Exception	Illegal Memory access, divide by 0,
2.	Software	Alarm(), kill()
3.	Terminal Input	Interrupt ‘^C’, Stop ‘^Z’
4.	Process Termination	`exit()`, child termination

1.3. When a process receives a signal, it can choose to

1.3.1. Ignore.
Nothing happens when the signal is sent.

1.3.2. Handle.
A “*signal handler*” function can be executed. It is a “designated” function that is executed when a signal is received. After a signal handler has executed, the process is supposed to resume exactly at the point where it left off, when the signal arrived.

1.3.3. Default action.
There is no “special” handling. For most signals the default action is to `exit()`.

1.4. List of Signals on Intel Linux

```
#define SIGHUP     1
#define SIGINT     2
#define SIGQUIT    3
#define SIGILL     4
#define SIGTRAP    5
#define SIGIOT     6
#define SIGBUS     7
#define SIGFPE     8
#define SIGKILL    9
#define SIGUSR1    10
#define SIGSEGV    11
#define SIGUSR2    12
#define SIGPIPE    13
#define SIGALRM    14
#define SIGTERM    15
#define SIGCHLD    17
#define SIGCONT    18
#define SIGSTOP    19
#define SIGTSTP    20
#define SIGTTIN    21
#define SIGTTOU    22
#define SIGURG     23
#define SIGXCPU    24
#define SIGXFSZ    25
#define SIGVTALRM  26
#define SIGPROF    27
#define SIGWINCH   28
#define SIGIO      29
#define SIGPWR     30
```

1.5. List of Signals in Solaris

```
#define SIGHUP    1   /* hangup */
#define SIGINT    2   /* interrupt (rubout) */
#define SIGQUIT   3   /* quit (ASCII FS) */
#define SIGILL    4   /* illegal instr (not reset when caught)*/
#define SIGTRAP   5   /* trace trap (not reset when caught) */
#define SIGIOT    6   /* IO Transfer instruction */
#define SIGABRT   6   /* used by abort,replace SIGIOT in future*/
#define SIGEMT    7   /* EMT instruction */
#define SIGFPE    8   /* floating point exception */
#define SIGKILL   9   /* kill (cannot be caught or ignored) */
#define SIGBUS    10  /* bus error */
#define SIGSEGV   11  /* segmentation violation */
#define SIGSYS    12  /* bad argument to system call */
#define SIGPIPE   13  /* write on a pipe with no readers */
#define SIGALRM   14  /* alarm clock */
#define SIGTERM   15  /* software termination signal from kill */
#define SIGUSR1   16  /* user defined signal 1 */
#define SIGUSR2   17  /* user defined signal 2 */
#define SIGCLD    18  /* child status change */
#define SIGCHLD   18  /* child status change alias (POSIX) */
#define SIGPWR    19  /* power-fail restart */
#define SIGWINCH  20  /* window size change */
#define SIGURG    21  /* urgent socket condition */
#define SIGPOLL   22  /* pollable event occured */
#define SIGIO SIGPOLL   /* socket IO possible(SIGPOLL alias) */
#define SIGSTOP   23  /* stop (cannot be caught or ignored) */
#define SIGTSTP   24  /* user stop requested from tty */
#define SIGCONT   25  /* stopped process has been continued */
#define SIGTTIN   26  /* background tty read attempted */
#define SIGTTOU   27  /* background tty write attempted */
#define SIGVTALRM 28  /* virtual timer expired */
#define SIGPROF   29  /* profiling timer expired */
#define SIGXCPU   30  /* exceeded cpu limit */
#define SIGXFSZ   31  /* exceeded file size limit */
#define SIGWAITING 32 /* process's lwps are blocked */
#define SIGLWP    33  /* special signal used by thread library */
#define SIGFREEZE 34  /* special signal used by CPR */
#define SIGTHAW   35  /* special signal used by CPR */
#define SIGCANCEL 36  /* thread cancel sig used by libthread */
#define SIGLOST   37  /* resource lost (eg, record-lock lost) */
```

2. POSIX Signals and default behavior

2.1. For most signals, the default behavior is to exit. The exit status in these cases is the signal number.

Name	Num	Action
SIGHUP	1	exit
SIGINT	2	exit
SIGKILL	9	exit
SIGPIPE	13	exit
SIGALRM	14	exit
SIGTERM	15	exit
SIGUSR1	16	exit
SIGUSR2	17	exit
SIGPOLL	22	exit
SIGVTALRM	28	exit
SIGPROF	29	exit
SIGCHLD	18	ignore
SIGPWR	19	ignore
SIGWINCH	20	ignore
SIGURG	21	ignore
SIGQUIT	3	core
SIGILL	4	core
SIGTRAP	5	core
SIGABRT	6	core
SIGFPE	8	core
SIGBUS	10	core
SIGSEGV	11	core
SIGSYS	12	core
SIGXCPU	30	core
SIGXFSZ	31	core
SIGCONT	25	restart
SIGSTOP	23	stop
SIGTSTP	24	stop
SIGTTIN	26	stop
SIGTTOU	27	stop

3. Chapter Reference: Robbins & Robbins #

`signal()`

- Old style ..
- Set signal handler
- Using Set-Reset paradigm

Problems with `signal()`

SYS V and its derivatives,

1. mask .. required for same signal when handler is executing
2. handler is reset.
3. system calls are interrupted, and not restarted.

Slide#3-3

Notes

1. **`signal()`** .. 3 arguments

   1.1. To set signal handler information

   ```
   #include <signal.h>

   oldfn() signal (signum, newfn)
      int     signum;
      void(*oldfn) (int);
      void(*newfn) (int);
   ```

   1.2. `int signum` is signal#

   1.3. `oldfn` and `newfn` are pointers to a function that takes an `int` and returns `void`. Also, can use `SIG_IGN` or `SIG_DFL`.

2. **Function Pointers**

   2.1. A function *resides* in the 'code' portion of the address map for a process.

   2.2. Each function has an address.

   2.2.1. The name of the function .. without arguments, provides the address of the function.

3. **Set Reset paradigm for signal handlers.**

   3.1. The signal handler information is stored in the task structure (or, user struct .. u_area) of a process. Therefore, if a subroutine function sets a handler, it will stay active even if the function is completed, and, can cause unintended effects in the calling function.

   3.2. Therefore, the signal handler needs to be reset to its previous state, just prior to returning from the function where a new handler was set.

4. **Code Example** // example for address of main()

```
// ch5_1.c

#include <stdio.h>

int main(int argc, char **argv) {
    fprintf(stderr, "Address of main is %#010x\n",
          main);
}

/***  OUTPUT
   $ make ch5_1   ## compile
   gcc    -o ch5_1 ch5_1.c

   $ ./ch5_1      ## run
   The address of main is 0x00010628
*/
```

5. **Code Example** // example for signal()

```
// ch5_2.c

#include <stdio.h>
#include <signal.h>
#include <errno.h>

void intr_handler(int sig) {
    fprintf(stderr, "\nHandling Signal: %d\n", sig);
}

int main(int argc, char **argv) {
    int c;

    signal(SIGINT, intr_handler); // set handler info

    fprintf(stderr, "Enter an input char:\n");
    while((c=getchar())!=EOF) { // loop until EOF
        putchar(c);
    }

    signal(SIGINT,SIG_DFL); // reset handler info

    fprintf(stderr, "EOF Entered:\n"); // end program
}
```

```
/***  OUTPUT

   $ make ch5_2             # Compile
   gcc    -o ch5_2 ch5_2.c

   $ ./ch5_2                # Run #1
   Enter an input char:
   a                        ### Enter 'a'
   a
   n                        ### Enter 'n'
   n
   ^C                       ### Enter ^C
   Handling Signal: 2
   EOF Entered:

   $ ./ch5_2                # Run #2
   Enter an input char:     ### Enter ^D
   EOF Entered:
*/
```

Notes

1. When executing a signal handler function for a signal, if a second occurrence of the signal is received, then the handler itself is interrupted.

   1.1. Therefore, to avoid interrupting the handler another call to `signal()` is needed, with SIG_IGN as its argument.

   1.2. It is usually the first step in the signal handler, when using `signal()` to set signal handler.

2. The signal handler is reset to default, just before the signal handler is called.

   2.1. Therefore, to respond to more than one instance of the signal, yet another call to `signal()` is needed each time the signal is handled.

   2.2. It is usually the last step in the signal handler, when using `signal()` to set signal handler.

3. What happens if a system call is interrupted?

   3.1. BACKGROUND:
   Normally, when a signal is received by a process, the thread of execution jumps to the interrupt handling routine and executes the code in the signal handler, and **resumes** execution at the next location where the thread of execution was, when the signal was received.

   3.2. This is true for system calls in BSD – system calls resume or restart, upon handling a signal. However, this is not true for system calls in SYS V - i.e. there is no restart .. by default, upon handling a signal.

   3.3. By default, when a system call is interrupted in SYS V, the system call does not complete, and instead it returns with an error status. `errno` is set to EINTR.

4. **Code Example** // example template .. `signal()` with SIG_IGN and signal handler.

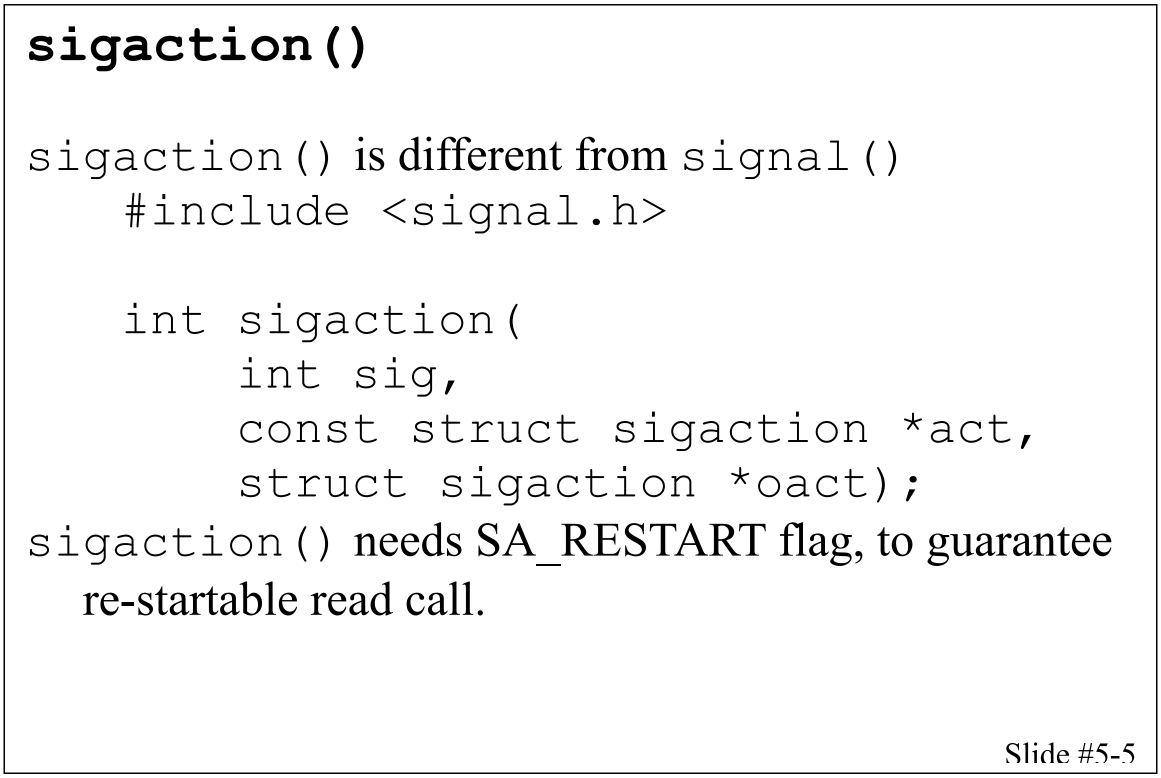
```
// ch5_3.c

#include <stdio.h>
#include <signal.h>
#include <errno.h>

void intr_handler(int sig) {
  // set mask for the current signal.
  signal(sig, SIG_IGN);

  // Handler code
  fprintf(stderr,"\nHandling Signal: %d\n",sig);

  // Set Signal Handler again ..
  signal(sig, intr_handler);
}

int main(int argc, char **argv) {
  int c;

  signal (SIGINT, intr_handler);   // set signal handler
  fprintf(stderr, "Enter an input char:\n");

  while((c=getchar())!=EOF) {     // loop until EOF
    putchar(c);
  }
  signal(SIGINT,SIG_DFL);       // restore signal info
  fprintf(stderr, "EOF Entered:\n");     // end program
}
```

```
/***  OUTPUT

   $ make ch5_3              # Compile
   gcc    -o ch5_3 ch5_3.c

   $ ./ch5_3               # Run #1
   Enter an input char:
   a                       ### Enter 'a'
   a
   n                       ### Enter 'n'
   n
   ^C                      ### Enter ^C
   Handling Signal: 2
   EOF Entered:

   $ ./ch5_3               # Run #2
   Enter an input char:    ### Enter ^D
   EOF Entered:
*/
```

5. **Code Example** // example template .. `signal()` with workaround for re-startable `read()` s.

```
// ch5_4.c

#include <stdio.h>
#include <signal.h>
#include <errno.h>

void intr_handler(int sig) {    // Handler code

  signal(sig, SIG_IGN);     // set mask for cur signal.
  fprintf(stderr,“\nHandling Signal: %d\n”,sig);

  signal(sig, intr_handler);  // Set Signal Handler again
}


int main(int argc, char **argv) {
   int c;

   signal(SIGINT, intr_handler);  // set signal handler
   fprintf(stderr, “Enter an input char:\n”);

   while(1) {                      // infinite loop
     errno=0;     // reset errno .. as not reset always!

     if((c=getchar())==EOF) {       // read next char

        if(errno==EINTR) {  // check, if intr by signal

          fprintf(stderr,“EOF and EINTR => SYSV\n”);
          fprintf(stderr,“Enter an input char:\n”);
          continue;
        }

        break;              //  ok .. a genuine EOF!
     }

     putchar(c);            // display char
   }

  signal(SIGINT,SIG_DFL);   // reset signal handler info
  fprintf(stderr, “EOF Entered:\n”);      // end program
}

/***  OUTPUT

   $ make ch5_4               # Compile
   gcc    -o ch5_4 ch5_4.c
   $ ./ch5_4                # Run #1
   Enter an input char:    ### Enter ‘a’
   a
   a
   ^C                      ### Enter ^C
   Handling Signal: 2
   EOF and EINTR => SYSV
   Enter an input char:    ### Enter ^C
   ^C
   Handling Signal: 2
   EOF and EINTR => SYSV
   Enter an input char:    ### Enter ^D
   EOF Entered:

*/
```

6. Chapter Reference: Robbins & Robbins #



sigaction()

`sigaction()` is different from `signal()`

```
    #include <signal.h>

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

`sigaction()` needs SA_RESTART flag, to guarantee re-startable read call.

Slide #5-5

Notes:

`signal()` is a *recent* addition to the POSIX standard.

`sigaction()` corrects two problems automatically.
The third problem .. re-startable read() gets corrected using the SA_RESTART flag.

```
struct sigaction {
  void      (*sa_handler)();
  sigset_t  sa_mask;    // list of other masked signals
  int       sa_flags;
}
```

`SA_RESTART` was non-POSIX till recent time. Therefore specifying _POSIX_SOURCE can remove SA_RESTART definition even on systems that provide SA_RESTART.

Therefore, you may need to add the following lines for those systems that do not restart read() by default.

1. Using the `sigset_t` data type

   1.1. A signal set was previously defined as an int datatype. The POSIX committee has changed the definition to `sigset_t`.

   1.2. The functions used to set and clear bits in the sigset_t type are:

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

2. **Code Example** // example template .. sigaction() without SA_RESTART.

```
// ch5_5.c

#define _POSIX_SOURCE 1

#include <stdio.h>
#include <signal.h>
#include <errno.h>

void intr_handler(int sig) {  // Signal Handler
  fprintf(stderr,“\nHandling Signal: %d\n”,sig);
}

int main(int argc, char **argv) {
  int c;
  struct sigaction oact, act;

  act.sa_handler = intr_handler; // sig handler info
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;

  sigaction(SIGINT, &act, &oact); // set handler

  fprintf(stderr, “Enter an input char:\n”);

  while(1) {        // infinite loop
    errno=0;        // reset errno .. always!

    if((c=getchar())==EOF) { // read next char

      if(errno==EINTR) {      // check, if intr by sig

                              // display messages
        fprintf(stderr, “EOF and EINTR => SYSV\n”);
        fprintf(stderr, “Enter an input char:\n”);
        continue;
      }
      break;        //  ok .. a genuine EOF!
    }
    putchar(c);     // display char
  }

  sigaction(SIGINT, &oact, 0); // reset handler info

  fprintf(stderr, “EOF Entered:\n”);    // end program
}
```

```
/***  OUTPUT

   $ make ch5_5               # Compile
   gcc    -o ch5_5 ch5_5.c

   $ ./ch5_5                 # Run #1
   Enter an input char:    ### Enter 'a'
   a
   a
   ^C                      ### Enter ^C
   Handling Signal: 2
   EOF and EINTR => SYSV
   Enter an input char:    ### Enter ^C
   ^C
   Handling Signal: 2
   EOF and EINTR => SYSV
   Enter an input char:    ### Enter ^D
   EOF Entered:

*/
```

3. **Code Example** // example template .. sigaction() using SA_RESTART

```
// ch5_6.c
#define _POSIX_SOURCE 1
#include <stdio.h>
#include <signal.h>
#include <errno.h>
#ifndef SA_RESTART
#   define SA_RESTART 0x00000004
#endif
void intr_handler(int sig) { // Handler code
    fprintf(stderr,"\nHandling Signal: %d\n",sig);
}
int main(int argc, char **argv) {
  int c;
  struct sigaction oact, act;
  act.sa_handler = intr_handler; // set handler info
  sigemptyset(&act.sa_mask);
  act.sa_flags = SA_RESTART;
  sigaction(SIGINT, &act, &oact); // set signal handler
  fprintf(stderr, "Enter an input char:\n");
  while((c=getchar())!=EOF) {
        putchar(c);
  }
  sigaction(SIGINT, &oact, 0);  // reset handler info
```

```
     fprintf(stderr, “EOF Entered:\n”); // end program
  }

  /***  OUTPUT
     $ make ch5_6              # Compile
     gcc    -o ch5_6 ch5_6.c

     $ ./ch5_6               # Run #1
     Enter an input char:
     a                       ### Enter ‘a’
     a
     n                       ### Enter ‘n’
     n
     ^C                      ### Enter ^C
     Handling Signal: 2
     a                       ### Enter ‘a’
     a
     n                       ### Enter ‘n’
     n
     EOF Entered:            ### Enter ^D
  */
```

4. **Code Example** // example template .. sigaction() using SA_RESTART and sa_mask

```
// ch5_7.c

#define _POSIX_SOURCE 1

#include <stdio.h>
#include <signal.h>
#include <errno.h>

#ifndef SA_RESTART
#   define SA_RESTART 0x00000004
#endif

void intr_handler(int sig) {  // Handler code
    fprintf(stderr,“\nHandling Signal: %d\n”,sig);
}

int main(int argc, char **argv) {

  int c;
  struct sigaction oact, act;

  act.sa_handler = intr_handler; // set handler info
  sigemptyset(&act.sa_mask);
  sigaddset(&act.sa_mask, SIGTSTP);
  act.sa_flags = SA_RESTART;

  sigaction(SIGINT, &act, &oact);  // set signal handler

  fprintf(stderr, “Enter an input char:\n”);

  while((c=getchar())!=EOF) {  // loop till EOF
```

```
        putchar(c);
    }
    sigaction(SIGINT, &oact, 0); // reset signal info

    fprintf(stderr, “EOF Entered:\n”); // end program

}

/***  OUTPUT

    $ make ch5_7                # Compile
    gcc    -o ch5_7 ch5_7.c

    $ ./ch5_7                  # Run #1
    Enter an input char:
    a                          ### Enter ‘a’
    a
    n                          ### Enter ‘n’
    n
    ^C                         ### Enter ^C
    Handling Signal: 2
    a                          ### Enter ‘a’
    a
    n                          ### Enter ‘n’
    n
    EOF Entered:               ### Enter ^D

*/
```

sigprocmask()

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

`sigprocmask()` blocks signals for the whole process.

Slide #5-6

Notes

1. `sigprocmask()` blocks signals for the whole process.

2. In contrast, the `sa_mask` field in `sigaction()` blocks signals while a particular handler is executing.

3. Using the sigset_t datatype (repeated from earlier):

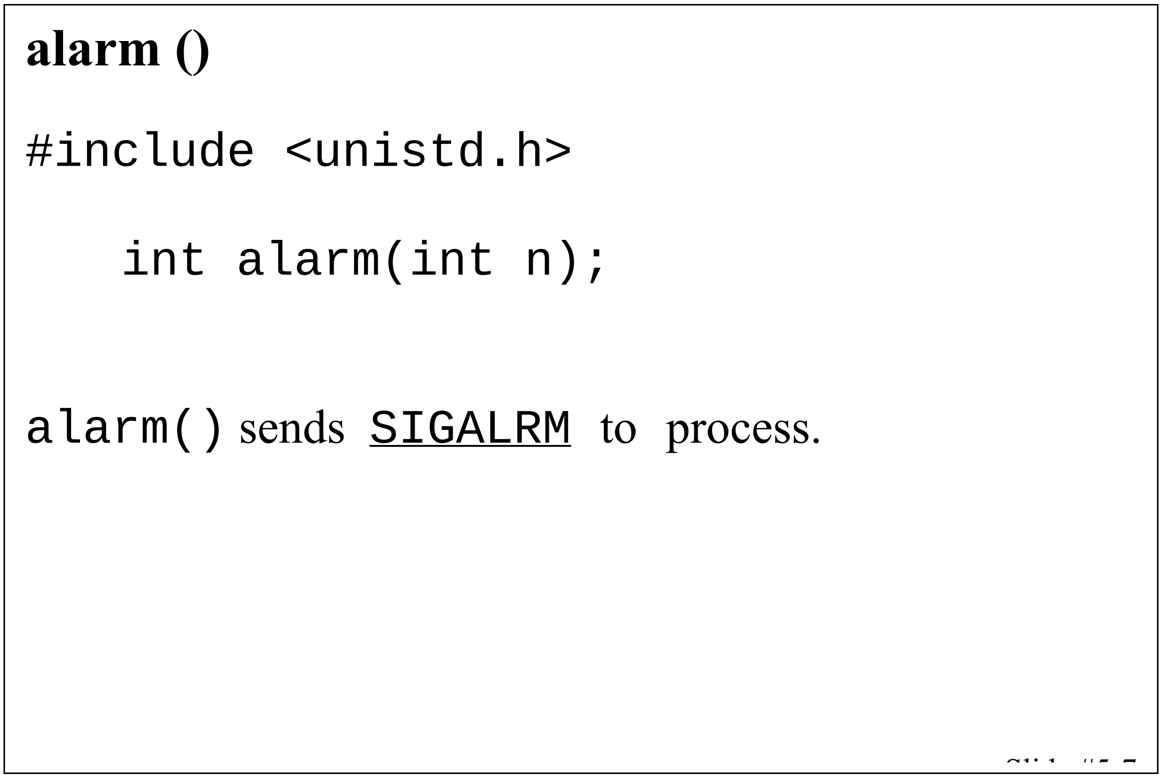
   3.1. The functions used to set and clear bits in the sigset_t type are:

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

4. **Code Example** // example template .. critical section using sigprocmask()

```
// ch5_8.c

#define  TMPFILNAME "mytempfile_XXXXXX"

#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

char *ProgName;
char *FuncName;

int my_mvcmd(char *ofn, char *fn, char *buf);
int my_cpcmd(char *ofn, char *fn, char *buf);

int main(int argc, char **argv) {

   int  status;
   char *ofn;
   char *fn;
   char errbuf[1024];

   ProgName=argv[0];

     if (argc!=3) {
        fprintf(stderr,"\nUsage: %s <src> <dest>\n",
          ProgName);
        exit(1);
     }

    ofn=argv[1];
    fn=argv[2];

    if ((status=my_mvcmd(ofn,fn,errbuf))==0)
         fprintf(stderr,"%s:'%s' renamed to '%s'.\n",
             ProgName,ofn, fn);
    else {
        fprintf(stderr,"%s: ‘%s’ rename failed:%s:#:\n",
             ProgName, ofn, errbuf);
         exit(1);
    }

    exit(0);
}

int my_mvcmd(char *ofn, char *fn, char *erbuf) {

   char       rbuf[1024];
   char       tmpnambuf[1024];
   char       *tfn; // generated name used by temp file

   int     st=0;
   sigset_t   oset;
   sigset_t   set;

   FuncName="my_mvcmd";

   sigemptyset(&set);

   sigaddset(&set, SIGHUP);  // hangup
   sigaddset(&set, SIGINT);  // interrupt .. ^C
   sigaddset(&set, SIGQUIT); // quit .. ^\
   sigaddset(&set, SIGTERM); // kill
   sigaddset(&set, SIGALRM); // alarm
   sigaddset(&set, SIGTSTP); // terminal stp ^Z

   if (access(fn,F_OK) == 0) {
      sprintf(erbuf,"%s.%s: '%s' rename failed:%s::",
        ProgName,FuncName,ofn,"File Exists");
      return -1;
    }
    sigprocmask(SIG_BLOCK, &set, &oset);
    //   begin critical section
    if (access(ofn,F_OK) == 0) { // is file accessible?
        sprintf(tmpnambuf,"%s",TMPFILNAME);

        tfn=mktemp(tmpnambuf);  // tmp name generation:
                // Step #2 of 2: use a generated suffix

        if ((st=rename(ofn,tfn))!= 0) { // rename() .. mv
          sprintf(erbuf,"%s.%s:'%s' rename failed:%s::",
            ProgName,FuncName,ofn, strerror(errno));
          return -1;
        }

        if ((st=my_cpcmd(tfn,fn,rbuf))!=0) { // now copy
          sprintf(erbuf,"%s.%s:'%s' copy failed:%s::",
            ProgName,FuncName,ofn, strerror(errno));

          // Ok .. copy failed .. time to undo

          // remove reference to newfile .. in case,
          // it got generated above.
          unlink(fn);

          // now try to restore back
          if ((st=rename(tfn,ofn)) != 0) {
            sprintf(erbuf,"%s.%s:'%s' restore fail:%s:",
              ProgName,FuncName,ofn, strerror(errno));
           }

          return -1;
        }
        unlink(tfn);   // remove temp file
    }

  // end critical section
```

```
   sigprocmask(SIG_BLOCK, &oset, 0);

   return(0);
}

int my_cpcmd(char *ofn, char *fn, char *erbuf) {

   FILE *rfp, *wfp;
   int  st,c;
   FuncName="my_cpcmd";

   if ((rfp=fopen(ofn,"r"))==NULL) { // open src for read

      // Format error message for open() fail
      sprintf(erbuf,"%s.%s:read: open '%s' fail:%s:",
         ProgName,FuncName,ofn, strerror(errno));

      return -1;

   }

   if ((wfp=fopen(fn,"w"))==NULL) { //open dest for write

      // Format Meaningful error message for open() fail
      sprintf(erbuf,"%s.%s:write: open '%s' fail:%s",
         ProgName,FuncName,fn, strerror(errno));
      return -1;
   }
   errno=0;       // reset errno

   while((c=getc(rfp))!=EOF) {        // get next char

      errno=0;                        // reset errno

      putc(c,wfp);                    // putc

      if (errno!=0) {
         sprintf(erbuf,"%s.%s:write:write '%s' fail:%s:",
              ProgName,FuncName,fn, strerror(errno));
         return -1;
      }
   }

   if (errno!=0) {
      sprintf(erbuf,"%s.%s:read: read '%s' fail:%s:",
         ProgName,FuncName,ofn, strerror(errno));
      return -1;
   }
   fclose(rfp);
   fclose(wfp);

   return 0;
}
```



alarm ()

```
#include <unistd.h>

    int alarm(int n);
```

`alarm()` sends <u>`SIGALRM`</u> to process.

Notes

1. #include <unistd.h>
   int alarm (int n);
   a. The system call alarm() causes the signal `SIGALRM` to the process after n seconds.
   b. The return value is the number of seconds left
   c. Subsequent calls to alarm() will reset the alarm

2. **Code Example** //example for alarm()

```
//ch5_9.c

#define  _POSIX_SOURCE   1
#define  ALARMTMOUT      5

#include <stdio.h>

int main() {

   int c;
   alarm(ALARMTMOUT);

   fprintf(stderr, "Enter Input:(in '%d' secs)\n",
       ALARMTMOUT);

   while ((c=getchar()) != EOF) {
      putchar(c);
   }

   // Alarm time did not expire.
   // If alarm time expired, the default signal
   // handler action would have caused the program
  // to exit.

   alarm(0); // cancel any previously set alarms

   //   Notify .. Exit
      fprintf(stderr, "EOF received. Exiting!\n");
}

   /***  OUTPUT
     $ make ch5_9                ## compile
     gcc    -o ch5_9 ch5_9.c

     $ ./ch5_9                ## run #1
     Enter Input:(in '5' secs)
      a                          ### Enter ‘a’
      a                          ### displayed
      ^C                         ### Enter ‘^C’
                                     ### program exit

     $ ./ch5_9                ## run #2
     Enter Input:(in '5' secs)
      a                          ### Enter ‘a’
      a                          ### displayed
      ^D                         ### Enter ‘^D’
      EOF received. Exiting!     ### program exit

     $ ./ch5_9                     ## run #3
     Enter Input:(in '5' secs)
      a                          ### Enter ‘a’
      a                          ### displayed
      Alarm Clock                ### Wait 5 secs
   */
```

pause ()

process blocks until any signal is delivered.

```
void pause(void);
```

Slide #5-8

Notes

1. pause( ) will cause the process to block unit **any** signal is delivered.
2. sigpause() waits for a particular signal.

sleep()

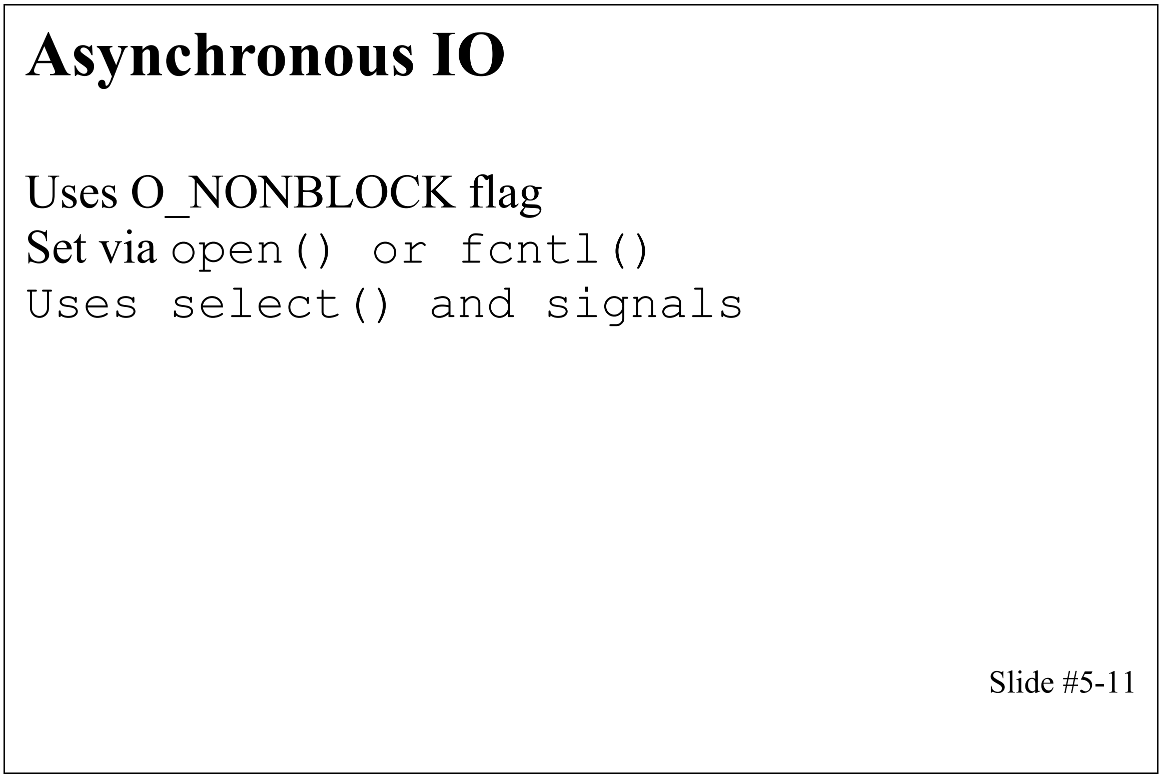
Typically, implemented
- the combination of `alarm()` and `pause()`.

Slide #5-9

Notes

Some implementations of sleep() can use a combination of alarm() and pause().

```
// ch5_10.c

#define  _POSIX_SOURCE   1
#define  ALARMTMOUT      5
#include <stdio.h>
#include <signal.h>

#ifndef SA_RESTART
#   define SA_RESTART 0x00000004
#endif

void alarm_handler(int sig) {
   fprintf(stderr, "BUZZ! Got Sig '%d'\n",sig);
}

int main() {

   int c;
   struct sigaction oact, act;
   int r;   // alarm_time_remaining

   // set new handler info in struct sigaction
   act.sa_handler=alarm_handler;
   sigemptyset(&act.sa_mask);
   act.sa_flags=SA_RESTART;

   // notify kernel on handler for SIGALRM
   sigaction(SIGALRM, &act, &oact);

   // alarm() returns time remaining on
   //   current alarm (if any)
   r=alarm(ALARMTMOUT);

   // ZZZ .. process is now sleeping!
   // - No CPU resources are consumed.

   // pause() .. returns when any signal
   //            is received

   pause();

   fprintf(stderr, "pause() returned!\n");

   // Alarm time expired.
   // Reset old alarm time expired.
   alarm(r);  // alarm_time_remaining
   sigaction(SIGALRM, &oact, 0);  // reset handler

}


/***  OUTPUT
   $ make ch5_10                ## compile
   gcc    -o ch5_10 ch5_10.c

   $ ./ch5_10                ## run #1
   BUZZ! Got Sig '14'
   pause() returned!
*/
```



Asynchronous IO

Uses O_NONBLOCK flag
Set via `open() or fcntl()`
`Uses select() and signals`

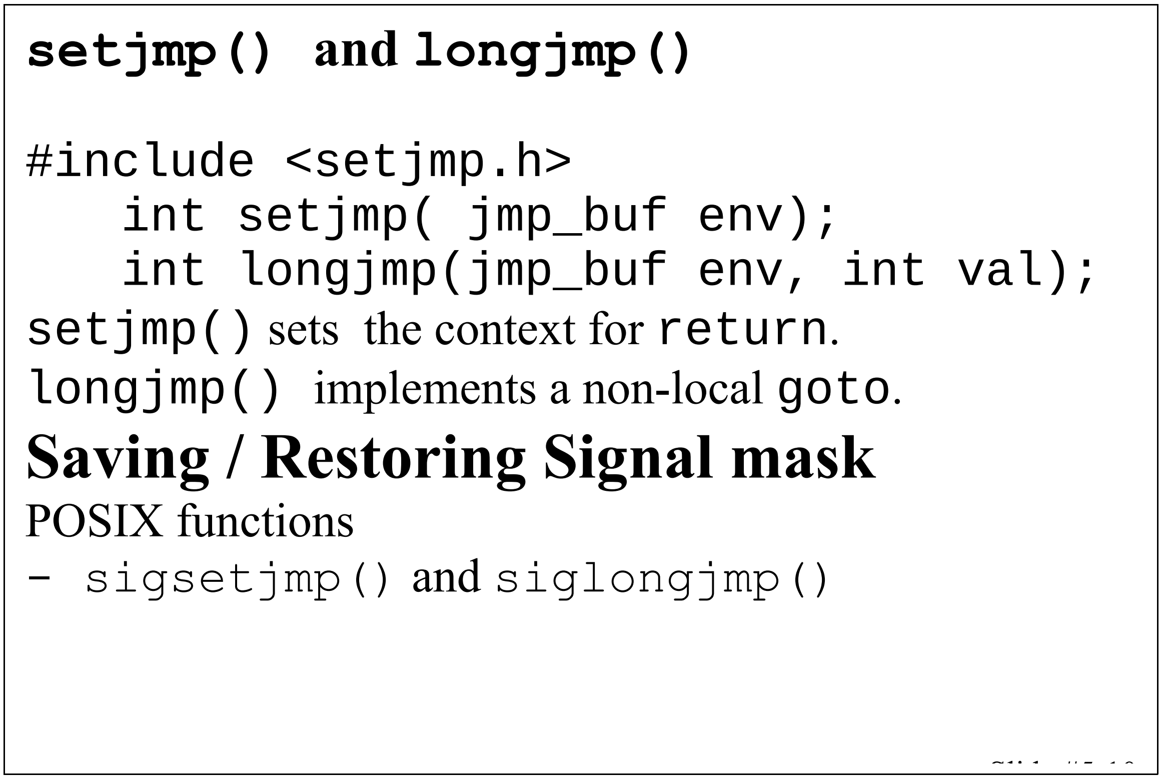
Slide #5-11

Notes

1. Allows IO access on multiple file descriptors

2. Code Example

```
#define _POSIX_SOURCE 1

#define uses_SYSV_sigpoll

#define ERRBUFSZ 512

#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <fcntl.h>

#ifdef _POSIX_SOURCE
      #define SA_RESTART 0x000004
#endif

#ifdef uses_SYSV_sigpoll
      #include <sys/types.h>
      #include <stropts.h>
      #include <sys/conf.h>
#endif

int init_async_IO(int fd, void(*handler_fn)(), char * errbuf);
static void sigio_handler(void);
static void sigio_handler() {

      int c;

      while (1) {

            errno=0;

            if ((c=getchar())==EOF) {
                  return;
            }

            putchar(c);
      }
}

#ifdef uses_SYSV_sigpoll
      int init_async_IO(int fd, void(*handler_fn)(), char *errbuf) {

            int st;

            struct sigaction nsigaction;

            nsigaction.sa_handler=handler_fn;
            sigemptyset(&nsigaction.sa_mask);
            nsigaction.sa_flags=SA_RESTART;

            sigaction(SIGPOLL,&nsigaction,0);

            st=ioctl(fd,I_SETSIG,S_RDNORM);

            if (st !=0) {
                  sprintf(errbuf,"ERR: ioctl (%d, I_SETSIG): %s",
                        fd, strerror(errno));
                  return -1;
            }

            return 0;
      }

#else
      int init_async_IO(int fd, void(*handler_fn)(), char *errbuf) {

            int st;
            int fl;

            struct sigaction nsigaction;

            nsigaction.sa_handler=handler_fn;
            sigemptyset(&nsigaction.sa_mask);
            nsigaction.sa_flags=SA_RESTART;

            sigaction(SIGIO,&nsigaction,0);

            fl=fcntl(fd,F_GETFL,0);
            if (fl !=0) {
                  sprintf(errbuf,"ERR: fcntl (%d, F_GETFL): %s",
                        fd, strerror(errno));
                  return -1;
            }

            fl |= FASYNC;

            st=fcntl(fd,F_SETFL,fl);

            if (st !=0) {
                  sprintf(errbuf,"ERR: fcntl (%d, F_SETFL): %s",
                        fd, strerror(errno));
                  return -1;
            }

            st=fcntl(fd,F_SETOWN,getpid()); /* give your pid to device driver
.. to  tell which process should receive signal */

            if (st !=0) {
                  sprintf(errbuf,"ERR: fcntl (%d, F_SETOWN): %s",
                        fd, strerror(errno));
                  return -1;
            }

            return 0;
      }

#endif

int main(int argc, char **argv) {

      int i;

      int st;
      char *errbuf=(char *)malloc(ERRBUFSZ*sizeof(char));

      /*
            st=init_nblock_IO(STDIN_FILENO, errbuf);
            if (st !=0) {
                  fprintf(stderr,"ERR: init_nblock_IO: %s\n", errbuf);
                  return -1;
            }
      */

      st=init_async_IO(STDIN_FILENO, sigio_handler, errbuf);
      if (st !=0) {
            fprintf(stderr,"ERR: init_async_io: %s\n", errbuf);
            return -1;
      }

      for (i=0;i<5;i++) {
            printf("Hello! from '%s' pid=%d\n",argv[0],getpid());
            sleep(5);
      }

      exit (0);
}
```



`setjmp()` and `longjmp()`

```
#include <setjmp.h>
   int setjmp( jmp_buf env);
   int longjmp(jmp_buf env, int val);
```

`setjmp()` sets the context for `return`.

`longjmp()` implements a non-local `goto`.

Saving / Restoring Signal mask

POSIX functions

- `sigsetjmp()` and `siglongjmp()`

Notes

1. Both `setjmp()` and `longjmp()` are std library functions

1. `setjmp()`

   a. sets up the return context,
   b. initializes the `jmp_buf` argument and
   c. returns 0.

2. longjmp()

   d. `longjmp()` uses the return context in the `jmp_buf` argument, and resumes execution inside the call to setjmp().
   e. jmp_buf is an array that contains the processors register and the return context..
   f. A successful call to long_jmp() returns within setjmp().

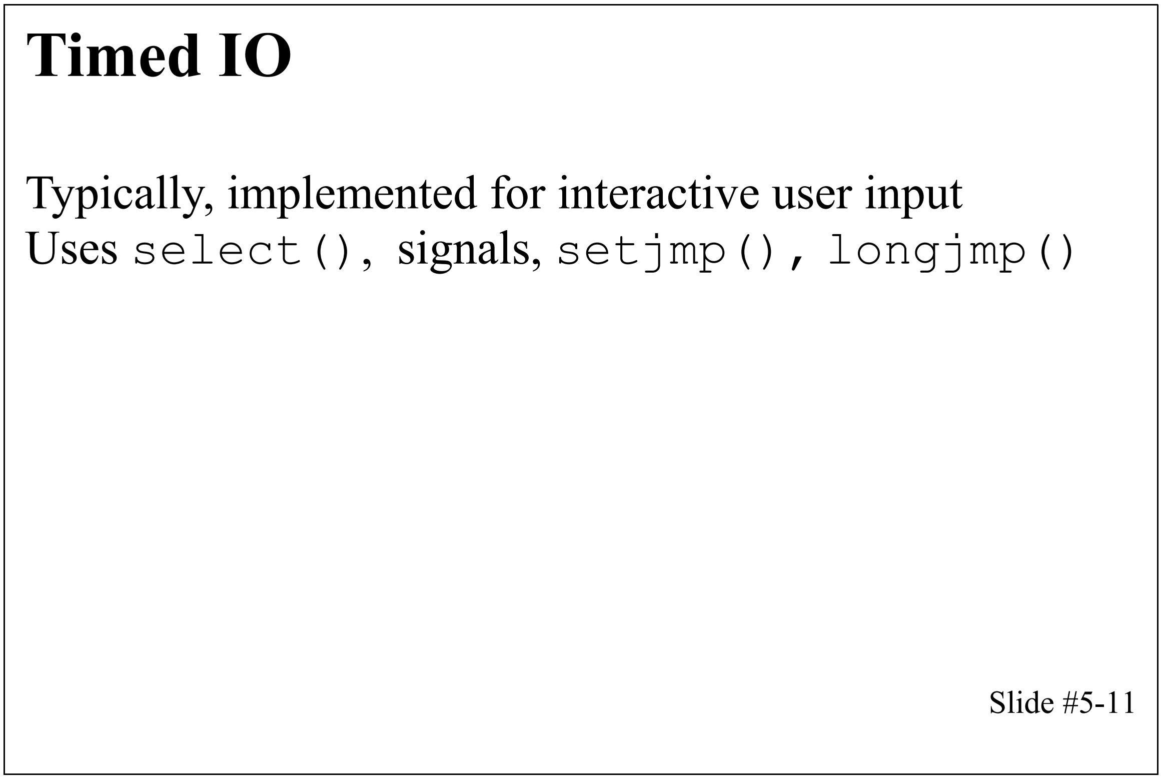
Notes

1. `setjmp()` under BSD saves CPU registers and current signal mask. The signal mask and CPU registers are restored by `longjmp()` to be the same as that at the time of `setjmp`.

2. Under system V, the signal mask remains as at the time of `longjmp()`.

3. **Code example 1.1.3**

```
// ch5_11.c
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

// subroutine function  ... uses longjmp()
extern void fn_a (void);

// StackFrame to save "context info"
static jmp_buf Buf;  // to store return context

int main() {

   if (setjmp(Buf) == 0) {
   fprintf(stderr,"setjmp..returning once!\n");
   fn_a();
   }
   else {

// setjmp() returns as a result of
//    calling longjmp()
fprintf(stderr,"setjmp..returning twice!\n");
exit(0);
}

// Code is never reached!
fprintf(stderr, "Hello World!\n");

}

void fn_a(void) {

// ok .. we get to call longjmp()
longjmp(Buf,1);

// Code is never reached!
fprintf(stderr, "Hello World!\n");

}

/***  OUTPUT
$ make ch5_11                ## compile
gcc    -o ch5_11 ch5_11.c

$ ./ch5_11                ## run #1
setjmp..returning once!
setjmp..returning twice!
*/
```



Timed IO

Typically, implemented for interactive user input
Uses `select()`, signals, `setjmp()`, `longjmp()`

Slide #5-11

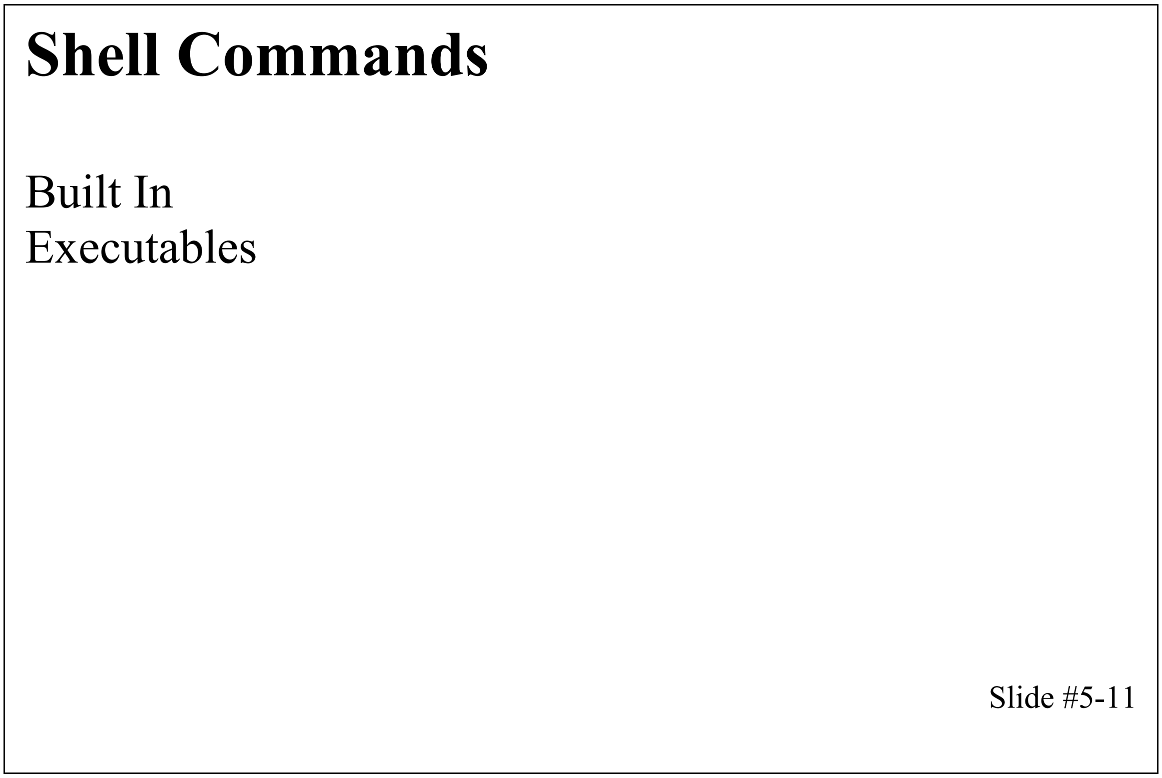
Notes

1. Used in interactive user programming.
2. Code Example

```
#define _POSIX_SOURCE 1

#define BUFSZ 512
#define ERRBUFSZ 512
#define TIMEOUT_TIMEDIO 5

#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <fcntl.h>
#include <setjmp.h>

#ifdef _POSIX_SOURCE
      #define SA_RESTART 0x000004
#endif

static sigjmp_buf jmpbuf;

int timed_io(char *buf, int len, FILE *rfp, int sec);
static void sigalrm_handler(int signo);

static void sigalrm_handler(int signo) {
      siglongjmp(jmpbuf,17);
}
```

```
int timed_io(char *buf, int len, FILE *rfp, int sec) {

      struct sigaction nsigaction[1];
      struct sigaction osigaction[1];

      int prev_alrm;
      int st=0;   /* if st == -1 at EOF; st == -2 Time Out reached; */

      if (sigsetjmp(jmpbuf,1) == 0) {
            nsigaction->sa_handler=sigalrm_handler;
            sigemptyset(&nsigaction->sa_mask);
            nsigaction->sa_flags = SA_RESTART;

            prev_alrm=alarm(0);
            sigaction(SIGALRM,nsigaction,osigaction);

            alarm(sec);

            if (fgets(buf,len,rfp)==NULL) {
                  st=-1;
            }
            buf[strlen(buf)-1]=0;
      }
      else {
            st=-2;
      }

      alarm(0);   /* reset old alarm and handler */
      sigaction(SIGALRM,osigaction,0);
      alarm(prev_alrm);

      return st;
}

int main(int argc, char **argv) {
      int st;
      int sec=TIMEOUT_TIMEDIO;
      char *buf=(char *)malloc(BUFSZ*sizeof(char));
      char *errbuf=(char *)malloc(ERRBUFSZ*sizeof(char));

      fprintf(stderr,"Enter Input (%d sec):",TIMEOUT_TIMEDIO);

      st=timed_io(buf, BUFSZ, stdin, sec);
      if (st !=0) {
            fprintf(stderr,"ERR: No Input: %s (Status=%d)\n", errbuf,st);
            return -1;
      }
      else {
            fprintf(stderr,"Input Buf=\"%s\" (Len=%d).\n",buf, strlen(buf));
      }

      exit(0);
```

}



Shell Commands

Built In
Executables

Slide #5-11

Notes

1. Shell Builtin Commands
2. Executable Commands
3. Code Example

```
#define _POSIX_SOURCE 1

#define BUFSZ 512
#define ERRBUFSZ 512

#define MYSH_PROMPT "mysh> "

#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

#include <string.h>

void do_cmd(char *buf, int len, int linenum, char *errbuf);
int parse_cmd(char *buf, char **vbuf, char *errbuf);
int builtin_cmd(char **argv, int linenum);
int process_cmd(char **argv, int linenum);
int printwaitstatus(FILE *wfp, int pid, int st);

int printwaitstatus(FILE *wfp, int pid, int st) {

      fprintf(wfp,"\n");
```

```
        fprintf(wfp,"%6d=wait()",pid);

        if (WIFEXITED(st)) {
              fprintf(wfp,"exit: %3d\n", WEXITSTATUS(st));
        }
        else if (WIFSTOPPED(st)) {
              fprintf(wfp,"stop status: %3d\n", WSTOPSIG(st));
        }
        else if (WIFSIGNALED(st)) {
              fprintf(wfp,"termination signal: %3d\n", WTERMSIG(st));

        // fprintf(wfp,"\tcore dump: %s\n", WIFCORE(st) ? "yes" : "no");
        }

        return 0;
}


int builtin_cmd(char **argv, int linenum) {

        int st;

        if (strcmp(*argv,"exit") == 0) {
              exit(0);
        }
        else if (strcmp(*argv,"cd") ==0) {
              if ((argv[1]) && (st=chdir(argv[1])) != 0) {
                    fprintf(stderr,"ERR: \"cd\" to '%s' failed! (Line=%d)\n",
                          argv[1],linenum);
                    return -1;
              }
              return 1;
        }
        else if (strcmp(*argv,"hello") ==0) {
                    fprintf(stderr,"\nHello! from process '%d'. (Line=%d)\n",
                          getpid(),linenum);
              return 1;
        }

        return 0;
}

int process_cmd(char **argv, int linenum) {

        pid_t cpid=fork();

        if (cpid<0) {
              fprintf(stderr,"ERR: \"fork\" error! (Line=%d)\n",
                    linenum);
              exit (-1);
        }
        else if (cpid==0) {
              if (execvp(argv[0],argv) < 0) {
                    fprintf(stderr,"ERR: \"execv(%s)\" error! (Line=%d)\n",
                    argv[0], linenum);
                    _exit (errno);
              }
        }
        else {
              int st;

              cpid=wait(&st);
              // printwaitstatus(stdout,cpid,st);
        }
  }

  int parse_cmd(char *buf, char **vbuf, char *errbuf) {

        int i=0;

        char *delim=" ,\t\n";

        char *tok;

        tok=strtok(buf,delim);

        while (tok) {
              vbuf[i]=(char *)malloc(BUFSZ*sizeof(char));

              strcpy(vbuf[i],tok);

              tok=strtok(NULL,delim);
              i++;
        }

        vbuf[i]=0;

        return i;

  }


  void do_cmd(char *buf, int len, int linenum, char *errbuf) {

        int i=0;
        char *vbuf[128];

        int maxargs=sizeof(vbuf)/sizeof(char *);
        int numargs;

        if ((numargs=parse_cmd(buf,vbuf,errbuf))==maxargs) {
              fprintf(stderr,"ERR: too many args (Line=%d)\n",linenum);
        }
        else {
              if (!builtin_cmd(vbuf,linenum) ) {
                    process_cmd(vbuf,linenum);
              }
        }

        for (i=0;i<numargs; i++) {
              free(vbuf[i]);
        }

        return;
  }

  int main(int argc, char **argv) {
  	int i;

  	int st;

  	int linenum=0;

  	char *buf=(char *)malloc(BUFSZ*sizeof(char));
  	char *errbuf=(char *)malloc(ERRBUFSZ*sizeof(char));

  	char *mysh = MYSH_PROMPT;

  	FILE *rfp=stdin;

  	if (isatty(fileno(rfp))) {
  		mysh="mysh> ";

  		fprintf(stderr,"%s",mysh);
  	}

  	while (fgets(buf,BUFSZ,rfp)) {
  		linenum++;

  		buf[strlen(buf)-1]=0;

  		if (*buf)
  			do_cmd(buf, BUFSZ, linenum,errbuf);

  		if (mysh)
  			fprintf(stderr,"%s",mysh);
  	}
  }
```

1. Links

   a) http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/signals.html

   b) http://users.actcom.co.il/~choo/lupg/tutorials/multi-thread/multi-thread.html

   c) http://www.linux-tutorial.info/modules.php?name=Tutorial&pageid=289

   d) http://www.advancedlinuxprogramming.com/alp-folder

   e) http://www.comptechdoc.org/os/linux/programming/linux_pgsignals.html

   f) http://pauillac.inria.fr/~xleroy/linuxthreads/faq.html

   g) http://rtportal.upv.es/apps/rtl-signals/posix_signals-0.1/signals-info.pdf

   h) http://rtportal.upv.es/apps/rtl-signals/

   i) http://www.llnl.gov/computing/tutorials/pthreads/

   j) http://www.mvista.com/dswp/wp_rtos_to_linux.pdf

Chapter 5

Signals

Terms & Concepts Worksheet

Table #1 ("Basic")

1. Signals.	Provides asynchronous event notification. .. For both, hardware and software events. .. Every signal has a name. .. Signal information is stored in the u_area.

Table #2 ("Signals")

2. Signals	a. Signal Delivery .. by the kernel b. Signal Handling .. by the process .. "what action to take?"
3. Signal Handling	A process can a. Ignore the signal .. b. Handle the signal .. or, c. take "Default Action" for signal .. .. exit, .. exit with core, .. stop or suspend processing, .. ignore.

4. `signal()`	`oldfunction = signal(int sig, void (*function) (int));`
	
	`#include <signal.h>`
	`void (*signal (int sig, void (*disp)(int)))(int);`

5. `sigaction ()`	signal() .. implemented differently in BSD and SYSV .. causes different behavior. In traditional SYS V a. after handling, signal handler information resets signal handler info to default .. therefore does not allow handling same signal again. b. Signal Masking .. if same signal or other signals are delivered during handler execution, there is no ability to mask signals .. same or different c. `read()` system call does not restart automatically upon handling a signal, unlike BSD. Therefore, `sigaction()` a. after handling, signal handler information allows handling same signal again. b. sa_mask .. Signal Masking .. if same signal or other signals are delivered during handler execution. c. provides a flag SA_RESTART that allows system calls to restart, upon handling a signal. sigaction() behavior is POSIX.

6. `sigprocmask()`	Blocks signals for a whole process.
	`#include <signal.h>`
	`int sigprocmask(int how,const sigset_t *set,sigset_t *oset);`

Table #3 ("alarm() and pause()")

7. alarm()	a. unsigned int tmleft=alarm(int sec) b. The system call alarm() causes SIGALRM to be sent after sec seconds. c. Default behavior for SIGALRM is exit.
8. pause()	a. #include <unistd.h> int pause(void); b. Suspend process until any signal is received. c. sigpause() waits for a particular signal.
9. sleep()	a. is implemented as a combination of alarm() and pause()

Table #4 ("setjmp() and longjmp() ")

10. setjmp() longjmp()	a. provides a mechanism to implement "Non-Local Goto" b. setjmp() .. sets stack frame and return address. c. longjmp() .. returns using return address set up in setjmp(). Again, BSD and SYS V implementations vary .. .. BSD saves CPU registers, current signal mask at setjmp() and, restores at longjmp(), but, .. SYSV retains CPU registers, current signal mask settings at longjmp(). POSIX provides following functions that save and restore the signal mask. a. sigsetjmp() .. sets stack frame and return address. b. siglongjmp() .. returns using return address set up in setjmp().

Chapter 5

Signals

Assignment Questions

Questions:

5.1 Modify "mysh" to implement timed IO capability. (Default Time: 15 secs)

5.2 Modify "mysh" to include builtin commands to call the version of 'mywc', 'mycat' and 'myls' from previous chapter(s).

Chapter 5

Signals

Useful Links

a) http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/signals.html
b) http://users.actcom.co.il/~choo/lupg/tutorials/multi-thread/multi-thread.html
c) http://www.linux-tutorial.info/modules.php?name=Tutorial&pageid=289
d) http://www.advancedlinuxprogramming.com/alp-folder
e) http://www.comptechdoc.org/os/linux/programming/linux_pgsignals.html
f) http://pauillac.inria.fr/~xleroy/linuxthreads/faq.html
g) http://rtportal.upv.es/apps/rtl-signals/posix_signals-0.1/signals-info.pdf
h) http://rtportal.upv.es/apps/rtl-signals/
i) http://www.llnl.gov/computing/tutorials/pthreads/
j) http://www.mvista.com/dswp/wp_rtos_to_linux.pdf