Objectives

- Ideas and Skills
  - Program driven by asynchronous events
  - The curses library: purpose and use
  - Alarms and interval timers
  - Reliable signal handling
  - Reentrant code, critical section
  - Asynchronous input
- System calls and Functions
  - alarm, setitimer, getitimer
  - kill, pause
  - sigaction, sigprocmask
  - fcntl, aio_read

How does a game work

- Space: Screen display management
- Time: How to track time?
- Interruptions: How does a game respond to interruptions?
- Doing several things at the same time

The project: Write single-player pong

Show sample program pong_curses.

Space programming: the curses library

The curses library is a set of functions for controlling the cursor and text display on a terminal screen. Below are nine basic curses functions

Basic Curses Functions	
initscr()	Initializes the curses library and the tty
endwin()	Turns off curses and resets the tty
refresh()	Makes screen look the way you want
move(r,c)	Moves cursor to screen position
addstr(s)	Draws string s on the screen at current position
addch(c)	Draws char c on the screen at current position
clear()	clears the screen
standout()	Turns on standout mode (usually reverse video)
standend()	Turns off standout mode

Curses Example 1: hello1.c

```
/* hello1.c
 *     purpose show the minimal calls needed to use curses
 *     outline  initialize, draw stuff, wait for input, quit
 */

#include        <stdio.h>
#include        <curses.h>

main()
{
        initscr() ;             /* turn on curses */

                                /* send requests */
        clear();                /* clear screen */
        move(10,20);                    /* row10,col20 */
        addstr("Hello, world");         /* add a string */
        move(LINES-1,0);                /* move to LL */

        refresh();              /* update the screen */
        getch();                /* wait for user input */

        endwin();               /* turn off curses */
}
```

Curses Example 2: hello2.c

```
/* hello2.c
 *      purpose show how to use curses functions with a loop
 *      outline initialize, draw stuff, wrap up
 */
#include        <stdio.h>
#include        <curses.h>
main()
{
        int    i;
        initscr();                      /* turn on curses */
        clear();                        /* draw some stuff */
        for(i=0; i<LINES; i++ ){        /* in a loop */
                move( i, i+i );
                if ( i%2 == 1 )
                        standout();
                addstr("Hello, world");
                if ( i%2 == 1 )
                        standend();
        }
        refresh();                      /* update the screen */
        getch();                        /* wait for user input */
        endwin();                       /* reset the tty etc */
}
```

Curses Internals: Virtual and Real screens

- Curses has two internal versions of the screen
  - One is a copy of the real screen
  - One is a workspace
- refresh is called, curses compares the work place with the copy of the real screen and send the change to the screen driver. At the same time, the internal copy of real screen is also updated.

Why does curses handle screen output this way?

Time programming: sleep

Animation example 1: hello3.c

```
/* hello3.c
 *      purpose using refresh and sleep for animated effects
 *      outline initialize, draw stuff, wrap up
 */
#include        <stdio.h>
#include        <curses.h>
main()
{
        int  i;
        initscr();  clear();
        for(i=0; i<LINES; i++ ){
                move( i, i+i );
                if ( i%2 == 1 )
                        standout();
                addstr("Hello, world");
                if ( i%2 == 1 )
                        standend();
                sleep(1);  refresh();
        }
        endwin();
}
```

Time programming: sleep

Animation example 2: hello4.c

```
/* hello4.c
 *     purpose show how to use erase, time, and draw for animation
 */
#include        <stdio.h>
#include        <curses.h>
main()
{
        int   i;
        initscr();  clear();
        for(i=0; i<LINES; i++ ){
                move( i, i+i );
                if ( i%2 == 1 ) standout();
                addstr("Hello, world");
                if ( i%2 == 1 ) standend();
                refresh(); sleep(1);
                move(i,i+i);                    /* move back */
                addstr("             ");        /* erase line */
        }
        endwin();
}
```

Time programming: sleep

Animation example 3: hello5.c

```
/* hello5.c
 *    purpose  bounce a message back and forth across the screen
 *    compile  cc hello5.c -lcurses -o hello5
 */
#include       <curses.h>
#define LEFTEDGE        10
#define RIGHTEDGE       30
#define ROW             10

main()
{
        char    message[] = "Hello",  blank[]   = "     ";
        int     dir = +1,  pos = LEFTEDGE ;

        initscr();  clear();
        while(1){
                move(ROW,pos);  addstr( message );   /* draw string            */
                move(LINES-1,COLS-1);                /* park the cursor        */
                refresh();                           /* show string            */
                sleep(1);  move(ROW,pos);
                addstr( blank );                     /* erase string           */
                pos += dir;                          /* advance position       */
                if ( pos >= RIGHTEDGE )              /* check for bounce       */
                        dir = -1;
                if ( pos <= LEFTEDGE )
                        dir = +1;
        }
}
```

Problems in the programs we have now

- The minimum delay using sleep is 1 second, which is too long
- Unable to handle user input

Solution

- use Alarms
- advanced signal handling

How sleep works

sleep can actually be implemented by SIGALARM, sent by the alarm clock of a process.

```
signal(SIGALARM, handler);
alarm(n);
pause();
```

Warning of the sleep manpage

```
sleep()  may  be implemented using SIGALRM; mixing calls to alarm() and sleep()
is a bad idea.
```

How sleep works

```
/* sleep1.c
 *      purpose show how sleep works
 *      usage sleep1
 *      outline sets handler, sets alarm, pauses, then returns
 */
#include        <stdio.h>
#include        <signal.h>
// #define      SHHHH
main()
{
        void wakeup(int);
        printf("about to sleep for 4 seconds\n");
        signal(SIGALRM, wakeup);                /* catch it */
        alarm(4);                               /* set clock */
        pause();                                /* freeze here */
        printf("Morning so soon?\n");           /* back to work */
}
void wakeup(int signum)
{
#ifndef SHHHH
        printf("Alarm received from kernel\n");
#endif
}
```

Interval timers

Every process has three timers

- ITIMER_REAL decrements in real time, and delivers SIGALRM upon expiration.
- ITIMER_VIRTUAL decrements only when the process is executing, and delivers SIGVTALRM upon expiration.
- ITIMER_PROF decrements both when the process executes and when the system is executing on behalf of the process. Coupled with ITIMER_VIRTUAL, this timer is usually used to profile the time spent by the application in user and kernel space. SIGPROF is delivered upon expiration.

Timer values are defined by the following structures:

```
struct itimerval {
    struct timeval it_interval; /* next value, the repeating interval */
    struct timeval it_value;    /* current value, the initial interval */
};
struct timeval {
    long tv_sec;                /* seconds */
    long tv_usec;               /* microseconds */
};
```

Interval Timer Example-ticker_demo.c

```
#include        <stdio.h>
#include        <sys/time.h>
#include        <signal.h>
int main()
{
        void    countdown(int);
        signal(SIGALRM, countdown);
        if ( set_ticker(500) == -1 )
                perror("set_ticker");
        else while( 1 )
                pause();
        return 0;
}
void countdown(int signum)
{
        static int num = 10;
        printf("%d ..", num--);
        fflush(stdout);
        if ( num < 0 ){
                printf("DONE!\n");
                exit(0);
        }
}
```

```
/* [from set_ticker.c]
 * set_ticker( number_of_milliseconds )
 *      arranges for interval timer to issue SIGALRM's at regular intervals
 *      returns -1 on error, 0 for ok
 *      arg in milliseconds, converted into whole seconds and microseconds
 *      note: set_ticker(0) turns off ticker
 */

int set_ticker( int n_msecs )
{
        struct itimerval new_timeset;
        long    n_sec, n_usecs;

        n_sec = n_msecs / 1000 ;                /* int part     */
        n_usecs = ( n_msecs % 1000 ) * 1000L ;  /* remainder    */

        new_timeset.it_interval.tv_sec  = n_sec;        /* set reload       */
        new_timeset.it_interval.tv_usec = n_usecs;      /* new ticker value */
        new_timeset.it_value.tv_sec     = n_sec  ;      /* store this       */
        new_timeset.it_value.tv_usec    = n_usecs ;     /* and this         */

        return setitimer(ITIMER_REAL, &new_timeset, NULL);
}
```

System call summaries

```
NAME
       getitimer, setitimer - get or set value of an interval timer

SYNOPSIS
       #include <sys/time.h>

       int getitimer(int which, struct itimerval *value);
       int  setitimer(int which, const struct itimerval *value,
            struct itimerval *ovalue);

DESCRIPTION
       The system provides each  process  with  three  interval  timers,  each
       decrementing in a distinct time domain.  When any timer expires, a
       signal is sent to the process, and the timer (potentially) restarts.

RETURN VALUE
       On  success,  zero is returned.  On error, -1 is returned, and errno is
       set appropriately.
```

Signal handling

Some concepts of signals

- Every signal has a name. These names all begin with the three characters SIG
- No signal has a single number of 0.
- Signals are classic of asynchronous events. Signals occur at what appear to be random times to the process

Conditions in which a signal will be generated

- The terminal-generated signals occur when users press certain terminal keys.
- Hardware exceptions generate signals: divide by 0, invalid memory reference, etc.
- kill(2) function allows a process to send any signal to another process or process group

- kill (1) command allows us to send signals to other processes. This is an interface to kill(2) function.
- Software condition can generate signals when something happens about which the process should be notified, such as SIGURG, SIGPIPE, SIGALRM.

Signal handling

SIGNAL	ID	DEFAULT	DESCRIPTION
SIGHUP	1	Termin.	Hang up on controlling terminal
SIGINT	2	Termin.	Interrupt. Generated when we enter CNRTL-C
SIGQUIT	3	Core	Generated when at terminal we enter CNRTL-\
SIGILL	4	Core	Generated when we executed an illegal instruction
SIGTRAP	5	Core	Trace trap (not reset when caught)
SIGABRT	6	Core	Generated by the abort function
SIGFPE	8	Core	Floating Point error
SIGKILL	9	Termin.	Termination (can't catch, block, ignore)
SIGBUS	10	Core	Generated in case of hardware fault
SIGSEGV	11	Core	Generated in case of illegal address
SIGSYS	12	Core	Generated when we use a bad argument in a system service call
SIGPIPE	13	Termin.	Generated when writing to a pipe or a socket while no process is reading at other end
SIGALRM	14	Termin.	Generated by clock when alarm expires
SIGTERM	15	Termin.	Software termination signal
SIGURG	16	Ignore	Urgent condition on IO channel
SIGCHLD	20	Ignore	A child process has terminated or stopped
SIGTTIN	21	Stop	Generated when a background process reads from terminal
SIGTTOUT	22	Stop	Generated when a background process writes

```
                              to terminal
SIGXCPU     24     Discard    CPU time has expired
SIGUSR1     30     Termin.    User defiled signal 1
SIGUSR2     31     Termin.    User defined signal 2
```

Signal handling-unreliable signal handling: signal

A process calls signal to select one of the three responses to a signal

- default action (usually termination), for example signal(SIGALRM, SIG_DFL)
- ignore the signal, for example signal(SIGALRM, SIG_IGN)
- invoke a function, for example signal(SIGALRM, handler)

Problems with the *signal* function

- Mouse trap problem

  In the old daysm (system V), you have to reset the signal handler after each catch of signals

```
void handler(int s)
{
    /*process is vulnerable here*/
    signal(SIGINT, handler);    /* reset */
    ...                         /* do work here */
}
```

- You do not know why the signal was sent
- You can not safely bock other signals while in a handler.

Portability of *signal* function

The *signal* function is defined by ISO C, implementation of it varies from one Unix version to another

- The original Unix signal() would reset the handler to SIG_DFL, and System V (and the Linux kernel and libc4,5) does the same.
- On the other hand, BSD does not reset the handler, but blocks new instances of this signal from occurring during a call of the handler. The glibc2 library follows the BSD behavior. FreeBSD and Mac OS X follow the BSD strategy.
- Solaris follows the System V semantics.

Signal handling: sigaction

```
SYNOPSIS
       #include <signal.h>
       int sigaction(int signum, const struct sigaction *act, struct sigaction
       *oldact);

DESCRIPTION
       The  sigaction system call is used to change the action taken by a
       process on receipt of a specific signal.

       The sigaction structure is defined as something like
              struct sigaction {
                  /*use only one of these two*/
                  void (*sa_handler)(int);    /*SIG_DFL, SIF_IGN, or function*/
                  void (*sa_sigaction)(int, siginfo_t *, void *); /* New handler*/
                  sigset_t sa_mask;           /*signals to block while handling*/
                  int sa_flags;               /*enable various behaviors*/
              }
RETURN VALUE
       sigaction returns 0  on  success  and -1 on error.
```

For more information about this function, check the man page

```
man sigaction
```

Signal handling: sa_flag in sigaction struct

sa_flags specifies a set of flags which modify the behavior of the signal handling process. It is formed by the bitwise OR of zero or more of the following (partial):

```
SA_RESETHAND
   Restore  the  signal action to the default state once the
   signal handler has been called.

SA_RESTART
   If a signal is caught while a process is executing a system call,
   the kernel will restart the system call after the signal handler
   returns. If this flag is not set in the sa_flag, after the signal
   handler returns, the system call will be aborted with a return
   value of -1 and will set errno to EINTR (aborted due to a signal
   interruption).

SA_NODEFER
     Do not prevent the signal from being received from within
   its own signal handler.

SA_SIGINFO
    The signal handler takes 3 arguments, not one.   In  this
   case,  sa_sigaction  should be set instead of sa_handler.
```

```
SA_NOCLDSTOP
    when the signal_num is SIGCHLD, the kernel will generate the
    SIGCHLD signal to the calling process whenever its child process
    is either terminated, but not when the child process has been stopped.
```

Example: Using sigaction

Run program sigactdemo.c

Blocking signals: sigprocmask and sigsetops

In certain situation, it is a good idea to block some signals. You can block signals at the signal-handler level and at the process level.

- To block signals in a signal handler, set the sa_mask member of the struct sigaction you pass to sigaction when you install the handler
- To block signals at all times for a process, use sigprocmask.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

Arguments of sigprocmask

- new_mask: defines a set of signals to be set or reset in a calling process signal mask. new_mask = NULL, current process signal mask unaltered.
- cmd: specifies how the new_mask value is to be used:
  - SIG_SETMASK: Overrides the calling process signal mask with the value specified in the new_mask argument.
  - SIG_BLOCK: Adds the signals specified in the new_mask argument to the calling process signal mask.
  - SIG_UNBLOCK: Removes the signals specified in the new_mask argument from the calling process signal mask
- old_mask: Address of a sigset_t variable that will be assigned the calling processings original signal mask. old_mask = NULL, no previous signal mask will be return.

Example of sigprocmask

```
/* The example checks whether the SIGINT signal is present in a process
signal mask and adds it to the mask if it is not there. It clears the
SIGSEGV signal from the process signal mask. */

int main( ){
       sigset_t sigmask;
       sigemptyset(&sigmask);              /*initialize set */
       if(sigprocmask(0,0,&sigmask)==-1)  /*get current signal mask*/
       {
           perro(sigprocmask);
           exit(1);
       }
       else
           sigaddset(&sigmask, SIGINT);   /* set SIGINT flag*/
       sigdelset(&sigmask, SIGSEGV);      /* clear SIGSEGV flag */
       if (sigprocmask(SIG_SETMASK,&sigmask,0) == -1)
           perro(sigprocmask);           /* set a new signal mask */
}
```

Reentrant Code/functions

Reentrant function: Any function, including signal handler, that can be called when it is already active and not cause any problems.

If the handlers are not reentrant functions, then they can not be interrupted by other signals, which means other signals must be blocked.

How can I know if a function is reentrant or not?

A function can be non-reentrant due to the following reasons

- It uses/modify static/global data structures in an non-atomic way.
- It call malloc or free
- It use part of the standard I/O library. Most implementation of standard I/O library use global data structures in a nonreentrant way

An example of nonentrant function

```
 char *strtoupper(char *string) {
  static char buffer[MAX_STRING_SIZE];
  int index;
  for (index = 0; string[index]; index++)
    buffer[index] = toupper(string[index]); buffer[index] = 0;
  return buffer;
}
(from AIX manual)
```

An example of a re-entrant function

```
char *strtoupper_r(char *in_str, char *out_str) {
  int index;
  for (index = 0; in_str[index]; index++)
    out_str[index] = toupper(in_str[index]);
  out_str[index] = 0;
  return out_str;
}
```

kill: sending signals from a process

- int signal_num: the integer value of a signal to be sent to one or more processes designated by pid.
- pid_t pid value
  - positive value: pid is process ID. Sends signal_num to that process.
  - 0: sends signal_num to all process whose group ID is the same as the calling process.
  - -1: Sends signal_num to all processes whose real user ID is the same as the effective user ID of the calling process.

The video sample games

This program demonstates the application of a timer in move a string on the screen.

Open and run bounce1d.c and bouce2d.c

The signal on input: Asynchronous I/O

There are two method for asynchronous I/O. One method is using O_ASYNC on the file descriptor flags. Another way is using aio_read.

O_ASYNC method

- install the handler by calling either signal or sigaction

  ```
  signal(SIGIO, on_input)
  ```

- Set the process ID or process group ID to receive the signal for the descriptor.

  ```
  fcntl(0, F_SETOWN, getid());
  ```

- Enable asynchronous I/O on the descriptor by calling fcntl with a command of F_SETFL to set the O_ASYNC file status flag. Note: This step can only be performed on descriptors that refer to terminals or networks.

  ```
  fd_flags = fcntl(0, F_GETFL);
  ```

```
fcntl(0, F_SEFL, (fd_flags|O_ASYNC));
```

Sample code bounce_async.c

The signal on input: Asynchronous I/O

aio_read method

- install the handler

```
signal(SIGIO, on_input)
```

- Set value in a struct aiobuf variable to describe what input to wait for and what signal to send when the input is read.

```
static char input[1];                  /* 1 char of input */

/* describe what to read */
kbcbuf.aio_fildes     = 0;             /* standard intput */
kbcbuf.aio_buf        = input;         /* buffer          */
kbcbuf.aio_nbytes     = 1;              /* number to read  */
kbcbuf.aio_offset     = 0;              /* offset in file  */

/* describe what to do when read is ready */
kbcbuf.aio_sigevent.sigev_notify = SIGEV_SIGNAL; /*another option is SIGEV_NONE*/
kbcbuf.aio_sigevent.sigev_signo  = SIGIO;  /* send sIGIO   */
```

- Place a read request for passing the variable to aio_read

```
aio_read(&kbcbuf);                  /* place a read request     */
```

Sample code bounce_aio.c