System Programming
(ELEC462)

Event-Driven Programming

Dukyun Nam

HPC Lab@KNU

Contents

Introduction

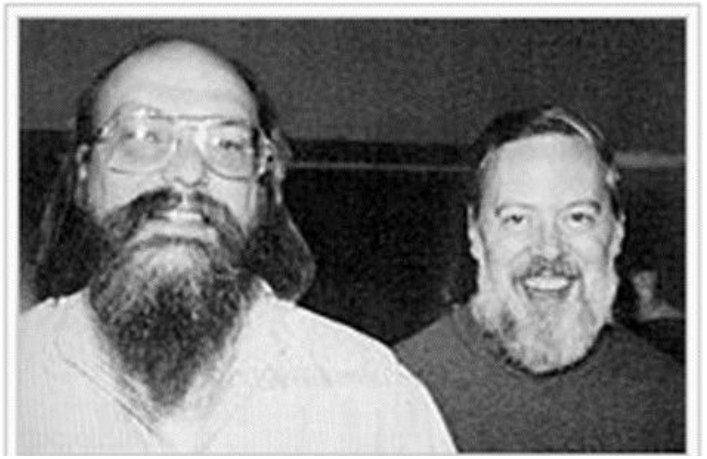
- Writing programs driven by asynchronous (unblocking) events
- The `curses` library: purpose and use
- Alarms and interval timers
  - `alarm(), setitimer(), getitimer()`
- Reliable signal handling, critical sections, and inter-process communication
  - `(u)sleep(), pause(), sigaction(), sigprocmask(), kill()`

Introduction (cont.)

- Featuring
  - screen control
  - tty control
  - time
  - doing two things at once
- General remarks
  - Writing space travel led to Unix
  - Video games are fun to write and use
  - Character-based games are sort of like graphics games, just with fatter pixels
  - Video games lead to important ideas about how to do several things at once. Kind of like multi-tasking

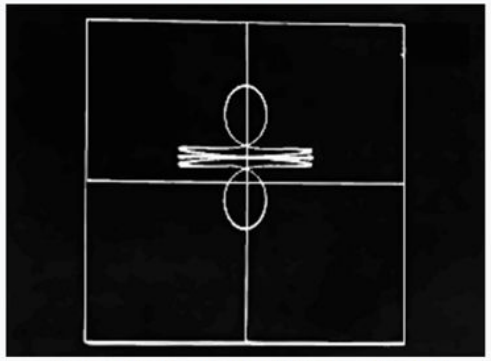
Video Games

- Space Travel
  - An early video game developed by Ken Thompson in 1969 that simulates travel in the solar system



< Ken Thompson (left) &
Dennis Ritchie (right) >

Space Travel



Gameplay image of *Space Travel*

Developer(s)	Ken Thompson
Designer(s)	Ken Thompson
Platform(s)	Multics, GECOS, PDP-7
Release	1969
Genre(s)	Simulation game
Mode(s)	Single-player

A modified PDP-7 under restoration in Oslo, Norway

Manufacturer	Digital Equipment Corporation
Product family	Programmed Data Processor
Type	Minicomputer
Release date	1965; 55 years ago
Introductory price	US$72,000 (equivalent to $584,135 in 2019)

Video Games (cont.)

- Space Travel (cont.)
  - Creates images of planets, asteroids (minor planets), spaceships, and others, and keeps those images moving
  - Each object
    - Has properties: speed, position, direction, momentum, and other attributes
    - Interacts: e.g., An asteroid may face a spaceship or another asteroid.
  - The player
    - Flies their ship around a 2D-scale model of the solar system,
    - Attempts to land on various planets and moons, with no specific purpose,
    - Can move and turn the ship, and
    - Adjusts the overall speed by adjusting the scale of the simulation.
  - The spaceship
    - Affected by the single strongest “gravitational pull” of the astronomical bodies

How a Video Game Works

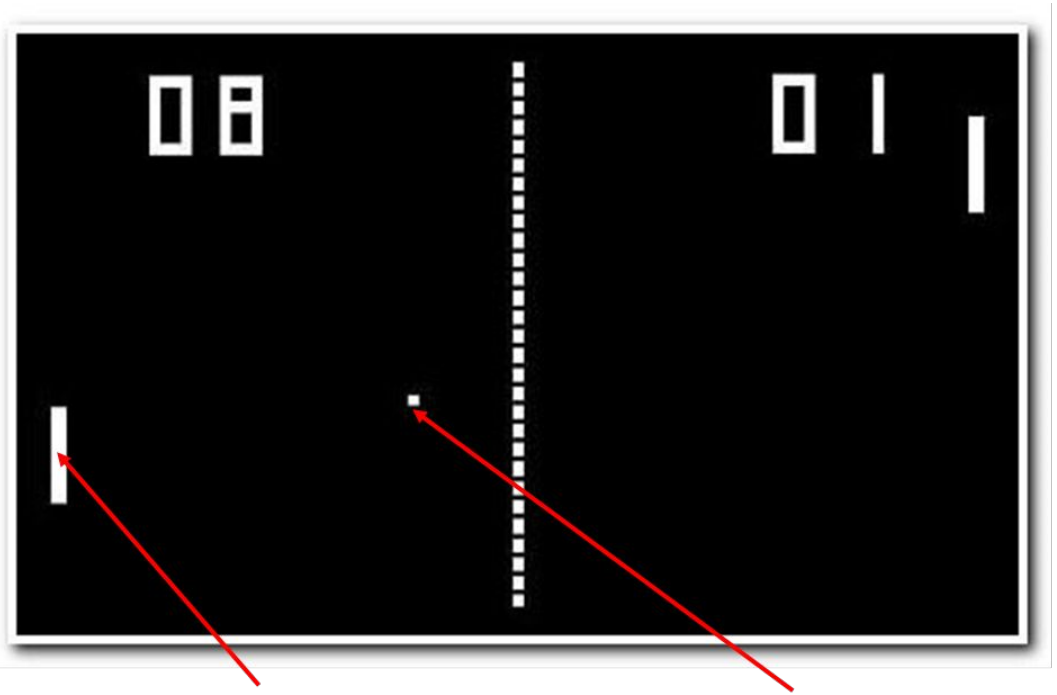
- A video game combines several basic ideas and principles
- Four major aspects:
  - 1) Space
    - The game has to draw images at specific locations on the screen
  - 2) Time
    - Images move across the screen at different “speeds”
    - Changes in position occur at certain “intervals”
  - 3) Interruptions
    - The program moves objects smoothly across the screen
    - Users can send input (causing interrupts) whenever they like
  - 4) Doing several things
    - The game has to keep “several objects moving” and also respond to “interruptions” at the same time (so called, support of multitasking)

Operating Systems Address Similar Questions

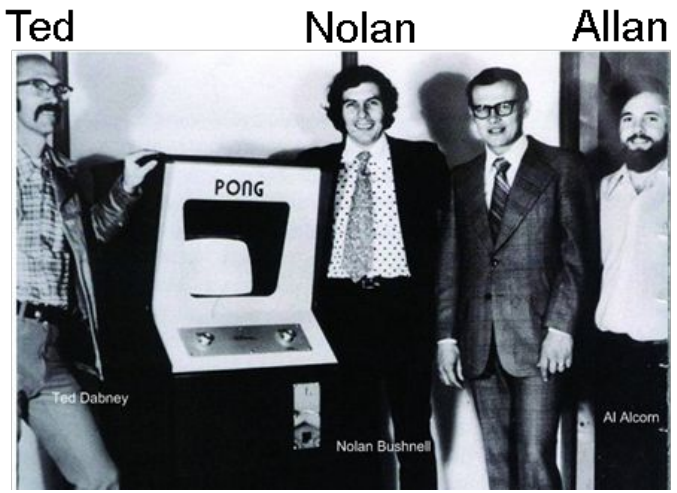
- Kernel
  - Loads programs into memory space and keeps track of the location of each program
  - Schedules programs to run for “short intervals” and also internal tasks to be done at “specific times”
  - Has to respond to quickly when users and other external devices send input at “unpredictable” times
- Question: How does the kernel keep data from becoming disordered and confused?
  - Doing several things at once may be very CHALLENGING

“PONG”

- One of the earliest arcade video game (in 1972)
  - Originally manufactured by Atari (Co-founder: Ted & Nolan)



Paddle: user control *Ball: bounces around the screen*



The Project: Write Single-Player `pong`

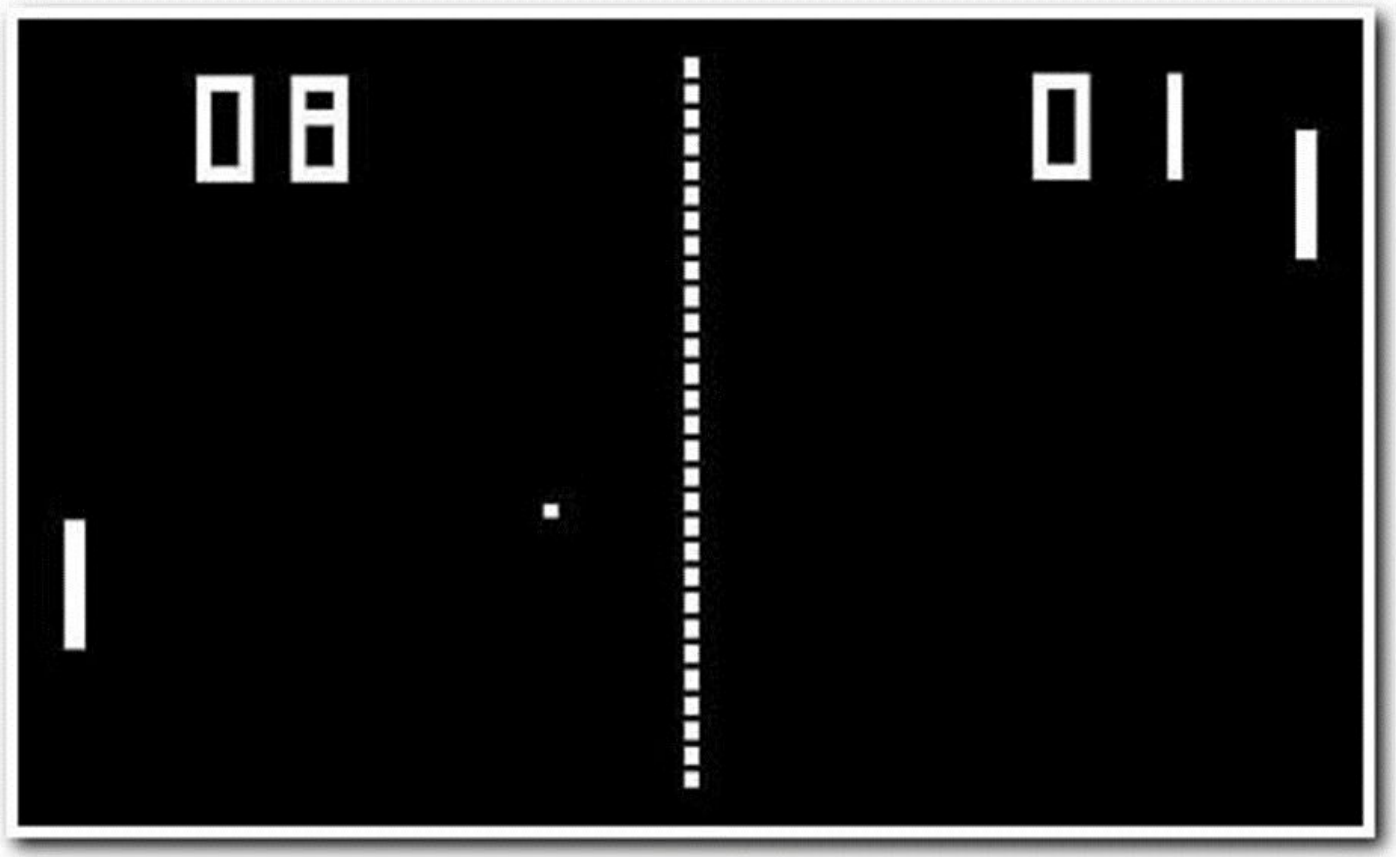
- General outline
  - Ball keeps moving at some speed
  - Ball bounces off walls and paddle
  - User presses keys to move paddle up and down
- Skills we need to learn
  - a) Draw stuff on screen at specific positions
  - b) Draw stuff at particular times
  - c) Use a) and b) to do animation
  - d) How to get user input without stopping action

The Project: Write Single-Player `pong` (cont.)

- Towards writing a video game
  - 1. **Space Programming**: Drawing images
  - 2. **Time Handling**: Making animated effects
  - 3. **Signal Handling**: Moving the bar when keyboard input
  - 4. **Inter-Process Communication**: Sending a signal to another process from a process

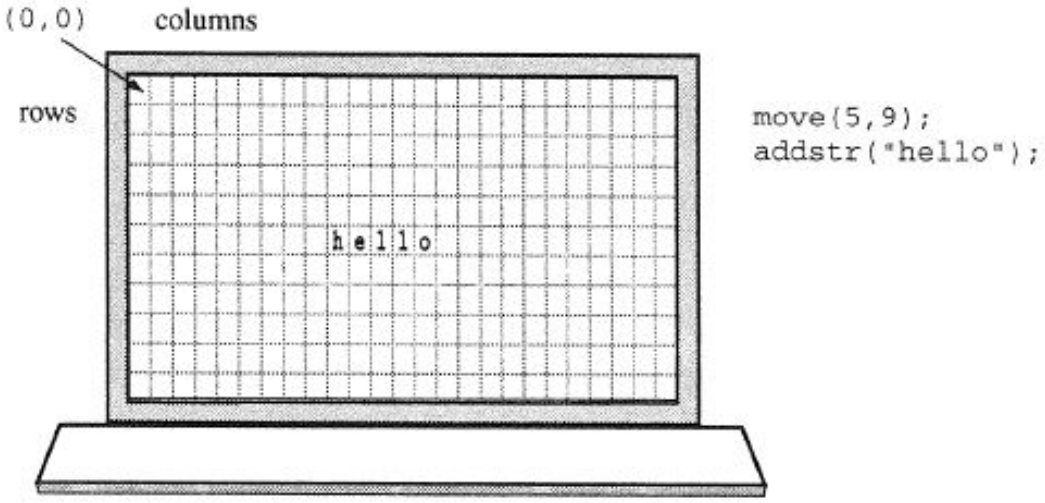
Space Programming: The `curses` Library

- How to draw images at specific location on the screen?



Space Programming: The `curses` Library (cont.)

- "Terminal control" library
  - Consists of a set of functions
    - Allowing a programmer to (i) set the position of the cursor and (ii) control the appearance of text on a terminal screen
  - Used by most programs controlling the terminal screen
- The terminal screen?
  - A grid of character cells
  - The origin – upper left corner of the screen



< The three standard file descriptors >

Space Programming: The `curses` Library (cont.)

- Includes functions to
  - Move the cursor to any cell on the screen: `move()`
  - Add chars to and erase chars from the screen: `addstr()`
  - Set visual attributes of chars (e.g., color, brightness, …): `standout()`
  - Create and control windows and other regions of text: `initscr(), endwin(),`…

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 (*r*,*c*)
`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

Space Programming: The `curses` Library (cont.)

- Where to find the curses library?
  - What if it is not installed? Try the following command:
  - `$ sudo apt install libncurses5-dev libncursesw5-dev`
  - `$ less /usr/include/curses.h`

Curses Example 1: `hello1.c`

- Compile with `curses` library
  - `$ gcc -o hello1 hello1.c -lcurses`
  - `-lcurses`: Link `curses` library

```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>

int 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      */

        return 0;
}
```

```
Hello, world
```

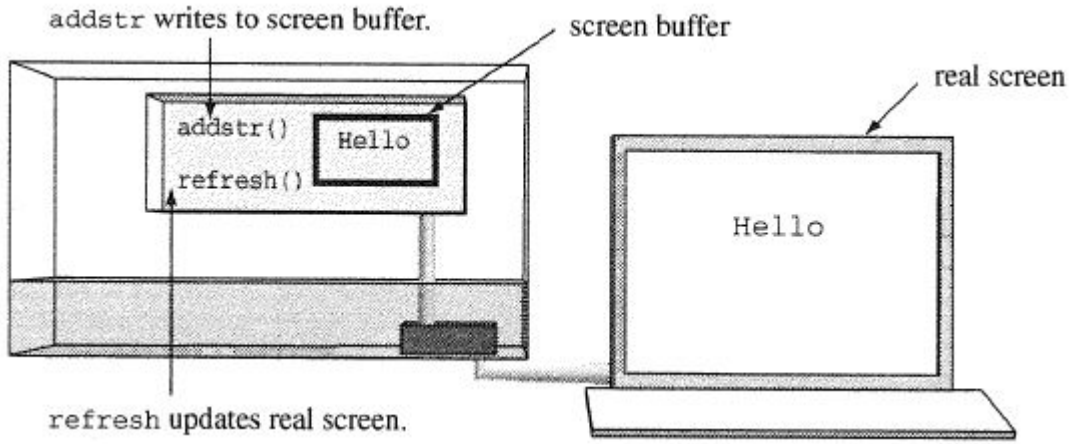
Curses Example 2: `hello2.c`

```c
/* hello2.c
 *      purpose  show how to use curses functions with a loop
 *      outline  initialize, draw stuff, wrap up
 */

#include        <unistd.h>
#include        <curses.h>

int 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    */
        sleep(3);                       /* wait for 3 seconds   */
        endwin();                       /* reset the tty etc    */

        return 0;
}
```

```
Hello, world
  Hello, world
    Hello, world
      Hello, world
        Hello, world
          Hello, world
            Hello, world
              Hello, world
                Hello, world
                  Hello, world
                    Hello, world
                      Hello, world
                        Hello, world
                          Hello, world
                            Hello, world
                              Hello, world
                                Hello, world
                                  Hello, world
                                    Hello, world
                                      Hello, world
                                        Hello, world
                                          Hello, world
                                            Hello, world
                                              Hello, world
                                                Hello, world
                                                  Hello, world
                                                    Hello, world
                                                      Hello, world
                                                        Hello, world
                                                          Hello, world
                                                            Hello, world
                                                              Hello, world
```

Curses Internals: Virtual and Real Screens

- What does the `refresh` function do?
  - Let's do an experiment
    - Comment out that line, recompile and run the program
    - Then what can you see on the (real) screen?

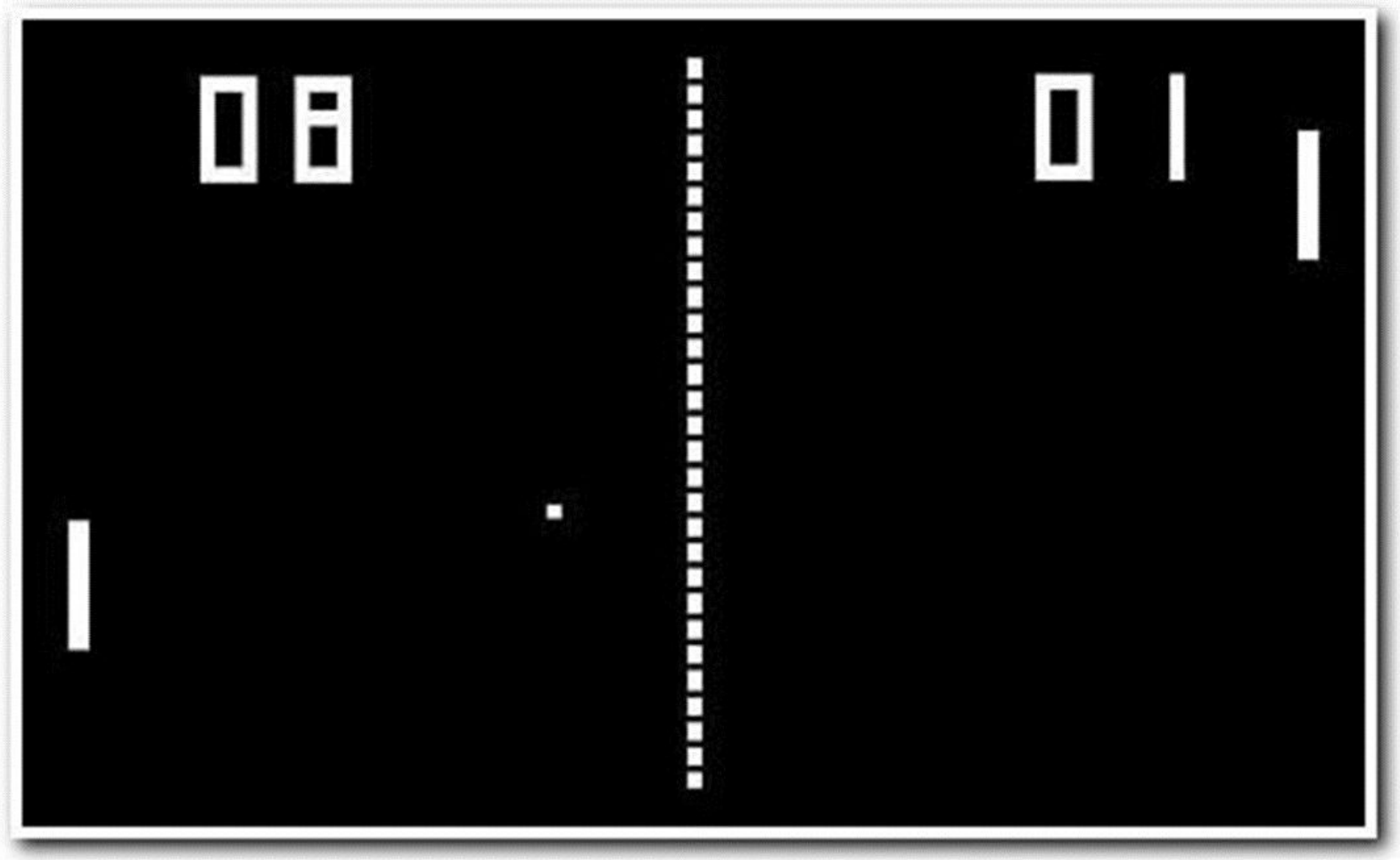


< Curses keeps a copy of the real screen >

- `curses` is designed to update your text screen
  - Minimizes data flow by working with virtual screens
- The `refresh` function
  - Compares the workspace screen to the copy of the real screen
  - Sends out through the terminal driver characters and screen control codes needed to make the real screen match the working screen, like "disk buffering"!

Time Handling

- How to move or to show “animated effects” on the images?



Time Programming: `sleep`

- To write a video game, we have to put images at specific places at specific times
  - Add time to our programs
  - For this, we use the system `sleep` function

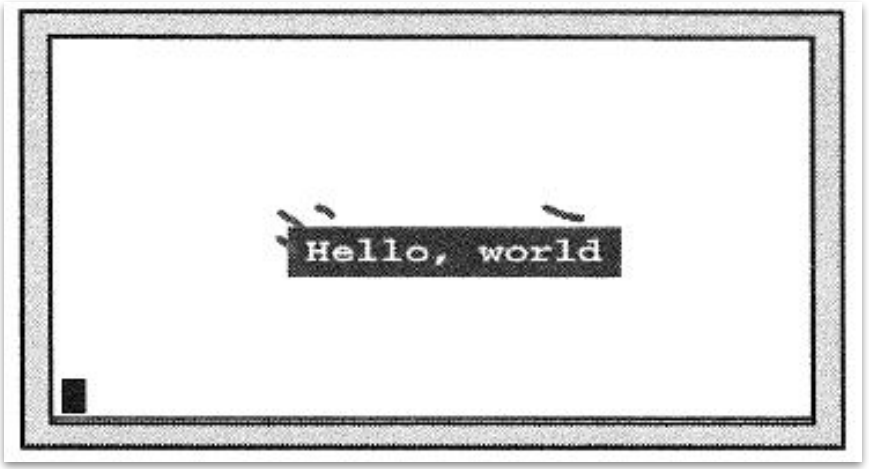
Animation Example 1: `hello3.c`

```c
/* hello3.c
 *      purpose  using refresh and sleep for animated effects
 *      outline  initialize, draw stuff, wrap up
 */
#include        <stdio.h>
#include        <unistd.h>
#include        <curses.h>

int 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();

        return 0;
}
```

```
Hello, world
  Hello, world
    Hello, world
      Hello, world
        Hello, world
          Hello, world
```

Animation Example 2: `hello4.c`

```c
/* hello4.c
 *      purpose show how to use erase, time, and draw for animation
 */
#include        <stdio.h>
#include        <unistd.h>
#include        <curses.h>

int 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();

        return 0;
}
```



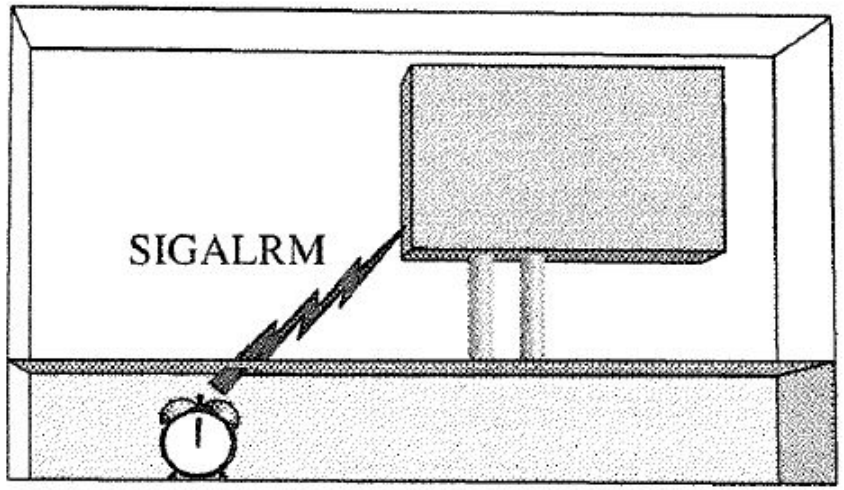
< A message drifts slowly down the screen >

Animation Example 3: `hello5.c`

```c
/* hello5.c
 *    purpose  bounce a message back and forth across the screen
 *    compile  cc hello5.c -lcurses -o hello5
 */
#include        <unistd.h>
#include        <curses.h>

#define LEFTEDGE        10
#define RIGHTEDGE       30
#define ROW             10

int main()
{
        char    message[] = "Hello";
        char    blank[]   = "     ";
        int     dir = +1;
        int     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);                  /* erase string         */
                addstr( blank );
                pos += dir;                     /* advance position     */
                if ( pos >= RIGHTEDGE )         /* check for bounce     */
                        dir = -1;
                if ( pos <= LEFTEDGE )
                        dir = +1;
        }
        return 0;
}
```

Programming with Time I: Alarms

- Adding a delay: `sleep`
  - `sleep(n)`: suspends the current process for *n* seconds or until an unignored signal (like `SIGINT`) arrives
- How `sleep()` works: using alarms in Unix (Linux)
  - Set an alarm for # secs you wish for a program to sleep
  - Pause until the alarm goes off



Every process has its own timer.

How the sleep function works:

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

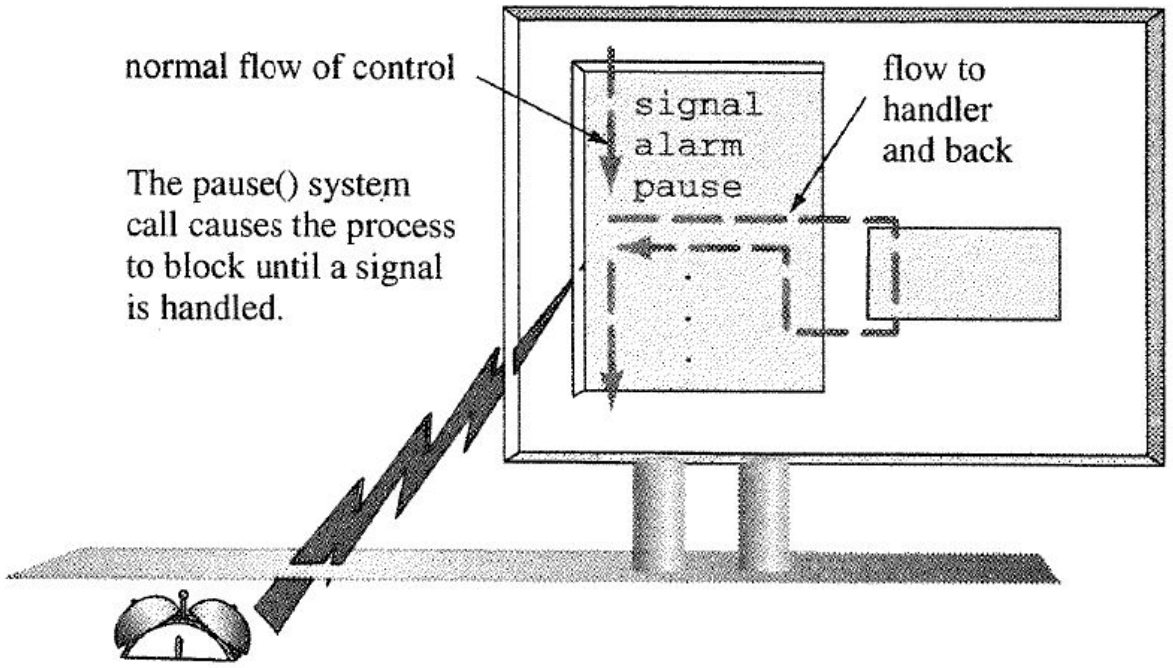
< A process sets an alarm then suspends execution >

Alarm Example 1: `sleep1.c`

```c
/* sleep1.c
 *      purpose show how sleep works
 *      usage   sleep1
 *      outline sets handler, sets alarm, pauses, then returns
 */
#include        <stdio.h>
#include        <unistd.h>
#include        <signal.h>
// #define      SHHHH

int 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 */

        return 0;
}

void wakeup(int signum)
{
#ifndef SHHHH
        printf("Alarm received from kernel\n");
#endif
}
```

Programming with Time I: Alarms (cont.)

- How `sleep()` works: using alarms in Unix (Linux) (Cont.)
  - After 4 secs pass on the alarm timer, the kernel sends `SIGALRM` to the process, causing control to jump from the pause to the signal handler



< Flow of execution into handler >

Programming with Time I: Alarms (cont.)

alarm	
PURPOSE	Set an alarm timer for delivery of a signal
INCLUDE	#include <unistd.h>
USAGE	unsigned old = alarm(unsigned seconds)
ARGS	seconds how long to wait
RETURNS	-1 if error old time left on timer

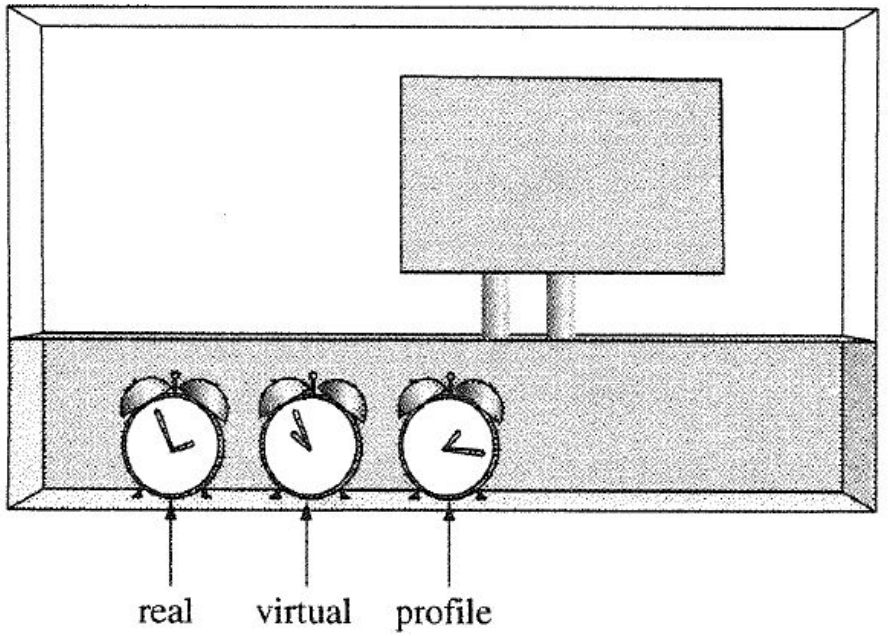
pause	
PURPOSE	Wait for signal
INCLUDE	#include <unistd.h>
USAGE	result = pause()
ARGS	no args
RETURNS	-1 always

Programming with Time II: Interval Timers

- Why do we want to need an interval timer?
  - e.g., Taxi meter device
    - The basic fare is 3,300 won for 3 mins. (initial)
    - It increases 100 won every 30 secs. (repeat)
    - There is a NEED to set a timer at “regular intervals.”
  - Or what other cases??
- What if we want to set a finer-grained delay?
  - e.g., The ball is getting faster every 10.5 seconds
  - For a finer delay: can use `usleep()`
  - `usleep(n)`
    - Suspends the current process n microseconds (1/1,000,000 sec)

Programming with Time II: Interval Timers (cont.)

- Every process has three timers
- Each timer has two settings:
  - (1) The time until the first alarm
  - (2) The interval between repeating alarms



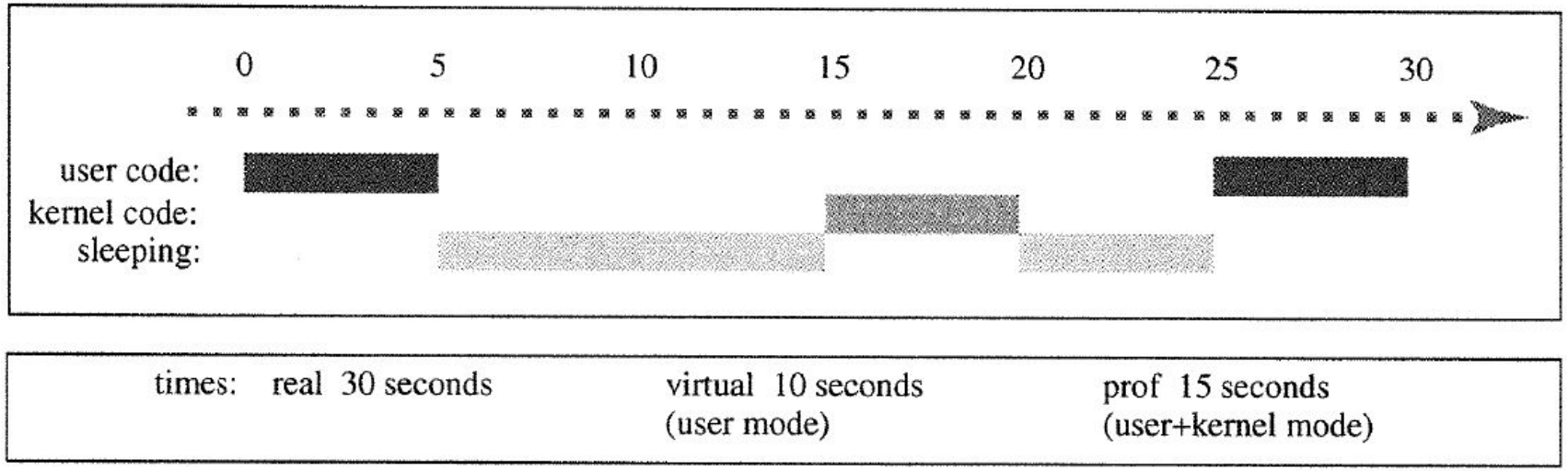
Every process has three timers.

Each timer has two settings: the time until the first alarm and the interval between repeating alarms.

< Every process has three timers >

Programming with Time II: Interval Timers (cont.)

- Three kinds of timers: real, process (virtual), and profile
  - e.g., Consider a program, terminating in 30 seconds after running
    - Total (real): 30 seconds
    - User mode: 10 seconds
    - Profile (user + kernel (or system)) mode: 15 seconds (i.e., program time)



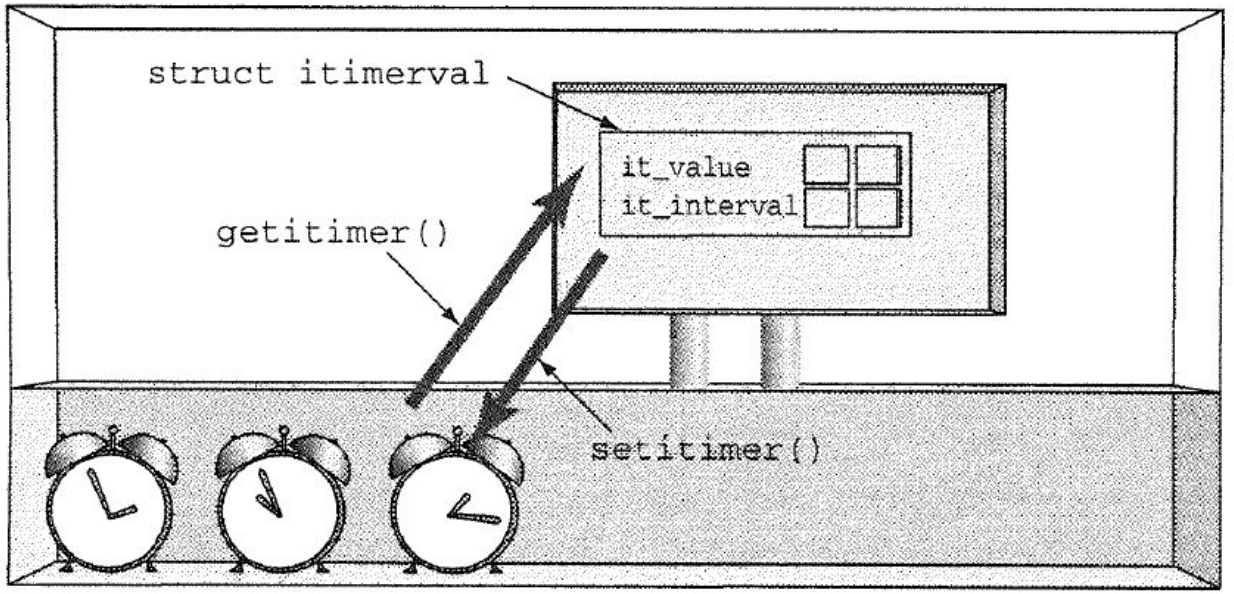
< Where does the time go? >

Programming with Time II: Interval Timers (cont.)

- The kernel provides timers to measure each of these types
  - `ITIMER_REAL`
    - Ticks in “real time” (wall clock time)
    - Send `SIGALRM` if this (real) timer expires
  - `ITIMER_VIRTUAL`
    - Ticks “only when the process runs in user mode” (like an NBA game)
    - Send `SIGVTALRM` if this (virtual) timer expires
  - `ITIMER_PROF`
    - Ticks when the process runs in user mode and when the kernel is running system calls made by this process
    - Send `SIGPROF` if this (profile) timer expires

Programming with Time II: Interval Timers (cont.)

- Programming with the interval timers
  - 1. Decide on an initial interval and a repeating interval
  - 2. Set values in a struct `itimerval`: Initial interval (ex. `it_value` <- 1 hr) and repeating interval (ex. `it_interval` <- 4 hrs)
  - 3. Pass the `itimerval` to the timer by calling `setitimer`



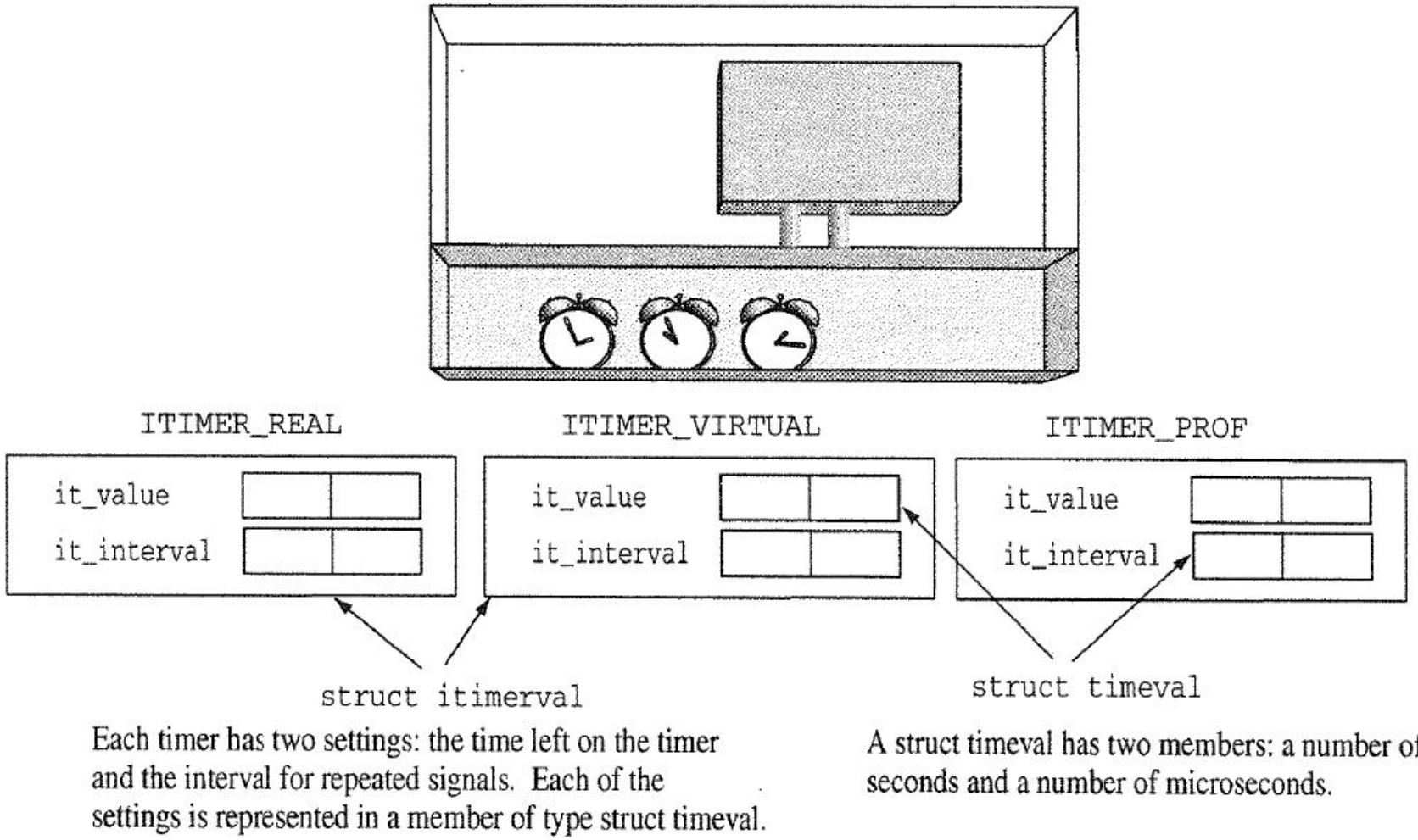
< Reading and writing timer settings >

Programming with Time II: Interval Timers (cont.)

- More details about the data structures

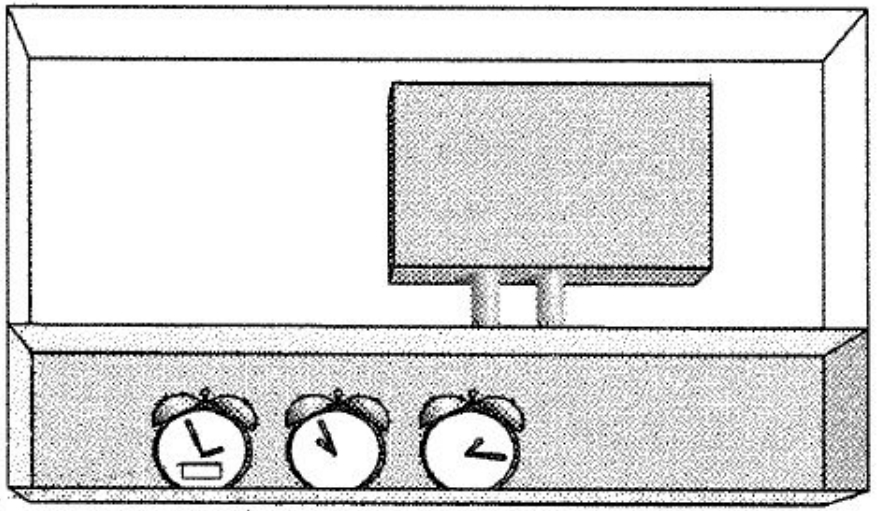
```
struct itimerval{
     /* time to next timer expiration */
     struct timeval it_value;
     /* reload it_value with this  */
     struct timeval it_interval;
};


struct timeval{
     time_t tv_sec;           /* seconds */
     suseconds_t       tv_usec; /* and microseconds
*/
};
```

Programming with Time II: Interval Timers (cont.)



< Inside the interval timers >

Programming with Time II: Interval Timers (cont.)



ITIMER_REAL

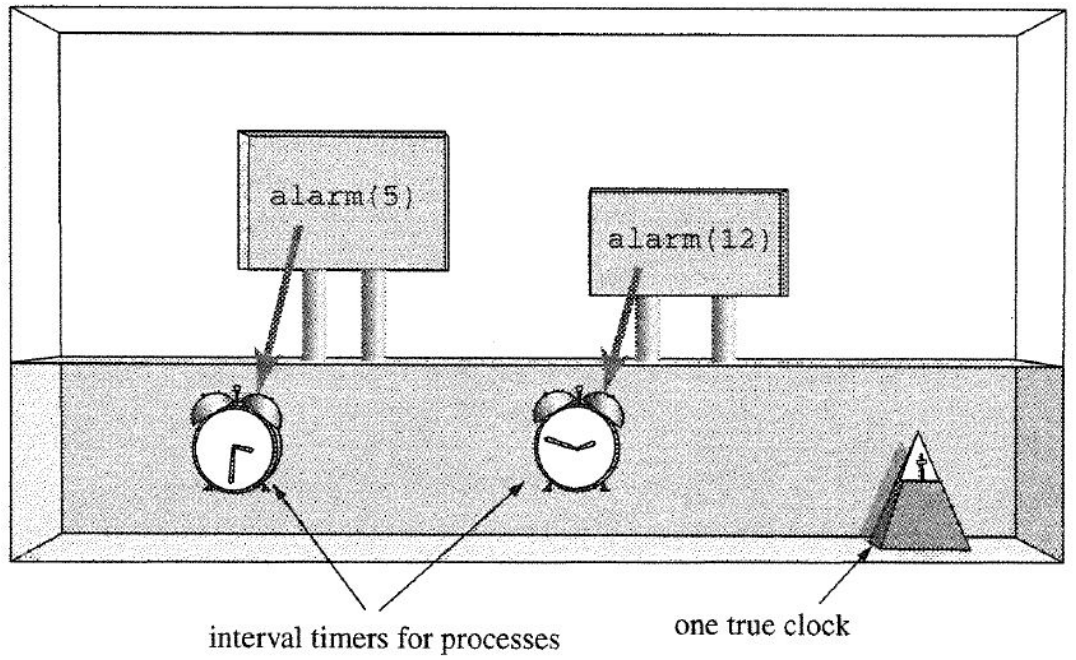
	tv_sec	tv_usec
it_value	60	500000
it_interval	240	250000

This example sets the real time interval timer to send a signal in 60.5 seconds and then to send signals every 240.25 seconds.

< Seconds and microseconds >

Programming with Time II: Interval Timers (cont.)

- How many clocks does the computer have?
  - Every process on the system have three separate clock?!
  - Actually, the system needs only one clock to set the tempo (for ticking)
  - Each process sets its private timer by calling alarm
  - The kernel updates all process timers at each signal from its clock
    - When the counters reach zero, then SIGALRM is fired



Each process sets its private timer by calling `alarm`. The kernel updates all process timers at each signal from its clock.

< Two timers, one clock >

System Call Summaries

getitimer,setitimer	
PURPOSE	Get or set value of interval timer
INCLUDE	`#include <sys/time.h>`

USAGE	`result = getitimer( int which,`
	`struct itimerval *val);`
	`result = setitimer( int which,`
	`const struct itimerval *newval,`
	`struct itimerval *oldval);`

ARGS	`which` timer being read or set `val` pointer to current settings `newval` pointer to settings to be installed `oldval` pointer to settings being replaced
RETURNS	`-1` on error `0` on success

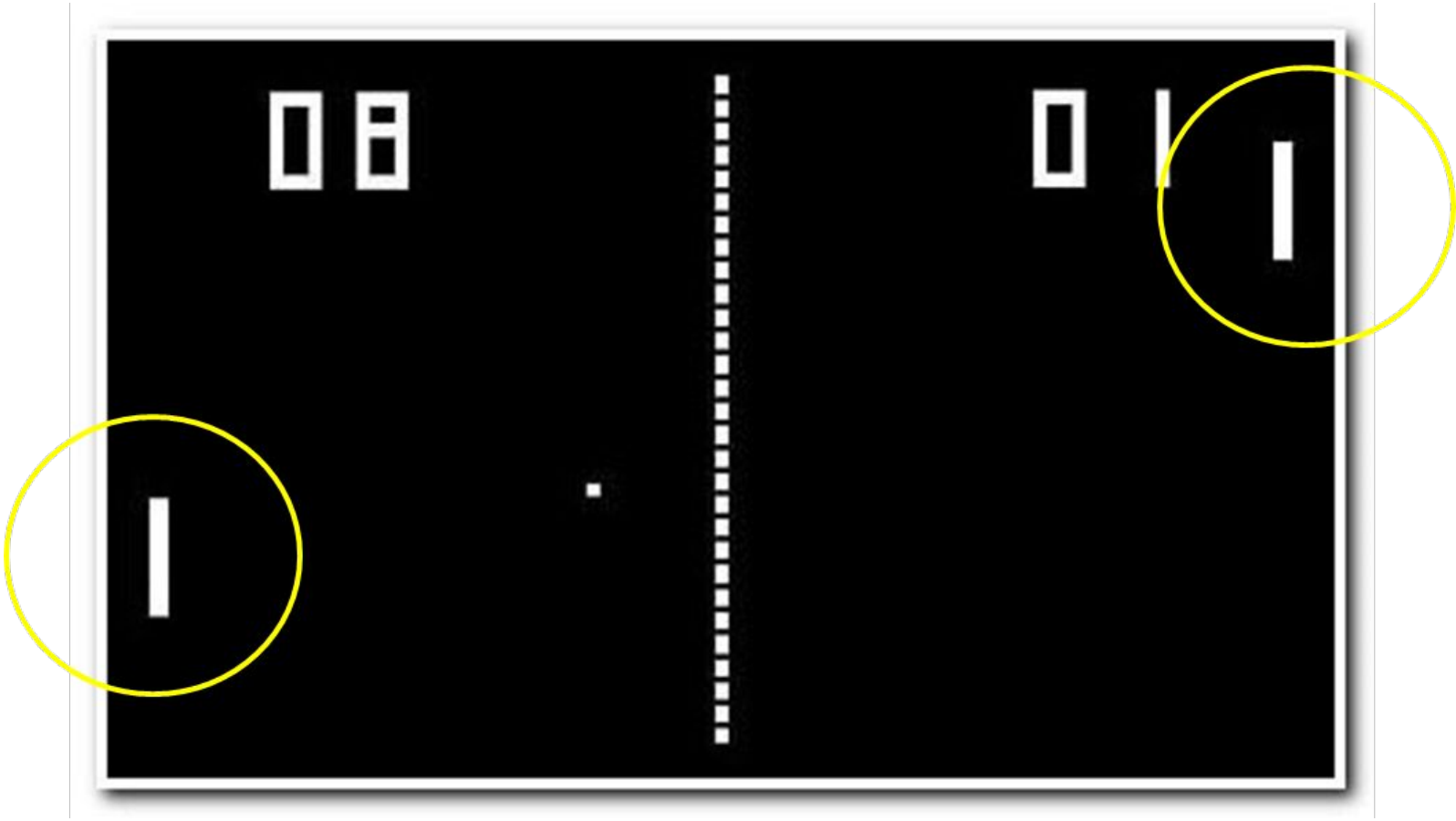
Interval Time Example 1: `ticker_demo.c`

```c
/* ticker_demo.c
 *    demonstrates use of interval timer to generate regular
 *    signals, which are in turn caught and used to count down
 */

#include        <stdio.h>
#include        <sys/time.h>
#include        <unistd.h>
#include        <stdlib.h>
#include        <signal.h>

int set_ticker(int);
void countdown(int);

int main()
{
        void    countdown(int);

        signal(SIGALRM, countdown);
        if ( set_ticker(500) == -1 )
                perror("set_ticker");
        else
                while( 1 )
                        pause();
        return 0;
}
```

```c
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);
}
```

Signal Handling

- How to move the bar when users type the keyboard?
  - Such interruptions (by keyboard) should be properly handled by the game

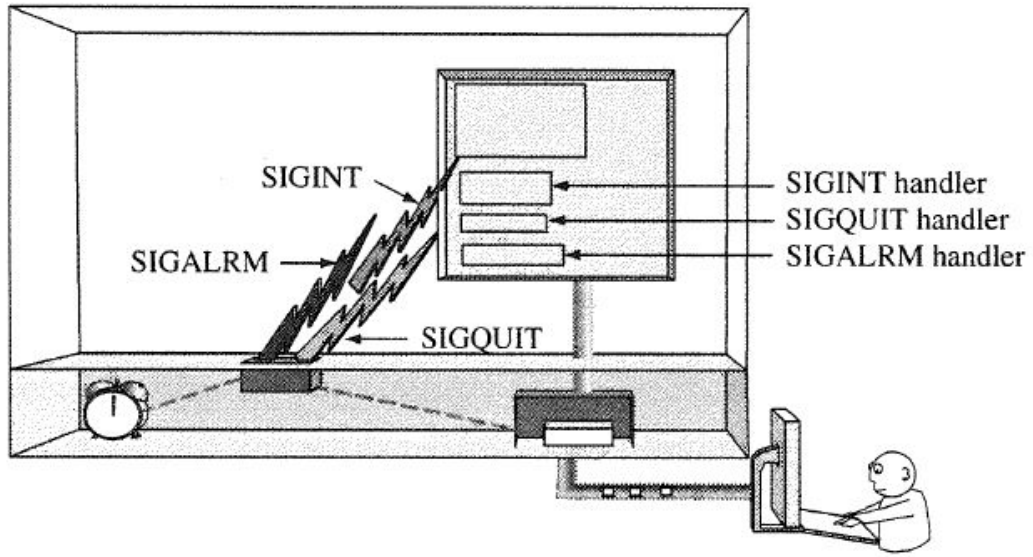


Signal Handling I: Using `signal`

- The kernel sends signals to a process to respond to a variety of events:
  - (i) Certain user keystrokes, (ii) illegal process behaviors, and (iii) elapsed timers, etc.
- Old-style signal handling (covered last class)
  - Default action (usually termination)
    - `signal(SIGALRM, SIG_DFL)`
  - Ignore the signal
    - `signal(SIGALRM, SIG_IGN)`
  - Invoke a (user-defined) function
    - `signal(SIGALRM, handler)`

Signal Handling I: Using `signal` (cont.)

- Handling multiple signals
  - If only one single arrives, then the original signal model works OK!
  - But what if a process receives multiple signals??
    - "Termination or ignore" (obvious) vs. "invoke a function" (not clear)
  - Q1. Is this handler disabled after each use?
  - Q2. What happens if a SIGY arrives while the process is in the SIGX handler?
  - Q3. What happens if a second SIGX arrives while the process is still in the SIGX handler? Or a third SIGX?
  - Q4. What happens if a signal arrives while the process is blocking on input in getchar() or read()?



< A process receiving multiple signals >

Multiple Signals Example 1: `sigdemo3.c`

```c
/* sigdemo3.c
 *      purpose:   show answers to signal questions
 *      question1: does the handler stay in effect after a signal arrives?
 *      question2: what if a signalX arrives while handling signalX?
 *      question3: what if a signalX arrives while handling signalY?
 *      question4: what happens to read() when a signal arrives?
 */

#include        <stdio.h>
#include        <unistd.h>
#include        <string.h>
#include        <signal.h>

#define INPUTLEN        100

void inthandler(int);
void quithandler(int);

int main(int ac, char *av[])
{
        void    inthandler(int);
        void    quithandler(int);
        char    input[INPUTLEN];
        int     nchars;

        signal( SIGINT,  inthandler );          /* set handler */
        signal( SIGQUIT, quithandler );         /* set handler */

        do {
                printf("\nType a message\n");
                nchars = read(0, input, (INPUTLEN-1));
                if ( nchars == -1 )
                        perror("read returned an error");
                else {
                        input[nchars] = '\0';
                        printf("You typed: %s", input);
                }
        }
        while( strncmp( input , "quit" , 4 ) != 0 );

        return 0;
}
```

```c
void inthandler(int s)
{
        printf(" Received signal %d .. waiting\n", s );
        sleep(2);
        printf("  Leaving inthandler \n");
}

void quithandler(int s)
{
        printf(" Received signal %d .. waiting\n", s );
        sleep(3);
        printf("  Leaving quithandler \n");
}
```

Multiple Signals Example 1: `sigdemo3.c` (cont.)

- Experiment for four questions:
  - Q1) Is the handler disabled after each use?

    (`^C` → `(handler)` → `^C`)

    - Each time handler must be reset after invoke.
    - Don't kill the process. Why?

```
dynam@DESKTOP-Q4IJBP7:~/lab8$ ./sigdemo3

Type a message
^C Received signal 2 .. waiting
  Leaving inthandler
^C Received signal 2 .. waiting
  Leaving inthandler
```

  - Q2) What happened if a SIG_INT arrived while the process is in the SIG_QUIT handler? (`^\` → `^C` → `^\`)

    - Back to the normal after jumping to the `quithandler`, `inthandler`, and back to `quithandler`

```
dynam@DESKTOP-Q4IJBP7:~/lab8$ ./sigdemo3

Type a message
^\ Received signal 3 .. waiting
  Leaving quithandler
^C Received signal 2 .. waiting
  Leaving inthandler
^\ Received signal 3 .. waiting
  Leaving quithandler
```

Multiple Signals Example 1: `sigdemo3.c` (cont.)

- Q3) What happened if a second SIG_INT arrived while the process is still in the SIG_INT handler? Or a third SIG_INT? (`^C → ^C → ^C`)
  - Recursively, calling the same handler
  - Ideally, better to block later signals…
- Q4) What happens if a signal arrives while the program is blocking on input? (`^\ → ^C → Enter`)
  - Back to the normal; somewhat strange...

Signal Handling I: Using `signal` (cont.)

- Besides, the original signal system has two other weaknesses.
  - W1) You don't know why the signal was sent
  - Tells the handler which signal invoked it (using signum)
    - inthandler is invoked with SIGINT, and quithandler with SIGQUIT.
  - However, it does not tell the handler why the signal was generated

Signal Handling I: Using `signal` (cont.)

- W2) You cannot safely block other signals while in a handler
  - Say our program wants to ignore `SIGQUIT` when answering `SIGINT`
  - A modified version looks like the one below. Any problem??

```
void inthandler(int s)  // A handler for SIGINT
{
    int rv ;
    void (*prev_qhandler)();                /* holds prev handler */
    prev_qhandler = signal(SIGQUIT, SIG_IGN); /* ignore QUIT's */
    ...
    signal( SIGQUIT, prev_qhandler );        /* restore handler */
}
```

  - Prob. 1: Handling each signal happens in sequence. Both should happen at the same time.
  - Prob. 2: Do not want to ignore SIGQUIT; want to block it until SIGINT processed

Signal Handling II: `sigaction`

- `sigaction`: the POSIX replacement for `signal`
  - POSIX (Portable Operating System Interface): a family of standards specified by IEEE for maintaining compatibility between operating systems (from Wiki)
    - Defines API, along with command line shells and utility interfaces, for software compatibility with variants of Unix/Linux and perhaps other operating systems
    - Tries to minimize any incompatibility issue that might occur to a system program that would run on a different operating system
  - Specifies “which” signal to handle and “how” to respond to that signal

Signal Handling II: `sigaction` (cont.)

- `sigaction`: the POSIX replacement for `signal`
  - `struct sigaction`: a struct to support customized signal handling; specifies how to invoke an old- or new-style handler

sigaction	
PURPOSE	Specify handling for a signal
INCLUDE	`#include <signal.h>`

USAGE	`res = sigaction(int signum,`
	`const struct sigaction *action,`
	`struct sigaction *prevaction);`

ARGS	`signum` signal to handle `action` pointer to struct describing action `prevaction` pointer to struct to receive old action
RETURNS	`-1` on error `0` on success

Signal Handling II: `sigaction` (cont.)

- Customized Signal Handling: Use `struct sigaction`

Only one between these

```
struct sigaction{ // struct, not a function
   /* use only one of the following two */
   void (*sa_handler)(int); // OLD: SIG_DFL, SIG_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
}
```

Using an old-style handler	Using a new-style handler

`struct sigaction action;`
`action.`**`sa_handler`** `=`
`handler_old``;`
	`struct sigaction action;`
	`action.`**`sa_sigaction`** `=`
	`handler_new``;`

- How do you tell the kernel you are using the new-style handler?
  - Easy, just set the `SA_SIGINFO` bit in (`sigaction.`)**`sa_flags`**

Signal Handling II: `sigaction` (cont.)

- sa_flags
  - A set of bits that control how the handler answers about the four questions

Flag	Meaning
SA_RESETHAND	Reset the handler when invoked. This enables mousetrap mode.
SA_NODEFER	Turn off automatic blocking of a signal while it is being handled. This allows recursive calls to a signal handler.
SA_RESTART	Restart, rather than return, system calls on slow devices and similar system calls. This enables BSD mode.
SA_SIGINFO	Use the value in `sa_sigaction` for the handler function. If this bit is not set, use the value in `sa_handler`. If the `sa_sigaction` value is used, that handler function is passed not only the signal number, but also pointers to structs containing information about why and how the signal was generated.

- sa_mask: an important technique for data corruption prevention
  - Decide if we would like to block any other signal while in the handler
  - Contains a set of signals to block

Signal Handling II: `sigaction` (cont.)

```c
/* sigactdemo.c
 *               purpose: shows use of sigaction()
 *               feature: blocks ^\ while handling ^C
 *                        does not reset ^C handler, so two kill
 */

#include        <stdio.h>
#include        <unistd.h>
#include        <signal.h>
#define INPUTLEN        100

void inthandler(int);

int main()
{
        struct sigaction newhandler;            /* new settings         */
        sigset_t         blocked;               /* set of blocked sigs */
        void             inthandler();          /* the handler          */
        char             x[INPUTLEN];

        /* load these two members first */
        newhandler.sa_handler = inthandler;      /* handler function    */
        newhandler.sa_flags = SA_RESETHAND | SA_RESTART;  /* options    */

        /* then build the list of blocked signals */
        sigemptyset(&blocked);                   /* clear all bits      */
        sigaddset(&blocked, SIGQUIT);            /* add SIGQUIT to list */

        newhandler.sa_mask = blocked;            /* store blockmask     */

        if ( sigaction(SIGINT, &newhandler, NULL) == -1 )
                perror("sigaction");
        else
                while( 1 ){
                        fgets(x, INPUTLEN, stdin);
                        printf("input: %s", x);
                }

        return 0;
}
```

- `sigactdemo.c`

```c
void inthandler(int s)
{
        printf("Called with signal %d\n", s);
        sleep(s);
        printf("done handling signal %d\n", s);
}
```

Protecting Data from Corruption

- Critical Sections
  - A section of code that modifies a data structure if interruptions to that section of code can produce incomplete or damaged data
    - e.g., A postal office with interrupting telephone calls and knocks on the door
- You must determine which parts of code are critical sections and arrange to protect those sections
  - When you write your code with signals
- Note that critical sections are “not always” in signal handlers
  - Actually, many of them are in regular flow of a program
- What is the “simplest” way to protect critical sections?
  - “ignoring” or “blocking” signals that call handlers that use or change the data

Blocking Signals: `sigprocmask()` and `sigsetops`

- We can block signals (i) at the signal-handler level and (ii) at the process level.
- How can we block signals in a "signal handler"? --- (i)
  - Set the `sa_mask` member
- How can we block signals for a "process"? --- (ii)
  - A process has a set of its blocking signals, called signal mask.
  - To modify that set of blocked signals, use `sigprocmask()`:
    - Takes a set of signals, and
    - Uses (in an atomic operation, such that no other process can alter the mask) that set to change the current set of blocked signals

How to Modify the Signal Mask?

- `sigprocmask()` modifies the present signal mask with the signals in `*sigs` as specified by a how value.
- `sigset_t`: defines a set of signals to be added or deleted for mask

sigprocmask		
PURPOSE	Modify current signal mask	
INCLUDE	`#include <signal.h>`	

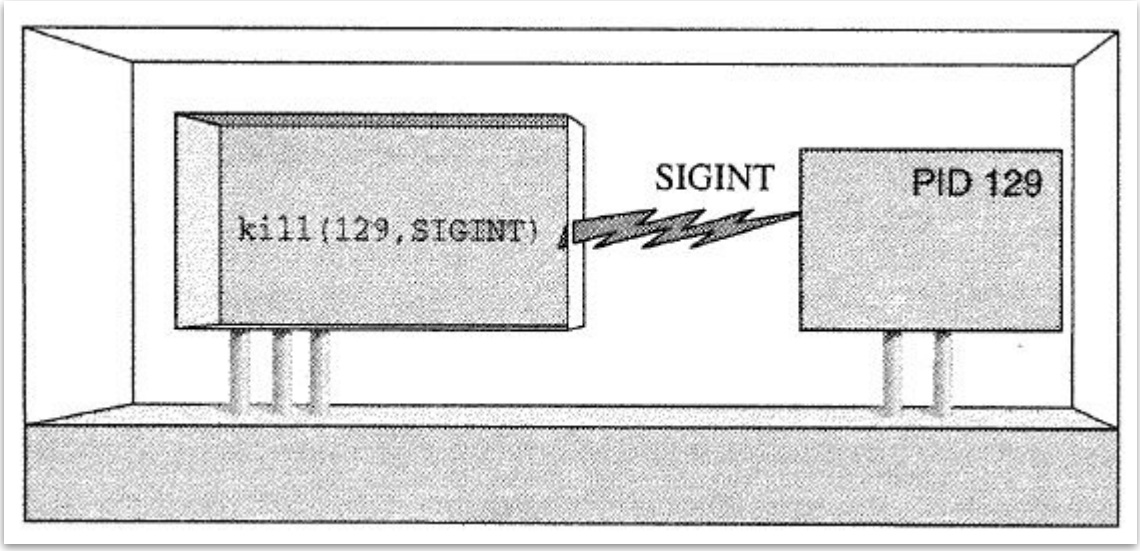
USAGE	`int res = sigprocmask( int how,`
	`const sigset_t *sigs,`
	`sigset_t *prev);`	

ARGS	how	how to modify the signal mask
	sigs	pointer to list of signals to use
	prev	pointer to list of previous signal mask (or NULL)
RETURNS	-1	on error
	0	on success

Building Signal Sets with `sigsetops` (Signal Set Operations)

- `sigset_t`
  - Abstract set of signals that has methods for signal addition or removal
  - `sigaddset(sigset_t* setp, int signum)`
    - Add `signum` to the set pointed by `setp`
  - `sigdelset(sigset_t* setp, int signum)`
    - Remove `signum` from the set pointed to by `setp`
  - `sigfillset(sigset_t* setp)`
    - Add all signals to the list pointed to by `setp`
  - `sigemptyset(sigset_t* setp)`
    - Clear all signals from the list pointed to by `setp`

`kill`: Sending Signals from a Process



< A process uses kill() to send a signal >

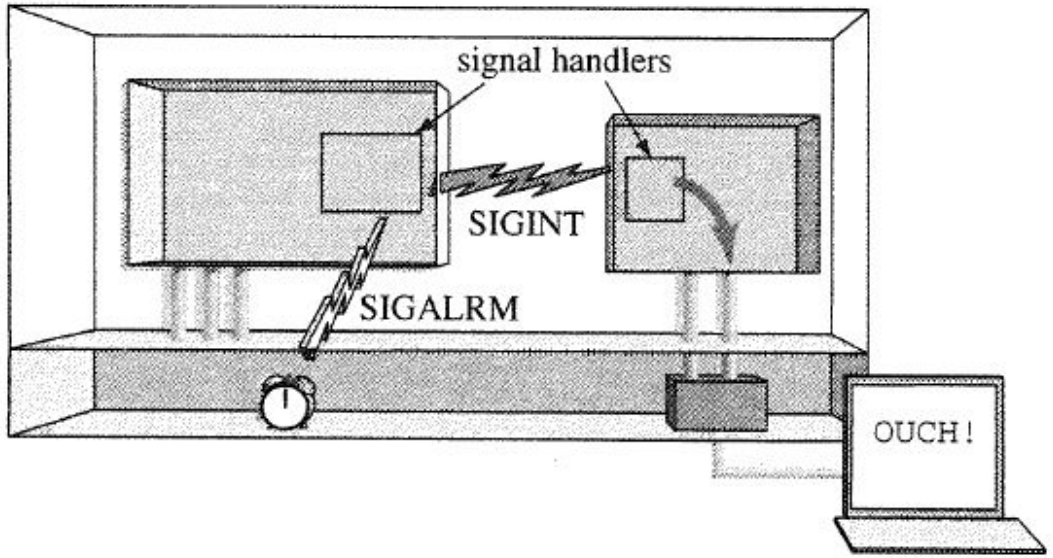
- Signals arise from many different objects:
  - Interval timers, the terminal driver, the kernel, and processes
- A process can send a signal to another process. How?
  - By using the kill system call

`kill`: Sending Signals from a Process (cont.)

- Lessons
  - The process sending the signal must have the same user ID as the target process
  - Or the sending process must be owned by the superuser
  - The kill program is used at inter-process communication (IPC) (on the local machine)

kill		
PURPOSE	Send a signal to a process	
INCLUDE	#include <sys/types.h> #include <signal.h>	
USAGE	int kill(pid_t pid, int sig)	
ARGS	pid sig	process id of target signal to throw
RETURNS	-1 0	on error on success

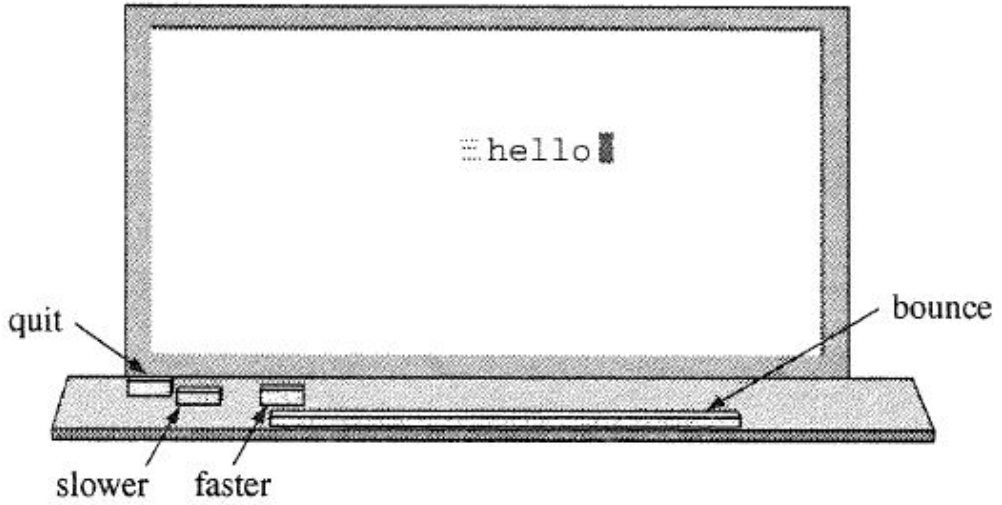
`kill`: Sending Signals from a Process (cont.)



< Complex use of signals >

- Implications for Interprocess Communication
  - Suppose that P1 sets its interval timer, which after the specified interval generates a signal, or SIGALRM, and calls a signal handler receiving it
  - P1's signal handler then sends SIGINT to P2 using kill().
  - P2 with a signal handler for SIGINT, receives it and prints out "OUCH!"

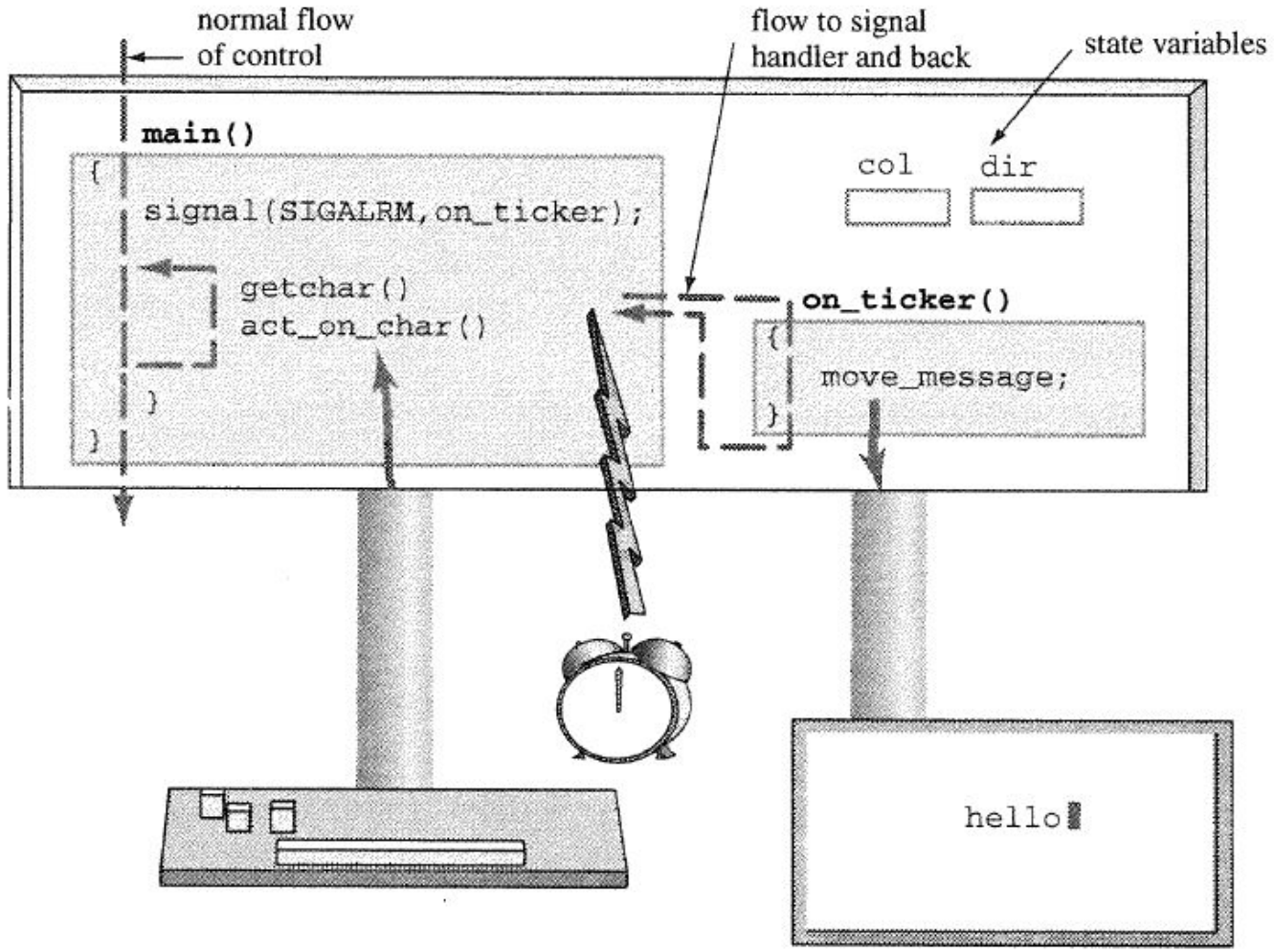
Using Timers and Signals: Video Games



< `bounce1d` in action: user-controlled animation >

- This program moves a single word smoothly across the screen.
  - Space bar: the message reverses direction.
  - “s” and “f”: make the message move slower and faster, respectively.
  - “Q” : quits the game
  - Besides, two variables for direction (“left” or “right”) and speed of motion (causing a “long” or “short” delay the interval between timer ticks)
- Let’s add user control of direction and speed to the program
  - State variables and event handling come into play for this extra work

Using Timers and Signals: Video Games (cont.)



< User input changes values. Values control action. >

bounce1d.c: Controlled Animation on a Line

```c
/* bounce1d.c
 *      purpose animation with user controlled speed and direction
 *      note    the handler does the animation
 *              the main program reads keyboard input
 *      compile cc bounce1d.c set_ticker.c -lcurses -o bounce1d
 */
#include        <stdio.h>
#include        <curses.h>
#include        <signal.h>
#include        <string.h>

/* some global settings main and the handler use */

#define MESSAGE "hello"
#define BLANK   "     "

void move_msg(int);
int set_ticker(int);

int     row;    /* current row          */
int     col;    /* current column       */
int     dir;    /* where we are going   */
```

```c
int main()
{
        int     delay;          /* bigger => slower     */
        int     ndelay;         /* new delay            */
        int     c;              /* user input           */
        void    move_msg(int);  /* handler for timer    */

        initscr();
        crmode();
        noecho();
        clear();

        row   = 10;             /* start here           */
        col   = 0;
        dir   = 1;              /* add 1 to row number  */
        delay = 200;            /* 200ms = 0.2 seconds  */

        move(row,col);          /* get into position    */
        addstr(MESSAGE);        /* draw message         */
        signal(SIGALRM, move_msg );
        set_ticker( delay );

        while(1)
        {
                ndelay = 0;
                c = getch();
                if ( c == 'Q' ) break;
                if ( c == ' ' ) dir = -dir;
                if ( c == 'f' && delay > 2 ) ndelay = delay/2;
                if ( c == 's' ) ndelay = delay * 2 ;
                if ( ndelay > 0 )
                        set_ticker( delay = ndelay );
        }
        endwin();
        return 0;
}
```

`bounce1d.c`: Controlled Animation on a Line (cont.)

```c
void move_msg(int signum)
{
        signal(SIGALRM, move_msg);      /* reset, just in case  */
        move( row, col );
        addstr( BLANK );
        col += dir;                     /* move to new column   */
        move( row, col );               /* then set cursor      */
        addstr( MESSAGE );              /* redo message         */
        refresh();                      /* and show it          */

        /*
         * now handle borders
         */
        if ( dir == -1 && col <= 0 )
                dir = 1;
        else if ( dir == 1 && col+strlen(MESSAGE) >= COLS )
                dir = -1;
}
```

```
$ gcc -o bounce1d bounce1d.c
set_ticker.c -lcurses
```

```c
#include        <stdio.h>
#include        <sys/time.h>
#include        <signal.h>

/*
 *      set_ticker.c
 *          set_ticker( number_of_milliseconds )
 *                  arranges for the interval timer to issue
 *                  SIGALRM's at regular intervals
 *          returns -1 on error, 0 for ok
 *
 *      arg in milliseconds, converted into micro seoncds
 */


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

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

        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);
}
```

Summary

- A video game responds to time and to user input
  - OS also reacts to time and to input from external devices
- The `curses` library is a set of functions that programs can call to manage the appearance of a text screen
- A process schedules events by setting “timers”; each set to ring once or at regular intervals
  - Each process can run three separate timers; real, virtual, and profile

Summary (cont.)

- Handling a single signal is easy while doing multiple ones is more complicated
  - A process can ignore or block signals; it should determine how to react to them
  - Should tell the kernel which signals to block or ignore at which times
- Some functions perform uninterruptable operations (in critical section)
  - A program can protect them via careful use of signal masks