Programming for Humans

Terminal Control and Signals

류은경
ekryu@knu.ac.kr

Objectives

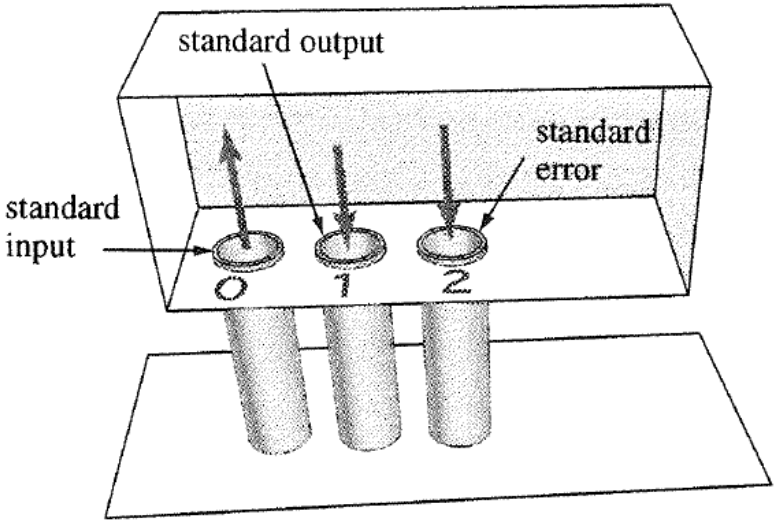
- **Ideas and Skills**
  - Software tools vs. user programs
  - Reading and changing settings of the terminal driver
  - Modes of the terminal driver
  - Nonblocking input
  - Timeouts on user input
  - Introduction to signals: How Ctrl-C works

- **System Calls**
  - `fcntl`
  - `signal`

Software Tools

- **Programs** don't distinguish between disk files and devices :
  - Ex) `who`, `ls`, `sort`, `uniq`, `grep`, `tr`, `du`

- These tools read from **stdin** and write to **stdout**



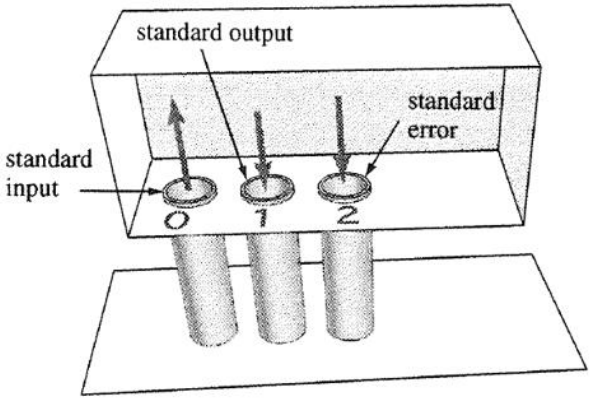
Fact: Most processes automatically have the first three file descriptors open. They do not need to call open() to make these connections.

FIGURE 6.1

The three standard file descriptors.

- **Input and output for these programs** can be easily attached to all sorts of connections:

```
$ sort x
$ sort x > outputfile
$ sort x > /dev/lp
$ who | tr '[a-z]' '[A-Z]'
```



Device-Specific Programs

- **Programs to control devices :**
  - e.g., scanners, CD recorders, tape drives, digital cameras

- **To explore the ideas and techniques of writing device-specific programs,**
  - we examine programs (**user programs**) that interact with **terminals**

User Programs : Device-Specific Programs

- **Ex) user programs :**
  `vi,emacs,pine,more,lynx,hangman,robots`

- These programs adjust **terminal driver settings**

- **Common concerns of user programs :**
  - (a) Immediate response to keys
  - (b) Limited input set
  - (c) Timeout on input
  - (d) Resistance to Ctrl-C

- **A short translation program :**

```
/* rotate.c : map a->b, b->c, .. z->a
 *   purpose: useful for showing tty modes
 */

#include   <stdio.h>
#include   <ctype.h>
int main()
{
    int c;
    while ( ( c=getchar() ) != EOF ){
        if ( c == 'z' )
            c = 'a';
        else if (islower(c))
            c++;
        putchar(c);
    }
}
```

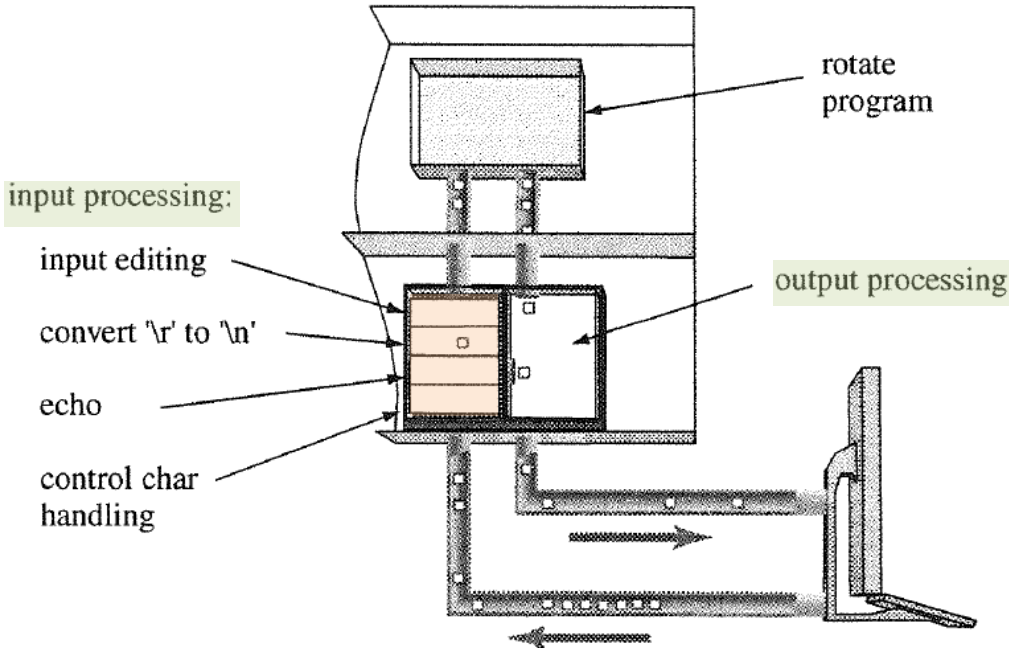
Canonical Mode: Buffering and Editing

- Run the program using the **default settings:**

```
$ cc rotate.c -o rotate
$ ./rotate
abx<-cd
bcde
efgCtrl-C
$
```

back space

```
$ cc rotate.c -o rotate
$ ./rotate
abx<-cd
bcde
efgCtrl-C
$
```



buffering, echoing, editing, **control key processing** : done **by the terminal driver**

FIGURE 6.3

Processing layers in the terminal driver.

Noncanonical Processing : NO Buffering and Editing

```
$ stty -icanon ; ./rotate
abbcxy^?cdde
effggh^C
$ stty icanon
```

Turning off canonical mode processing in the driver

```
$ stty -icanon -echo ; ./rotate
bcy^?de
fgh
$ stty icanon echo
```

(Note: You won't see this. Why?)

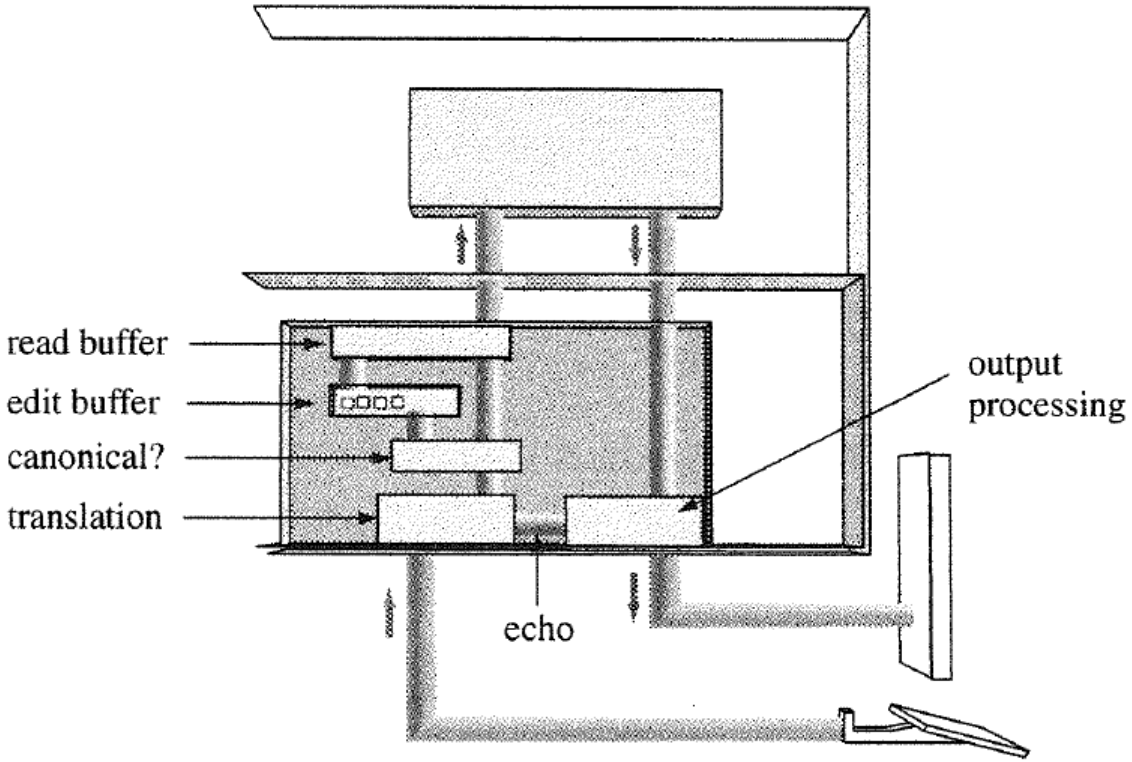
Output comes only from the program

Terminal Modes : Summary

- **canonical mode (cooked mode)**
  - Buffering and editing **enabled**
  - This is the **default** mode
  - Input is sent **after Enter is pressed**

- **noncanonical mode (cbreak or crmode)**
  - Buffering and editing **disabled**
  - Terminal driver still handles some processing (e.g., Ctrl-C, newline ↔ carriage return)

- **raw mode (non-anything mode)**
  - **All processing disabled**
  - Input is passed **directly** to the program, with **no interpretation**



The terminal driver is a complex set of routines in the kernel

FIGURE 6.4

Major components of the terminal driver.

- **A shell script for a bank machine(ATM):**

```
#!/bin/sh
#
# atm.sh - a wrapper for two programs
#

while true
do
     do_a_transaction      # run a program
     if play_again         # run our program
     then
          continue         # if "y" loop back
     fi
     break                 # if "n" break
done
```

does the work of the ATM

obtains a yes or no answer from the user

- **The logic of play_again.c:**

prompt user with question
accept input
if “y”, return 0
if “n”, return 1

```
while true
do
        do_a_transaction
        if play_again
        then
                continue
        fi
        break
done
```

```c
/* play_again0.c
 *      purpose: ask if user wants another transaction
 *       method: ask a question, wait for yes/no answer
 *      returns: 0=>yes, 1=>no
 *       better: eliminate need to press return
 */
#include        <stdio.h>

#define QUESTION        "Do you want another transaction"
int get_response( char * );
int main(void)
{
        int     response;

       response = get_response(QUESTION);      /* get some answer      */

        return response;
}
```

```
int get_response(char *question)
/*
 * purpose: ask a question and wait for a y/n answer
 *  method: use getchar and ignore non y/n answers
 * returns: 0=>yes, 1=>no
 */
{
        printf("%s (y/n)?", question);
        while(1){
                switch( getchar() ){
                        case 'y':
                        case 'Y': return 0;
                        case 'n':
                        case 'N':
                        case EOF: return 1;
                }
        }
}
```

- **Two Problems with play_again0**
  - 1. User has to press the **Enter key**
  - 2. Program receives and processes an **entire line** of data when the user presses Enter

```
$ play_again0
Do you want another transaction (y/n)? sure thing!
```

Ex: play_again1.c – immediate response

```
/* play_again1.c
 *      purpose: ask if user wants another transaction
 *       method: set tty into char-by-char mode, read char, return result
 *      returns: 0=>yes, 1=>no
 *       better: do no echo inappropriate input
 */
#include        <stdio.h>
#include        <termios.h>

#define QUESTION        "Do you want another transaction"

int get_response(char *);
void set_crmode(void);
void tty_mode(int);

int main(void)
{
        int     response;

        tty_mode(0);                            /* save tty mode        */
        set_crmode();                           /* set chr-by-chr mode  */
        response = get_response(QUESTION);      /* get some answer      */
        tty_mode(1);                            /* restore tty mode     */
        return response;
}
```

```c
int get_response(char *question)
/*
 * purpose: ask a question and wait for a y/n answer
 *  method: use getchar and complain about non y/n answers
 * returns: 0=>yes, 1=>no
 */
{
        int input;
        printf("%s (y/n)?", question);
        while(1){
                switch( input = getchar() ){
                        case 'y':
                        case 'Y': return 0;
                        case 'n':
                        case 'N':
                        case EOF: return 1;
                        default:
                                printf("\ncannot understand %c, ", input);
                                printf("Please type y or no\n");
                }
        }
}
```

```
void set_crmode(void)
/*
 * purpose: put file descriptor 0 (i.e. stdin) into chr-by-chr mode
 *  method: use bits in termios
 */
{
        struct  termios ttystate;

        tcgetattr( 0, &ttystate);               /* read curr. setting   */
        ttystate.c_lflag        &= ~ICANON;     /* no buffering         */
        ttystate.c_cc[VMIN]     =  1;           /* get 1 char at a time */
        tcsetattr( 0 , TCSANOW, &ttystate);     /* install settings     */
}
```

```
/* how == 0 => save current mode,  how == 1 => restore mode */
void tty_mode(int how)
{
        static struct termios original_mode;
        if ( how == 0 )
                tcgetattr(0, &original_mode);
        else
                tcsetattr(0, TCSANOW, &original_mode);
}
```

```
$ make play_again1
cc     play_again1.c   -o play_again1
$ ./play_again1
Do you want another transaction (y/n)?s
cannot understand s, Please type y or no
u
cannot understand u, Please type y or no
r
cannot understand r, Please type y or no
e
cannot understand e, Please type y or no
y$
```

→ **without** waiting for the **Enter key!**

Ex: play_again2.c – ignore illegal keys

```
/* play_again2.c
 *      purpose: ask if user wants another transaction
 *       method: set tty into char-by-char mode and no-echo mode
 *               read char, return result
 *      returns: 0=>yes, 1=>no
 *       better: timeout if user walks away
 *
 */
#include        <stdio.h>
#include        <termios.h>

#define QUESTION        "Do you want another transaction"

int get_response(char *);
void set_cr_noecho_mode(void);
void ttv mode(int);
int main(void)
{
        int     response;

        tty_mode(0);                            /* save mode */
        set_cr_noecho_mode();                   /* set -icanon, -echo   */
        response = get_response(QUESTION);      /* get some answer      */
        tty_mode(1);                            /* restore tty state    */
        return response;
}
```

```c
int get_response(char *question)
/*
 * purpose: ask a question and wait for a y/n answer
 *  method: use getchar and ignore non y/n answers
 * returns: 0=>yes, 1=>no
 */
{
        printf("%s (y/n)?", question);
        while(1){
                switch( getchar() ){
                        case 'y':
                        case 'Y': return 0;
                        case 'n':
                        case 'N':
                        case EOF: return 1;
                }
        }
}
```

※ No error reports for illegal input.
Nothing shows up!

```
void set_cr_noecho_mode(void)
/*
 * purpose: put file descriptor 0 into chr-by-chr mode and noecho mode
 *  method: use bits in termios
 */
{
        struct  termios ttystate;

        tcgetattr( 0, &ttystate);               /* read curr. setting   */
        ttystate.c_lflag        &= ~ICANON;     /* no buffering         */
        ttystate.c_lflag        &= ~ECHO;       /* no echo either       */
        ttystate.c_cc[VMIN]     =   1;          /* get 1 char at a time */
        tcsetattr( 0 , TCSANOW, &ttystate);     /* install settings     */
}


/* how == 0 => save current mode,  how == 1 => restore mode */
void tty_mode(int how)
{
        static struct termios original_mode;
        if ( how == 0 )
                tcgetattr(0, &original_mode);
        else
                tcsetattr(0, TCSANOW, &original_mode);
}
```

play_again2.c

- **Blocking Input**
  - `getchar()` or `read()` waits for input: **blocked**

- **What if**
  - this program were used at a real ATM and a **customer** wandered **away without pressing y or n**? …

- It needs one more feature; **Timeout** feature

Nonblocking Input: play_again3.c

- **How to turn off input blocking:**
  - Use `fcntl`() or `open`()
  - In play_again3, `fcntl`() sets the **O_NDELAY** or **O_NONBLOCK** flag

- **Behavior with O_NDELAY:**
  - If data is available:
    - `read`() returns the number of characters read
  - If no data is available:
    - `read`() returns **0**
  - If an error occurs:
    - `read`() returns **-1**

Ex: play_again3.c

→ **Using fcntl () + O_NDELAY (or O_NONBLOCK) flag for fd**

```
/* play_again3.c
 *      purpose: ask if user wants another transaction
 *       method: set tty into chr-by-chr, no-echo mode
 *               set tty into no-delay mode
 *               read char, return result
 *      returns: 0=>yes, 1=>no, 2=>timeout
 *       better: reset terminal mode on Interrupt
 */
#include        <stdio.h>
#include        <termios.h>
```

```
#include        <fcntl.h>
#include        <string.h>

#define ASK           "Do you want another transaction"
#define TRIES       3                                  /* max tries */
#define SLEEPTIME   2                                  /* time per try */
#define BEEP       putchar('\a')                       /* alert user */

main()
{
        int     response;

        tty_mode(0);                            /* save current mode    */
        set_cr_noecho_mode();                   /* set -icanon, -echo   */
        set_nodelay_mode();                     /* noinput => EOF       */
        response = get_response(ASK, TRIES);    /* get some answer      */
        tty_mode(1);                            /* restore orig mode    */
        return response;
}
```

```c
get_response( char *question , int maxtries)
/*
 * purpose: ask a question and wait for a y/n answer or maxtries
 *  method: use getchar and complain about non-y/n input
 * returns: 0=>yes, 1=>no, 2=>timeout
 */
{
        int     input;

       printf("%s (y/n)?", question);                    /* ask          */
       fflush(stdout);                                   /* force output */
        while ( 1 ){
              sleep(SLEEPTIME);                          /* wait a bit   */
              input = tolower(get_ok_char());            /* get next chr */
               if ( input == 'y' )
                       return 0;
               if ( input == 'n' )
                       return 1;
               if ( maxtries-- == 0 )                    /* outatime?    */
                      return 2;                          /* sayso        */
               BEEP;
        }
}
```

```
/*
 *  skip over non-legal chars and return y,Y,n,N or EOF
 */
get_ok_char()
{
        int c;
        while( ( c = getchar() ) != EOF && strchr("yYnN",c) == NULL )
                ;
        return c;
}
```

※ returns a pointer to the first occurrence of the character c in the string s

```
set_nodelay_mode()
/*
 * purpose: put file descriptor 0 into no-delay mode
 *  method: use fcntl to set bits
 *   notes: tcsetattr() will do something similar, but it is complicated
 */
{
        int     termflags;
        termflags = fcntl(0, F_GETFL);          /* read curr. settings  */
        termflags |= O_NDELAY;                  /* flip on nodelay bit  */
        fcntl(0, F_SETFL, termflags);           /* and install 'em      */
}
```

```
set_cr_noecho_mode()
/*
 * purpose: put file descriptor 0 into chr-by-chr mode and noecho mode
 *  method: use bits in termios
 */
{
        struct  termios ttystate;

       tcgetattr( 0, &ttystate);                /* read curr. setting   */
       ttystate.c_lflag        &= ~ICANON;      /* no buffering         */
       ttystate.c_lflag        &= ~ECHO;        /* no echo either       */
       ttystate.c_cc[VMIN]     =  1;            /* get 1 char at a time */
       tcsetattr( 0 , TCSANOW, &ttystate);      /* install settings     */
}
```

```
/* how == 0 => save current mode,  how == 1 => restore mode */
/* this version handles termios and fcntl flags             */

tty_mode(int how)
{
        static struct termios original_mode;
        static int            original_flags;
        if ( how == 0 ){
                tcgetattr(0, &original_mode);
                original_flags = fcntl(0, F_GETFL);
        }
        else {
                tcsetattr(0, TCSANOW, &original_mode);
                fcntl( 0, F_SETFL, original_flags);
        }
}
```

- **Problem with play_again3 :**
  What happens if the user presses **Ctrl-C?**

```
$ make play_again3
cc     play_again3.c   -o play_again3
$ ./play_again3
Do you want another transaction (y/n)?  press Ctrl-C now
$ logout
Connection to host closed.
bash$
```

- **How did it happen?**

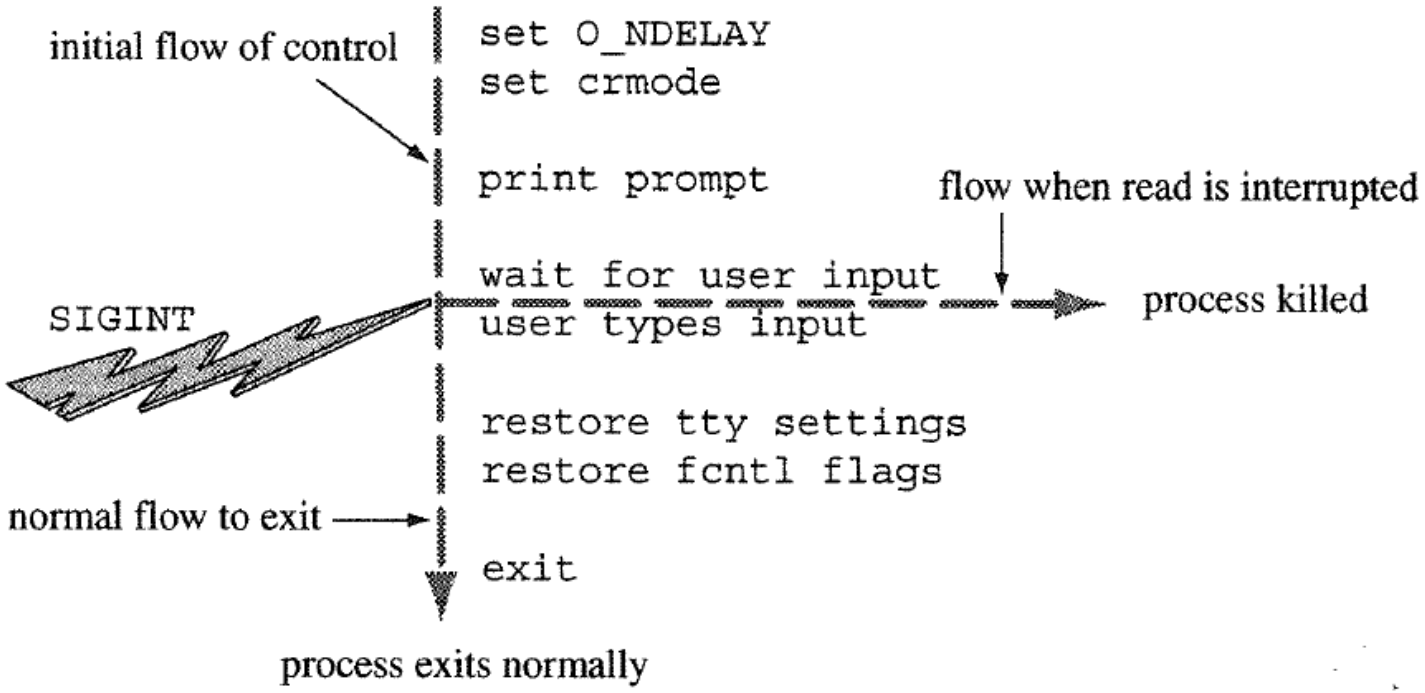
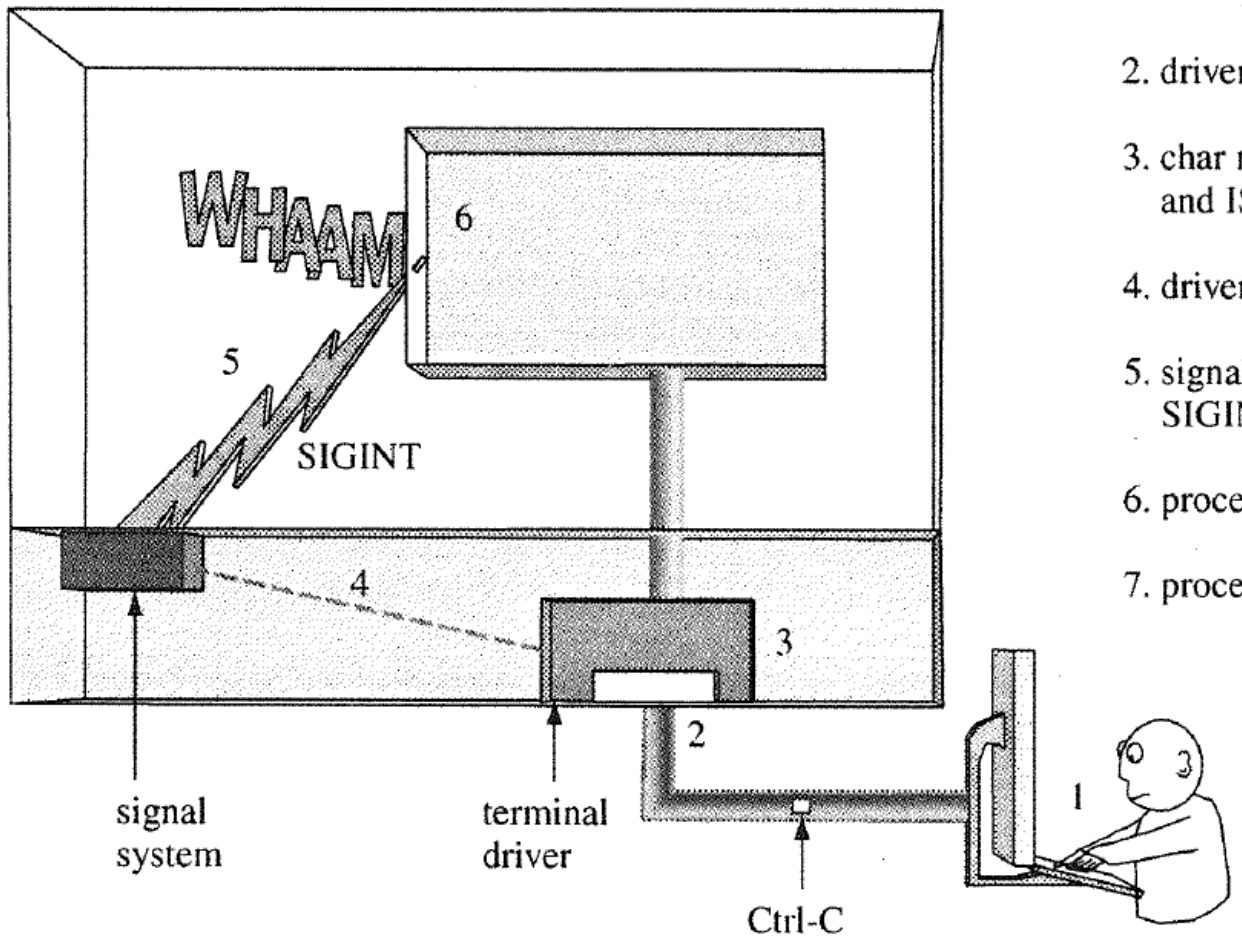


FIGURE 6.5

Ctrl-C kills a program. It leaves terminal unrestored.



1. User presses Ctrl-C
2. driver receives char
3. char matches VINTR and ISIG is on
4. driver calls signal system
5. signal system sends SIGINT to process
6. process receives SIGINT
7. process dies.

FIGURE 6.6
How Ctrl-C works.

What is a Signal?

- A **signal** is **a one-word message** from kernel to process
  - Each signal has a **numerial code**;
  - Pressing `Ctrl-C` sends an **interrupt signal** (`SIGINT`) to the current process.
  - `SIGINT` usually is **signal number 2**

- **Where Signals come from : (1)(2)(3)**

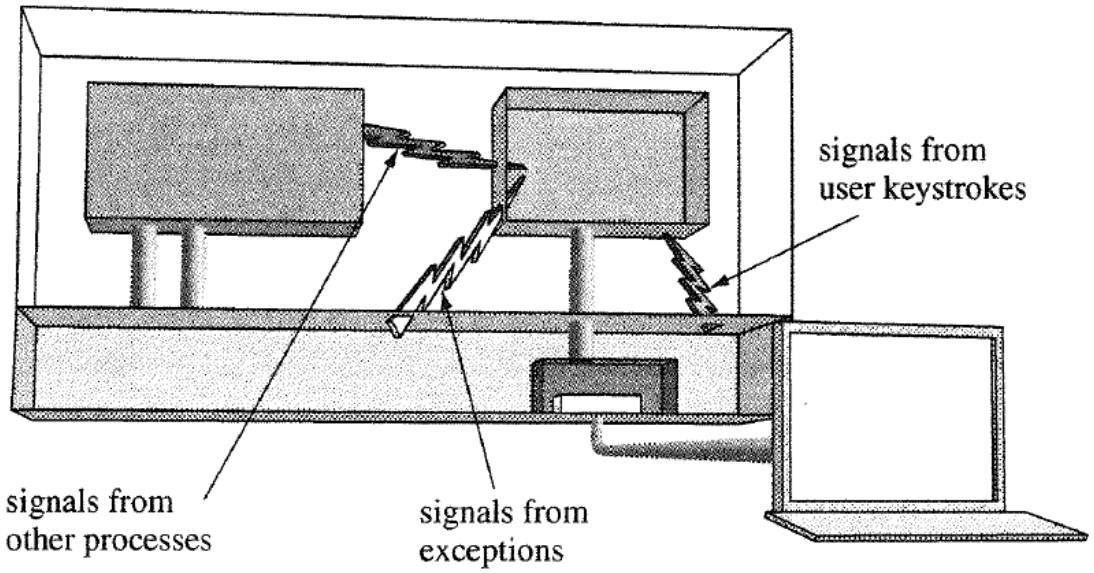


FIGURE 6.7

Three sources of signals.

Types of Signals

- **Synchronous signals**
  - Caused by something the **process itself does**
  - Ex) **divide by zero**, segmentation fault

- **Asynchronous signals**
  - Caused by events **outside the process**
  - Ex) **user presses Ctrl-C**

- **Sample Signal List:**
  - /usr/include/signal.h

※ $ man 7 signal

```
#define SIGHUP    1   /* hangup, generated when terminal disconnects */
#define SIGINT    2   /* interrupt, generated from terminal special char */
#define SIGQUIT   3   /* (*) quit, generated from terminal special char */
#define SIGILL    4   /* (*) illegal instruction (not reset when caught)*/
#define SIGTRAP   5   /* (*) trace trap (not reset when caught) */
#define SIGABRT   6   /* (*) abort process */
#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 (specification exception) */
#define SIGSEGV  11   /* (*) segmentation violation */
#define SIGSYS   12   /* (*) bad argument to system call */
#define SIGPIPE  13   /* write on a pipe with no one to read it */
#define SIGALRM  14   /* alarm clock timeout */
#define SIGTERM  15   /* software termination signal */
```

What Can a Process Do about a Signal?

- A process has **three options** when it receives a signal like SIGINT (e.g., from Ctrl-C):
  - **Accept the default action** (usually death)

    ```
    signal(SIGINT, SIG_DFL);
    ```

  - **Ignore the signal**

    ```
    signal(SIGINT, SIG_IGN);
    ```

  - **Call a function (Custom Handler)**

    ```
    signal(signum, functionname);
    ```

    Signal handler

signal	
PURPOSE	Simple signal handling
INCLUDE	`#include <signal.h>`
USAGE	`result = signal (int signum, void (*action)(int))`
ARGS	signum the signal to respond to action how to respond
RETURNS	-1 if error prevaction if success

- Function name (custom handler)
- SIG_IGN → ignore the signal
- SIG_DFL → use default behavior

Example of Signal Handling

- **Ex 1: Catching a Signal**

```
/* sigdemo1.c - shows how a signal handler works.
 *            - run this and press Ctrl-C a few times
 */

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

main()
{
        void    f(int);                 /* declare the handler  */
        int     i;
        signal( SIGINT, f );            /* install the handler  */
        for(i=0; i<5; i++ ){            /* do something else    */
                printf("hello\n");
                sleep(1);
        }
}

void f(int signum)                      /* this function is called */
{
        printf("OUCH!\n");
}
```

```
$ ./sigdemo1
hello
hello      press Ctrl-C now
OUCH!
hello      press Ctrl-C now
OUCH!
hello
hello
$
```

```
$ ./sigdemo1
hello
hello      press Ctrl-C now
OUCH!
hello      press Ctrl-C now
OUCH!
hello
hello
$
```

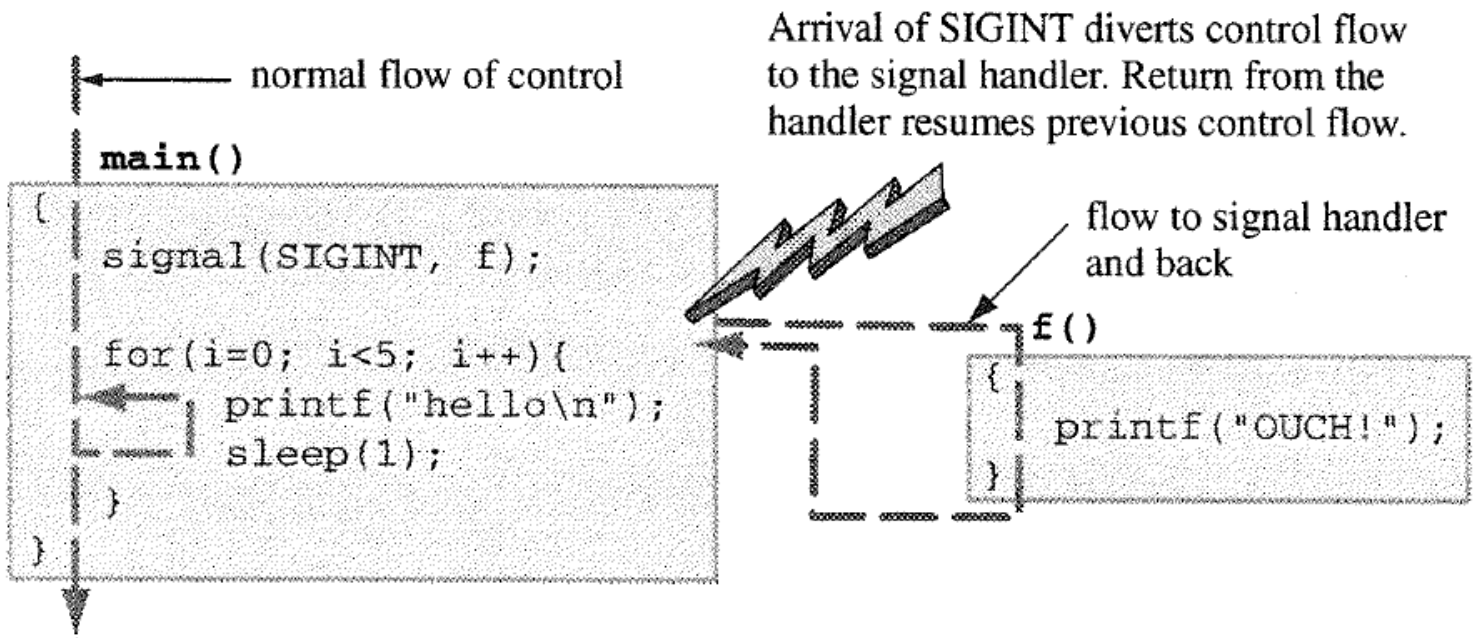


FIGURE 6.8

A signal causes a subroutine call.

Ex 2: Ignoring a Signal

```
/* sigdemo2.c - shows how to ignore a signal
 *            - press Ctrl-\ to kill this one
 */

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

main()
{
        signal( SIGINT, SIG_IGN );

        printf("you can't stop me!\n");
        while( 1 )
        {
                sleep(1);
                printf("haha\n");
        }
}
```

```
$ ./sigdemo2
you can't stop me!
haha
haha
haha      press Ctrl-C now
haha      press Ctrl-C nowpress Ctrl-C now
haha
haha
haha      press ^\ now
Quit
$
```

Interrupt signals

quit signal

```
$ ./sigdemo2
you can't stop me!
haha
haha
haha      press Ctrl-C now
haha      press Ctrl-C nowpress Ctrl-C now
haha
haha
haha      press ^\ now
Quit
$
```

Interrupt signals

quit signal

A process can tell the kernel it wants to ignore SIGINT.

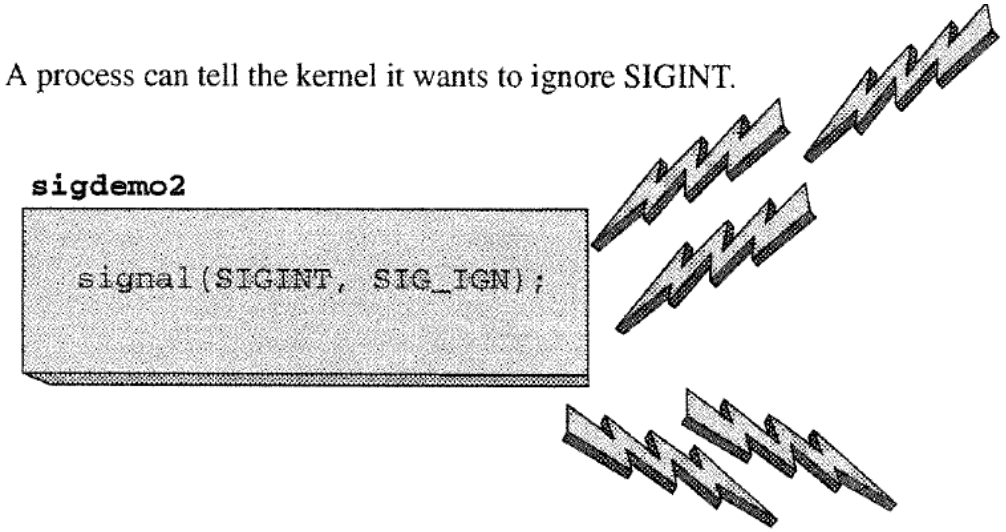


FIGURE 6.9

The effect of `signal(SIGINT, SIG_IGN)`.

play_again4.c

```
/* play_again4.c
 *      purpose: ask if user wants another transaction
 *       method: set tty into chr-by-chr, no-echo mode
 *               set tty into no-delay mode
 *               read char, return result
 *               resets terminal modes on SIGINT, ignores SIGQUIT
 *      returns: 0=>yes, 1=>no, 2=>timeout
 *       better: reset terminal mode on Interrupt
 */
#include        <stdio.h>
#include        <termios.h>
#include        <fcntl.h>
#include        <string.h>
#include        <signal.h>

#define ASK           "Do you want another transaction"
#define TRIES      3                                  /* max tries */
#define SLEEPTIME  2                                  /* time per try */
#define BEEP       putchar('\a')                      /* alert user */
```

```
main()
{
        int     response;
        void    ctrl_c_handler(int);

        tty_mode(0);                            /* save current mode    */
        set_cr_noecho_mode();                   /* set -icanon, -echo   */
        set_nodelay_mode();                     /* noinput => EOF       */
        signal( SIGINT, ctrl_c_handler );       /* handle INT           */
        signal( SIGQUIT, SIG_IGN );             /* ignore QUIT signals  */
        response = get_response(ASK, TRIES);    /* get some answer      */
        tty_mode(1);                            /* reset orig mode      */
        return response;
}
```

```
get_response( char *question , int maxtries)
/*
 * purpose: ask a question and wait for a y/n answer or timeout
 *  method: use getchar and complain about non-y/n input
 * returns: 0=>yes, 1=>no
 */

{
        int     input;

        printf("%s (y/n)?", question);                  /* ask          */
        fflush(stdout);                                 /* force output */
        while ( 1 ){
                sleep(SLEEPTIME);                       /* wait a bit   */
                input = tolower(get_ok_char());         /* get next chr */
                if ( input == 'y' )
                        return 0;
                if ( input == 'n' )
                        return 1;
                if ( maxtries-- == 0 )                  /* outatime?    */
                        return 2;                       /* sayso        */
                 BEEP;
        }
}
/*
 *  skip over non-legal chars and return y,Y,n,N or EOF
 */
get_ok_char()
{
        int c;
        while( ( c = getchar() ) != EOF && strchr("yYnN",c) == NULL )
                ;
        return c;
}
```

```
set_cr_noecho_mode()
/*
 * purpose: put file descriptor 0 into chr-by-chr mode and noecho mode
 *  method: use bits in termios
 */
{
        struct  termios ttystate;

        tcgetattr( 0, &ttystate);               /* read curr. setting   */
        ttystate.c_lflag        &= ~ICANON;     /* no buffering         */
        ttystate.c_lflag        &= ~ECHO;       /* no echo either       */
        ttystate.c_cc[VMIN]     =  1;           /* get 1 char at a time */
        tcsetattr( 0 , TCSANOW, &ttystate);     /* install settings     */
}


set_nodelay_mode()
/*
 * purpose: put file descriptor 0 into no-delay mode
 *  method: use fcntl to set bits
 *   notes: tcsetattr() will do something similar, but it is complicated
 */
{
        int     termflags;

        termflags = fcntl(0, F_GETFL);          /* read curr. settings  */
        termflags |= O_NDELAY;                  /* flip on nodelay bit  */
        fcntl(0, F_SETFL, termflags);           /* and install 'em      */
}
```

```
/* how == 0 => save current mode,  how == 1 => restore mode */
/* this version handles termios and fcntl flags             */

tty_mode(int how)
{
        static struct termios original_mode;
        static int            original_flags;
        static int            stored = 0;

        if ( how == 0 ){
                tcgetattr(0, &original_mode);
                original_flags = fcntl(0, F_GETFL);
                stored = 1;
        }
        else if ( stored ) {
                tcsetattr(0, TCSANOW, &original_mode);
                fcntl( 0, F_SETFL, original_flags);
        }
}


void ctrl_c_handler(int signum)
/*
 * purpose: called if SIGINT is detected
 *  action: reset tty and scram
 */
{
        tty_mode(1);
        exit(1);
}
```

Programming for Humans : Terminal Control and Signals

- 6.1 Software Tools vs. Device-Specific Programs
- 6.2 Modes of the Terminal Driver
- 6.3 Writing a User Program: play-again.c
- 6.4 Signals
- 6.5 Prepared for Signals: play_again4.c