System Programming
(ELEC462)

Programming for Humans

Dukyun Nam

HPC Lab@KNU

Contents

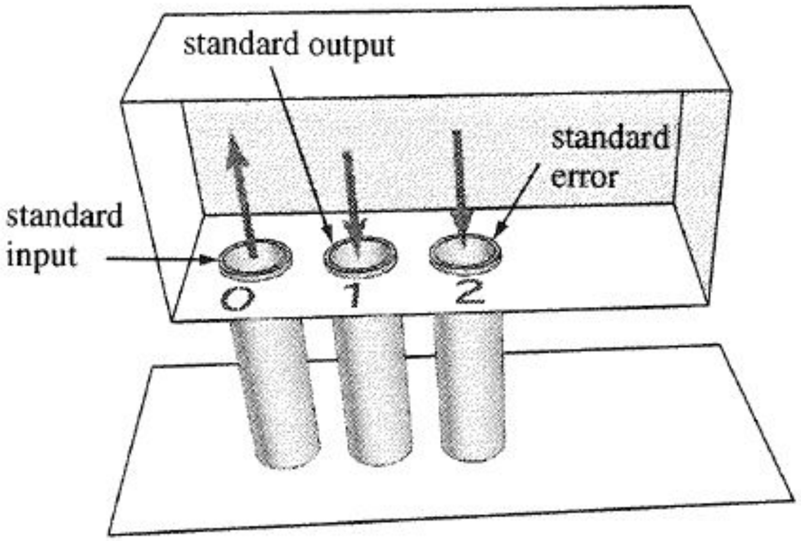
- Introduction
- Software Tools vs Device-Specific Programs
- Modes of the Terminal Driver
- Writing a User Program
- Signals
- Prepared for Signals
- Processes are Mortal
- Programming for Devices
- Summary

Introduction

- 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, that see no difference between disk files and devices
  - e.g., `who, ls, sort, uniq, grep, tr, du`, etc.
  - Reads bytes from *standard input*
  - Does some processing, and then
  - Writes a resulting stream of bytes to *standard output*, or sends error message, streams of bytes, to *standard error*



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

< The three standard file descriptors >

Software Tools (cont.)

- These file descriptors could be connected to files, terminals, mice, printers, and pipes
- Such tools make no assumptions about sources and destinations of data the tools process
- Many of the programs also read from file names on the command line
  - Ex 1) `ls | sort`
  - Ex 2) `ls | uniq`
  - Ex 3) `tr "[:lower:]" "[:upper:]" < file1`

Software Tools (cont.)

- Read stdin or files; write to stdout and stderr
  - `$ sort > file2` # sort input and save to file2

    # [Ctrl-D] = EOF
  - `$ sort x > /dev/lp` # read x and send to printer
  - `$ ls | tr ‘[a-z]’ ‘[A-Z]’` # translate `ls`’s output to uppercase

User-Program: A Common Type of Device-Specific Program

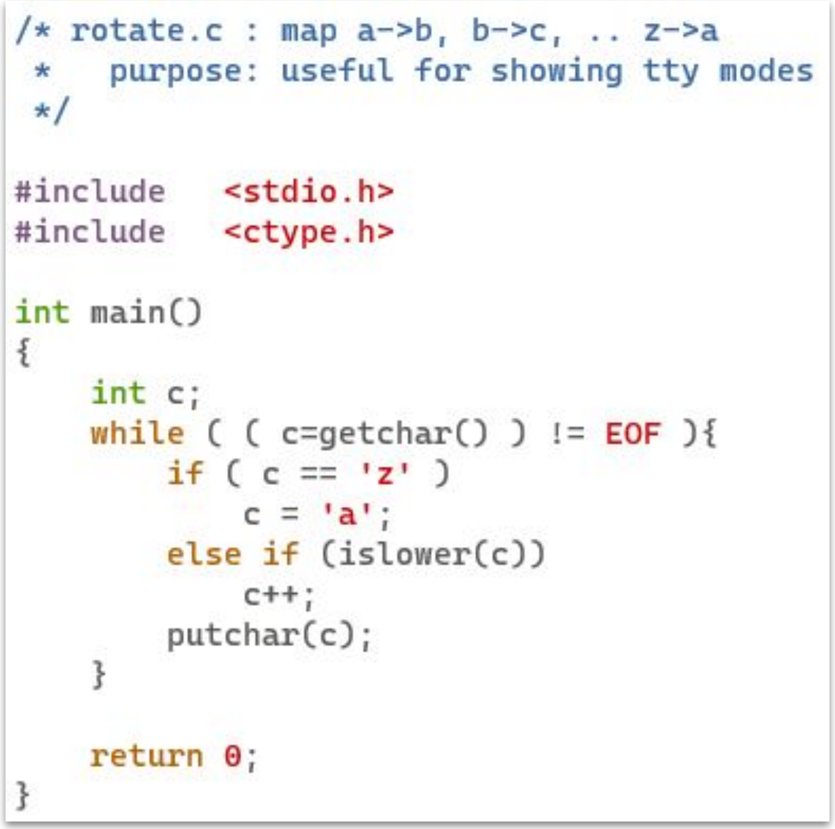
- Device-specific programs are written to interact with specific devices
  - e.g., scanner, camera, cd-rom, printer, terminal, …
- In this chapter, we explore the ideas and techniques of writing a *device-specific programs*, by looking at the most common type of device-specific programs interacting with terminals designed to be used by human beings
  - We refer to the terminal-oriented programs as a *user-program*

User-Program (cont.)

- Examples of user programs:
  - `vi, emacs, more, lynx, hangman, …`
- These programs can adjust setting in the terminal driver to control how the **keystrokes** are handled and output is processed
  - The driver has lots of settings
- Among the settings, common concerns of user program:
  - (a) immediate response to keys
  - (b) limited input set
  - (c) timeout on input
  - (d) resistance to Ctrl-C

Modes of the Terminal Driver

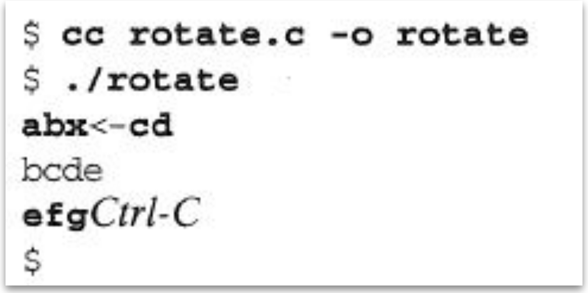
- To explore the terminal driver, let’s experiment with a toy translation program: `rotate.c`



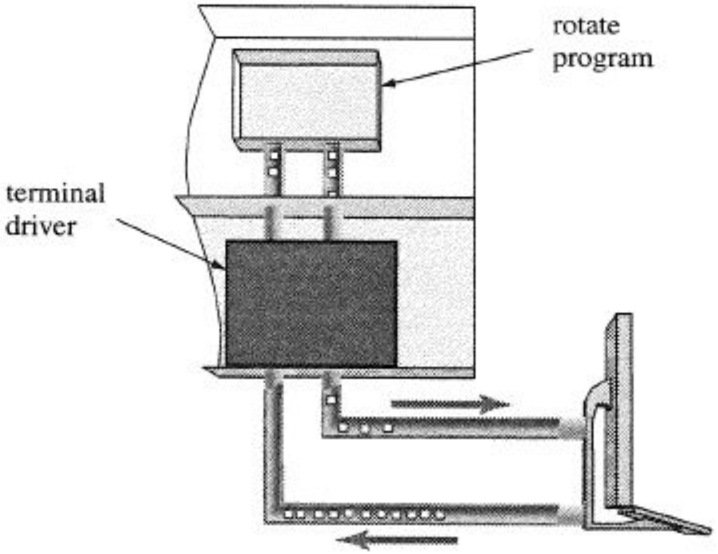
```c
/* rotate.c : map a->b, b->c, .. z->a
 *   purpose: useful for showing tty modes
 */

#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);
    }

    return 0;
}
```

Modes of the Terminal Driver (cont.)

- Canonical (표준) mode: Buffering and Editing
  - Run the program using the default settings:



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

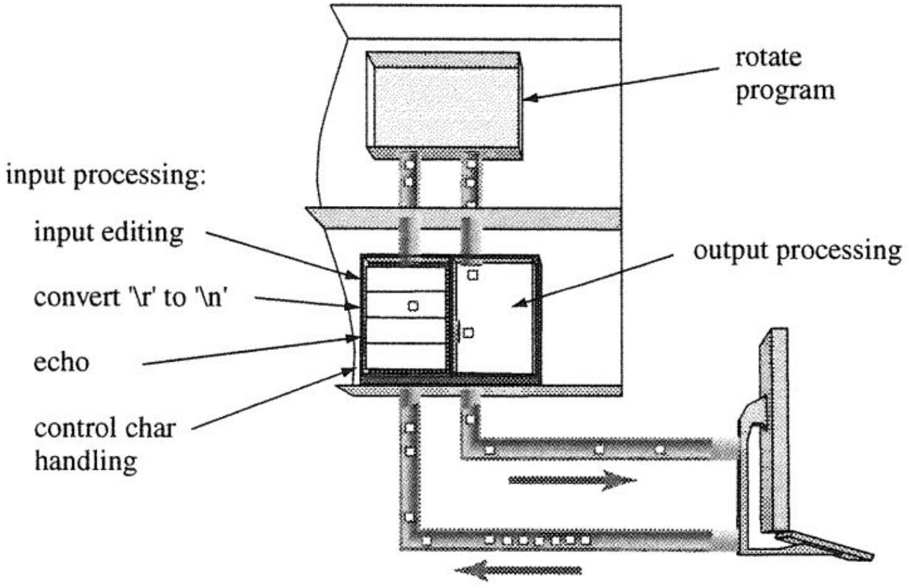


< What you type and what the program gets >

  - Features revealed by this experiment:
    - (a) ‘x’ is never seen by the program; backspace erases it
    - (b) Chars appear on the screen as you type them, but
    - (c) The program does not receive any input until you press the ‘Enter’ key
    - (d) The Ctrl-C key discards input and stops the program

Modes of the Terminal Driver (cont.)

- Canonical mode: Buffering and Editing (Cont’d)
  - Buffering, echoing, editing, and control key processing are all done by the terminal driver
    - When buffering + editing enabled, the terminal connection is said to be in canonical mode

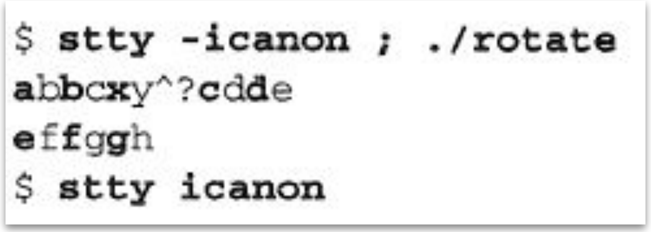


< Processing layers in the terminal driver >

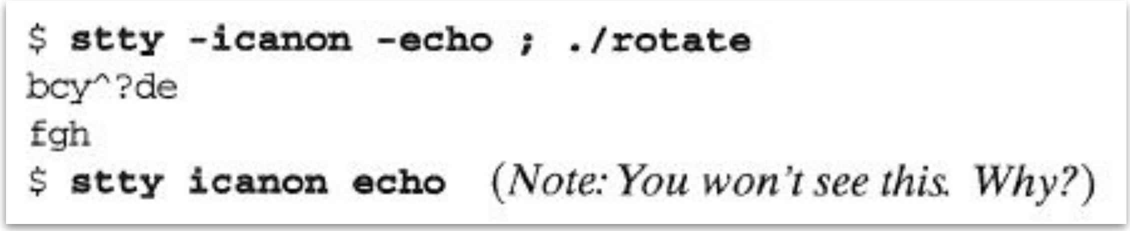
Modes of the Terminal Driver (cont.)

- Noncanonical processing
  - No buffering
    - The command `stty -icanon` turns off *canonical mode* processing in the driver
  - Another experiment:

Input: `abx<-cd`



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



```
$ stty -icanon -echo ; ./rotate
bcy^?de
fgh
$ stty icanon echo   (Note: You won't see this. Why?)
```

- Turn off canonical mode and also turn off echo mode
  - The driver no longer prints back the characters as we type them
  - Output comes only from the program

Summary of Terminal Modes

- Example input

  Hello data DEL DEL DEL DEL world

- RAW mode (all processing off) output

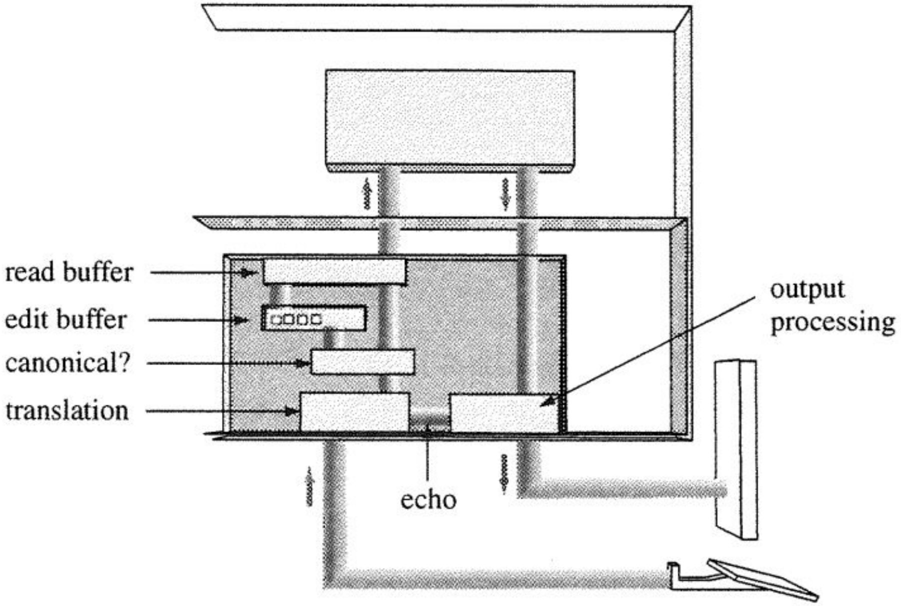
  Hello data DEL DEL DEL DEL world

- Canonical (cooked) mode output

  Hello world

Summary of Terminal Modes (cont.)

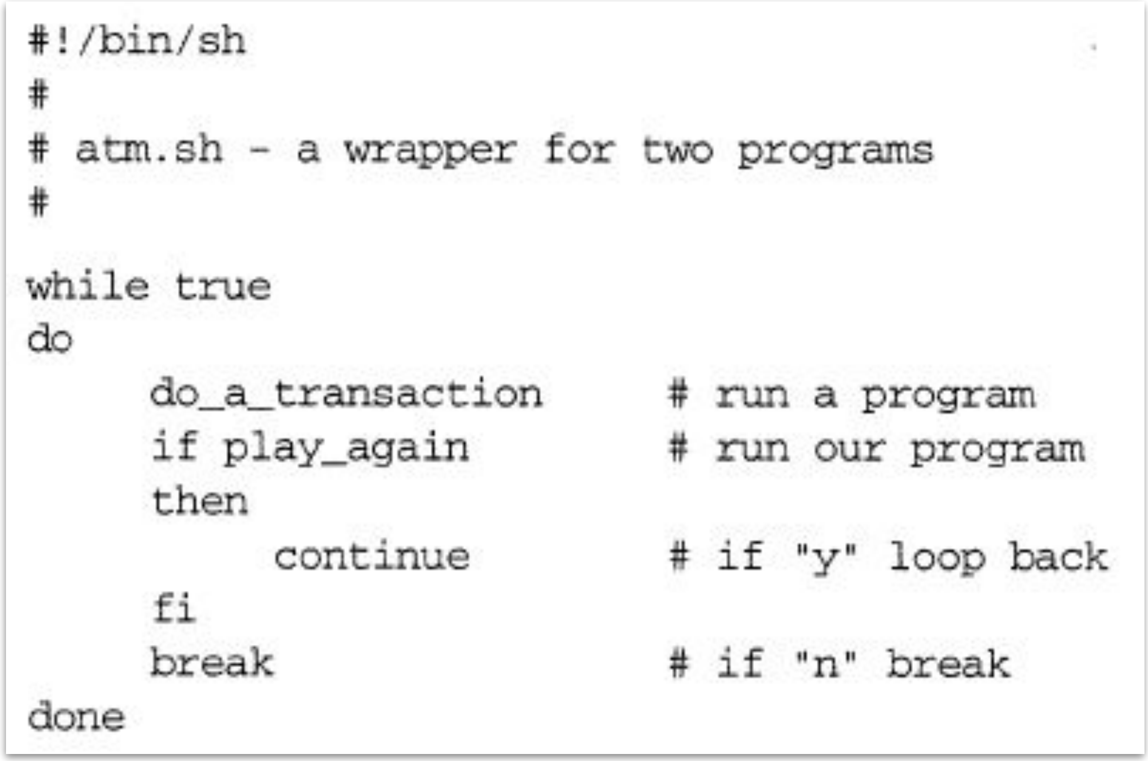
- The terminal driver is a complex set of routines in the kernel
  - To understand the practical value of these modes, we develop a user program that uses various driver modes



< Major components of the terminal driver >

Writing a User Program: `play_again.c`

- Many user applications ask users *yes/no* questions
- The following shell script is the main loop for a bank machine:



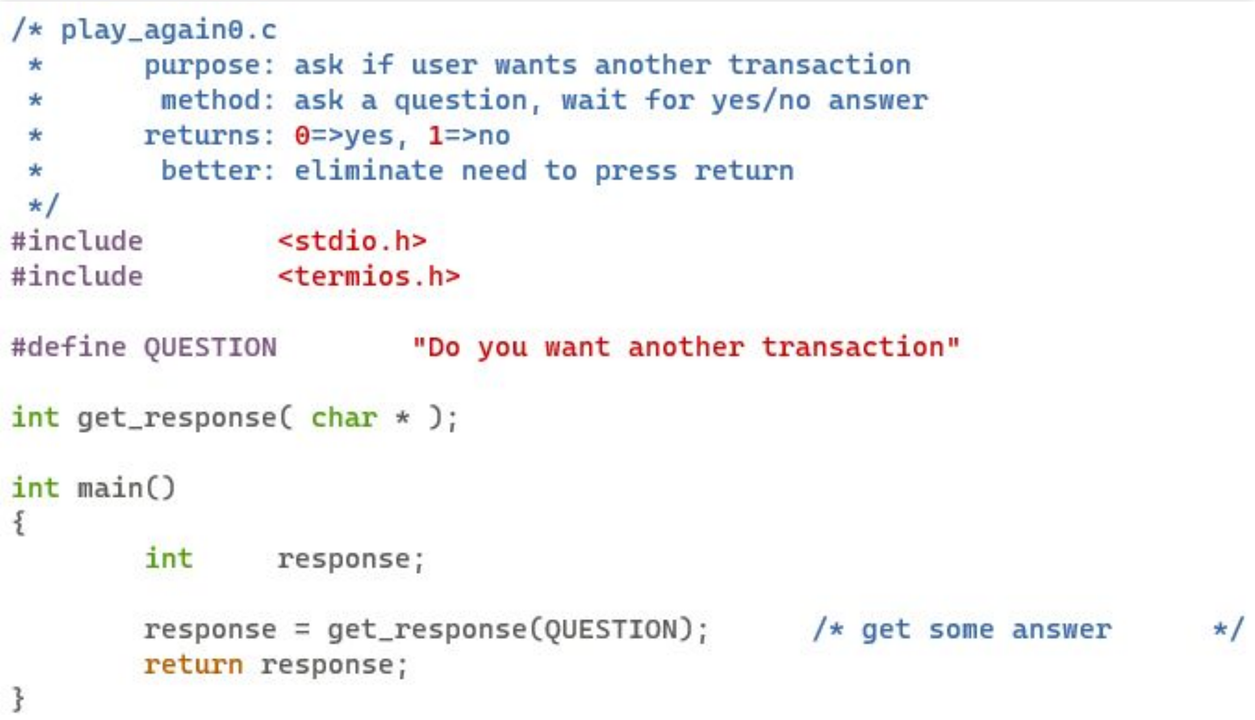
```
#!/bin/sh
#
# atm.sh - a wrapper for two programs
#

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
```

Writing a User Program: `play_again.c` (cont.)

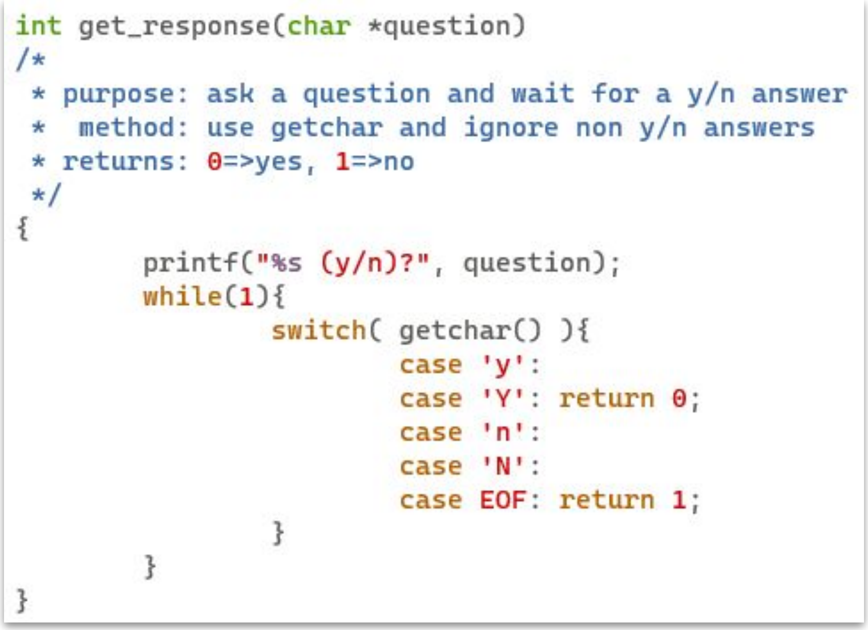
- What does `play_again` do?
  - The logic of `play_again.c`:
    - Prompt a user with a question
    - Accept input
    - If “y”, return 0
    - If “n”, return 1

Writing a User Program: `play_again0`

- How: write a function with a loop
- But, a user needs to press RETURN key
    - ATMs don’t require that



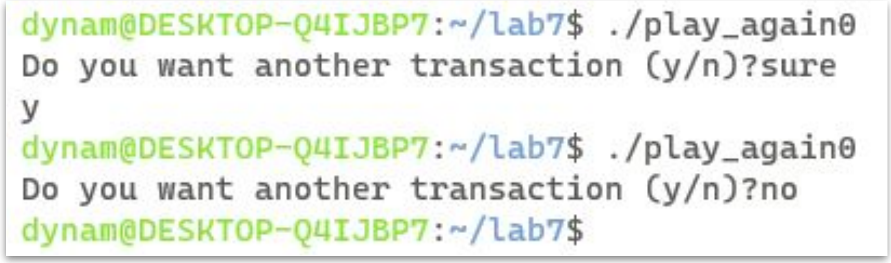
```
/* play_again0.c
 *      purpose: ask if user wants another transaction
 *       method: ask a question, wait for yes/no answer
 *      returns: 0=>yes, 1=>no
 *       better: eliminate need to press return
 */
#include        <stdio.h>
#include        <termios.h>

#define QUESTION        "Do you want another transaction"

int get_response( char * );

int main()
{
        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;
                }
        }
}
```

Writing a User Program: `play_again0` (cont.)

- Output



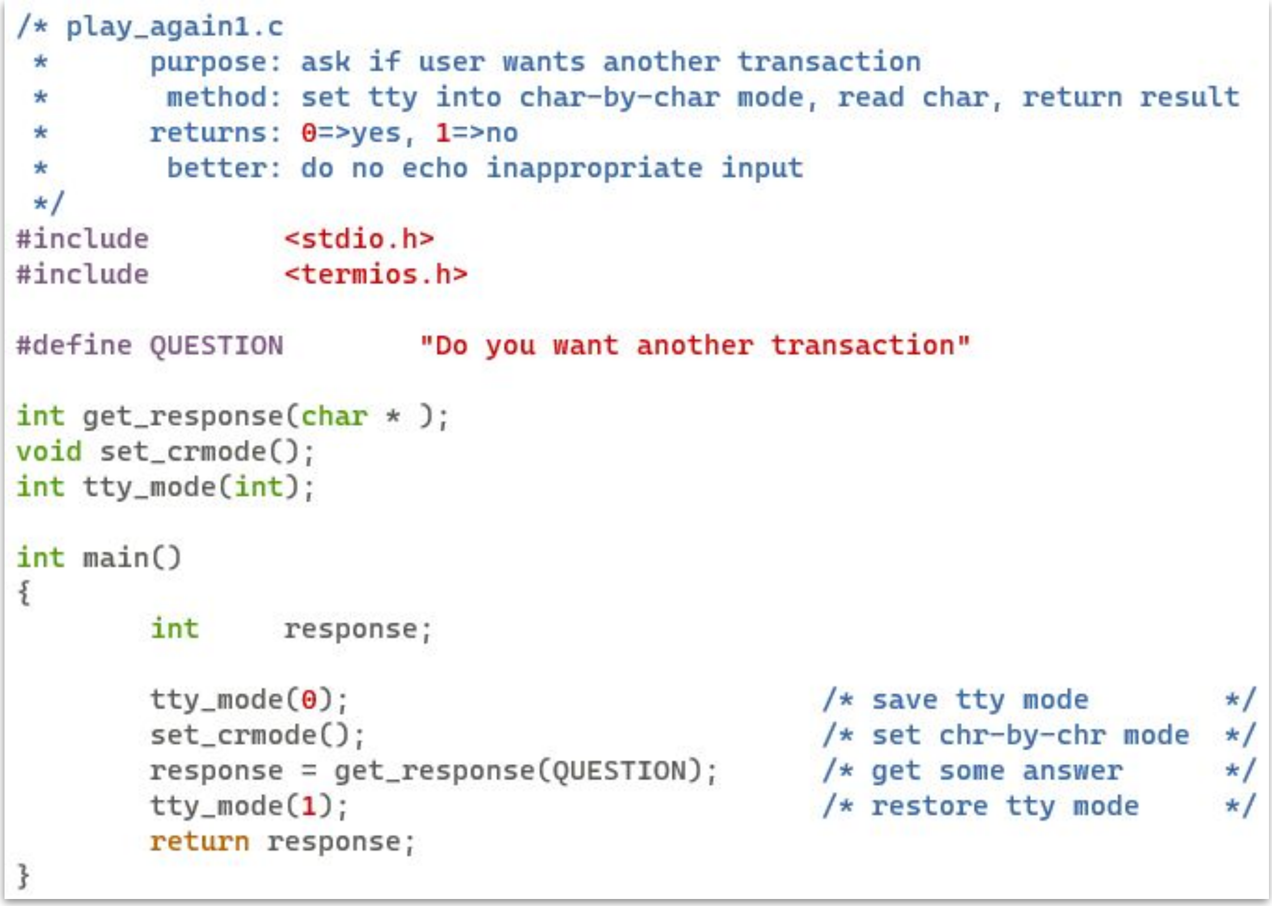
```
dynam@DESKTOP-Q4IJBP7:~/lab7$ ./play_again0
Do you want another transaction (y/n)?sure
y
dynam@DESKTOP-Q4IJBP7:~/lab7$ ./play_again0
Do you want another transaction (y/n)?no
dynam@DESKTOP-Q4IJBP7:~/lab7$
```

- Two problems
  - 1) The user has to press ‘Enter’ before the program can act on input
  - 2) When the user presses ‘Enter’, the program receives and processes an entire line of data
- So, let’s turn off canonical input to have the program act based on characters

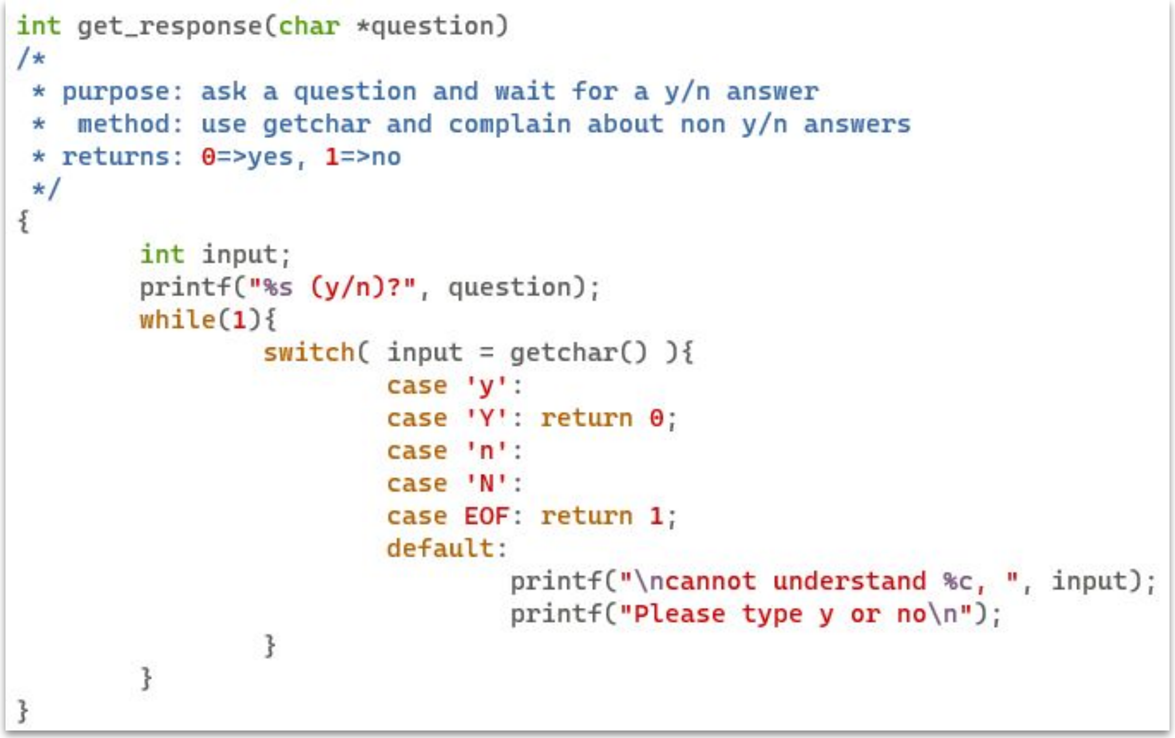
Writing a User Program: `play_again1`

- Immediate response
  - Idea: process each char as typed
  - How: use `tcsetattr()` to
    - Turn off editing (`&= ~ICANON`)
    - Set input size to 1 char
  - Note: need to reset tty at the end of program
    - Choice: do we set ICANON on or just restore old settings?
  - But, responds to each char, usually with an error

Writing a User Program: `play_again1` (cont.)

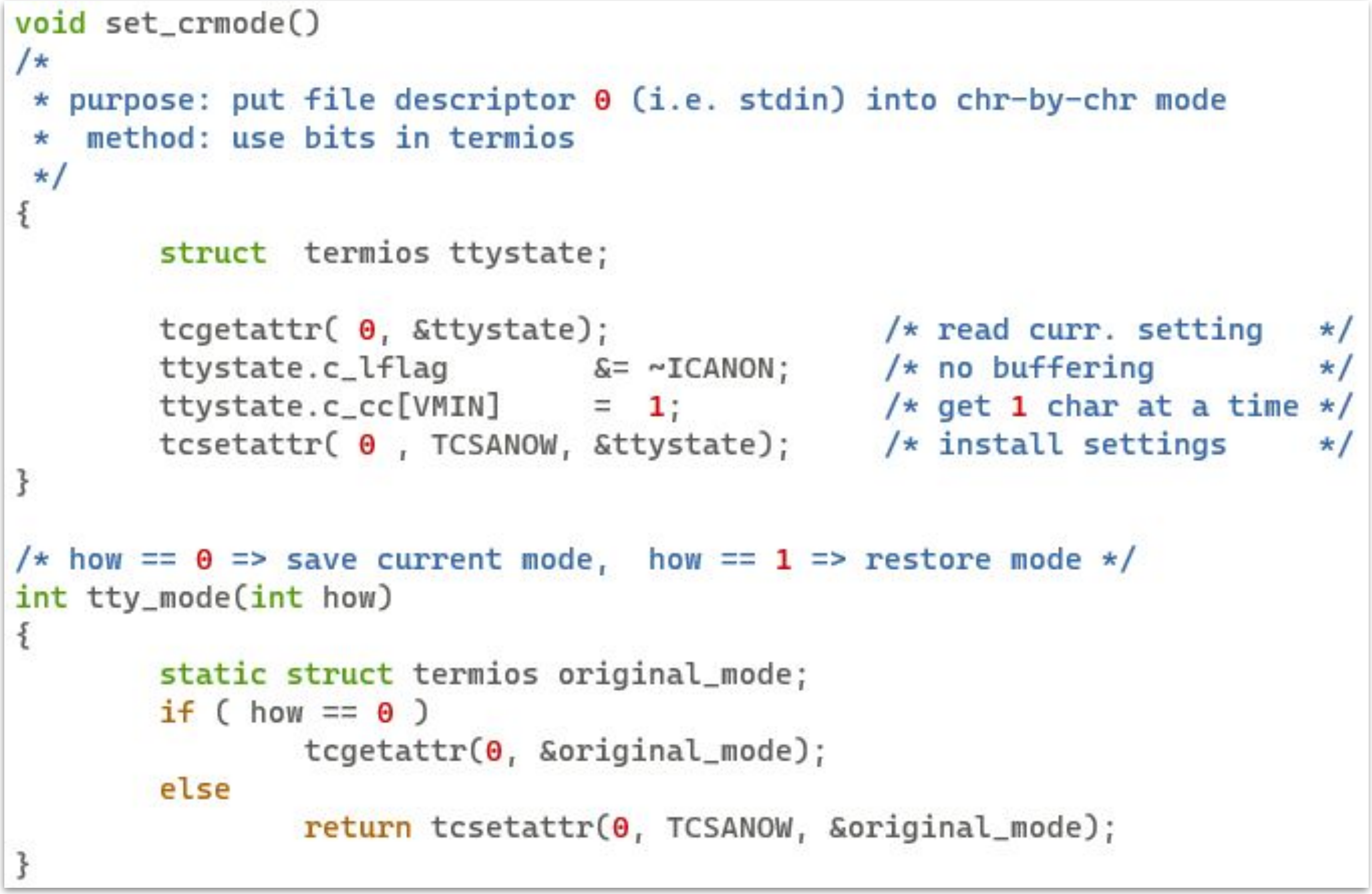


```
/* play_again1.c
 *      purpose: ask if user wants another transaction
 *       method: set tty into char-by-char mode, read char, return result
 *      returns: 0=>yes, 1=>no
 *       better: do no echo inappropriate input
 */
#include        <stdio.h>
#include        <termios.h>

#define QUESTION        "Do you want another transaction"

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

int main()
{
        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;
}
```



```
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");
                }
        }
}
```

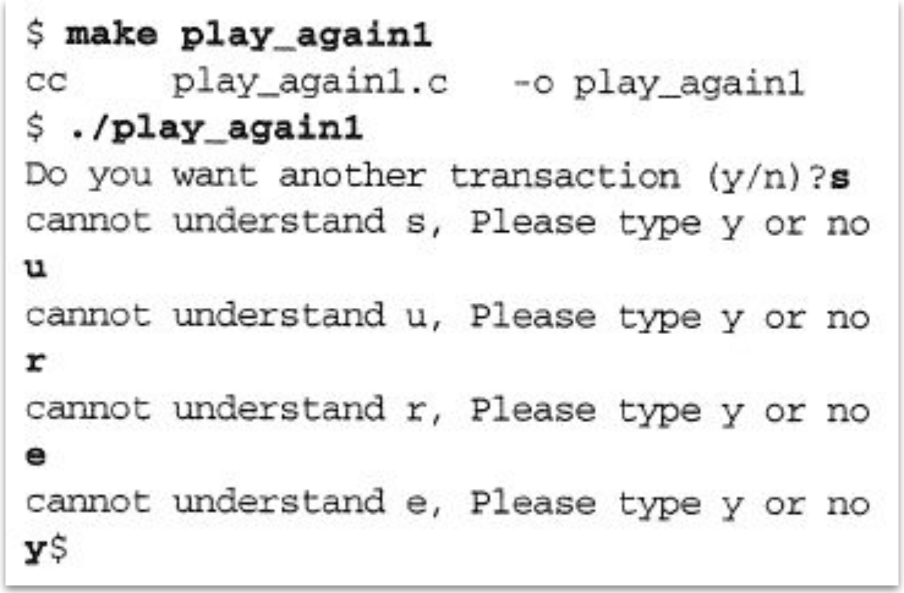
Writing a User Program: `play_again1` (cont.)



```c
void set_crmode()
/*
 * 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 */
int tty_mode(int how)
{
        static struct termios original_mode;
        if ( how == 0 )
                tcgetattr(0, &original_mode);
        else
                return tcsetattr(0, TCSANOW, &original_mode);
}
```

Writing a User Program: `play_again1` (cont.)

- Output
    - Type *sure* as a response:

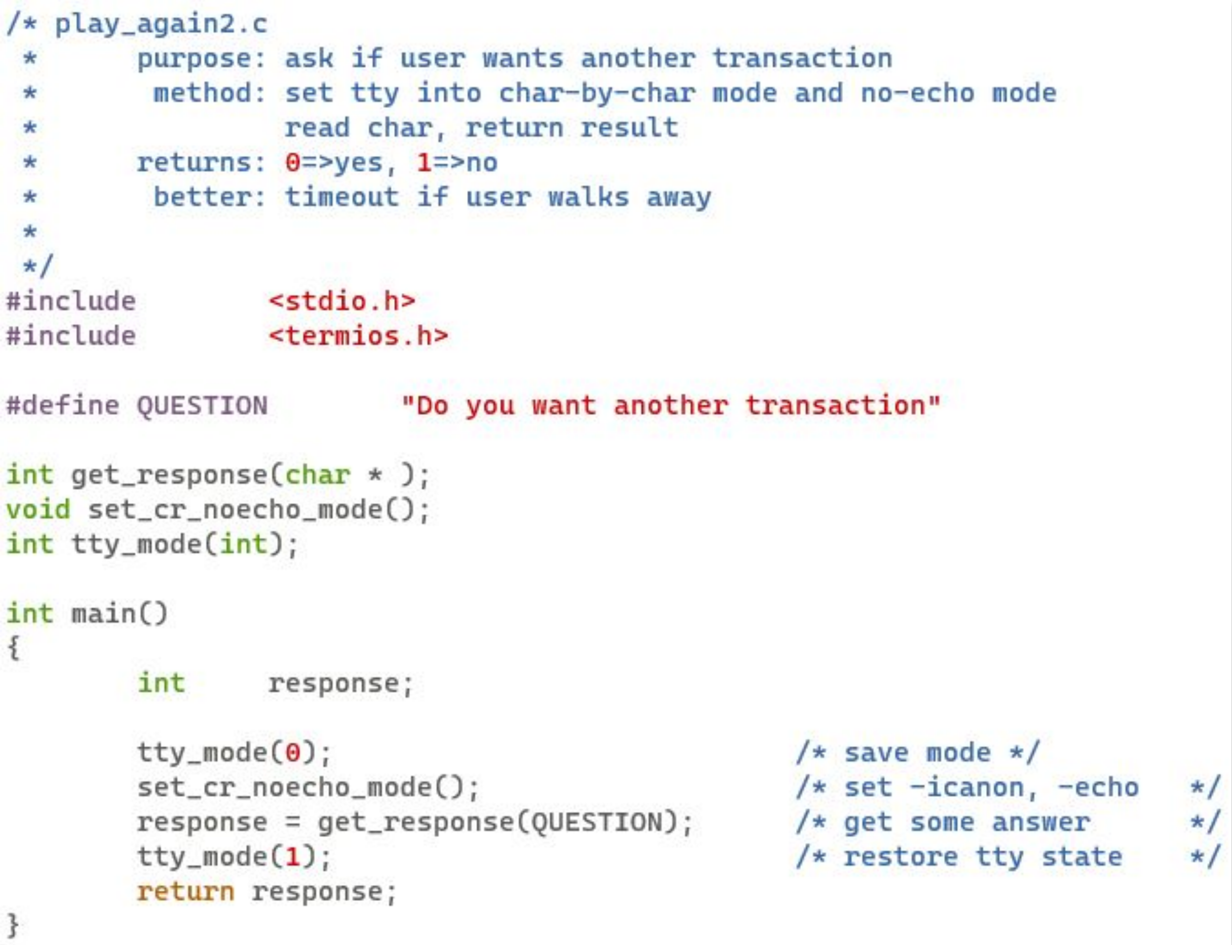


```
$ make play_again1
cc     play_again1.c   -o play_again1
$ ./play_again1
Do you want another transaction (y/n)?s
cannot understand s, Please type y or no
u
cannot understand u, Please type y or no
r
cannot understand r, Please type y or no
e
cannot understand e, Please type y or no
y$
```

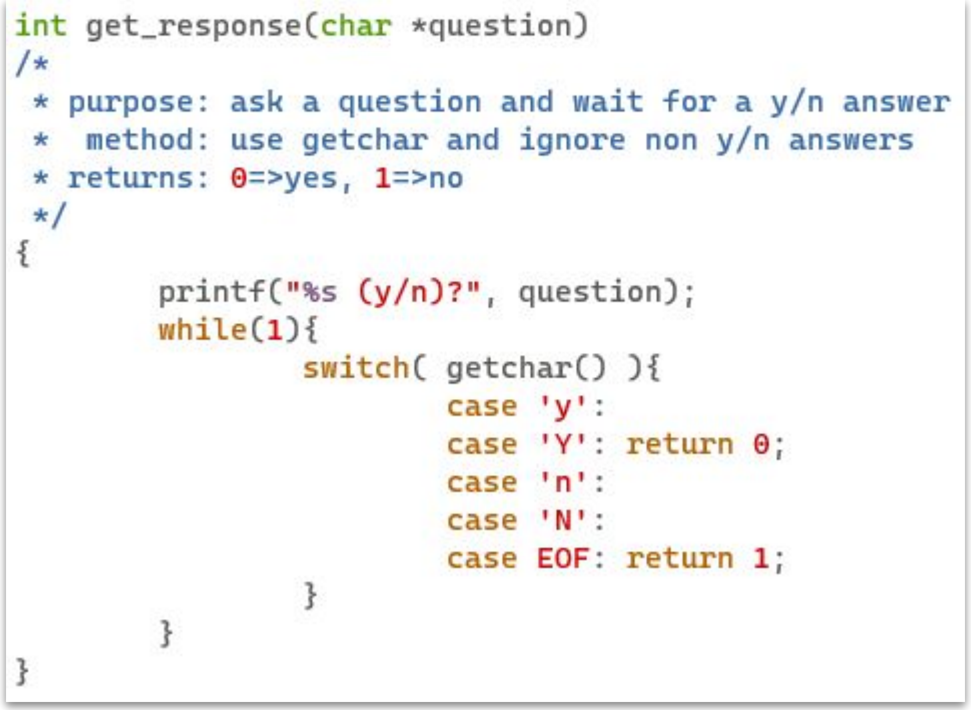
Writing a User Program: `play_again2`

- Ignore illegal chars
  - Idea: turn off echo
    - Simply ignore non y/n input
  - How: `&= ~ECHO`
    - No error messages
    - Program echos on legal input
  - Note: need to use `putchar()` to echo
  - But, what if a user wanders away without entering any key?

Writing a User Program: `play_again2` (cont.)

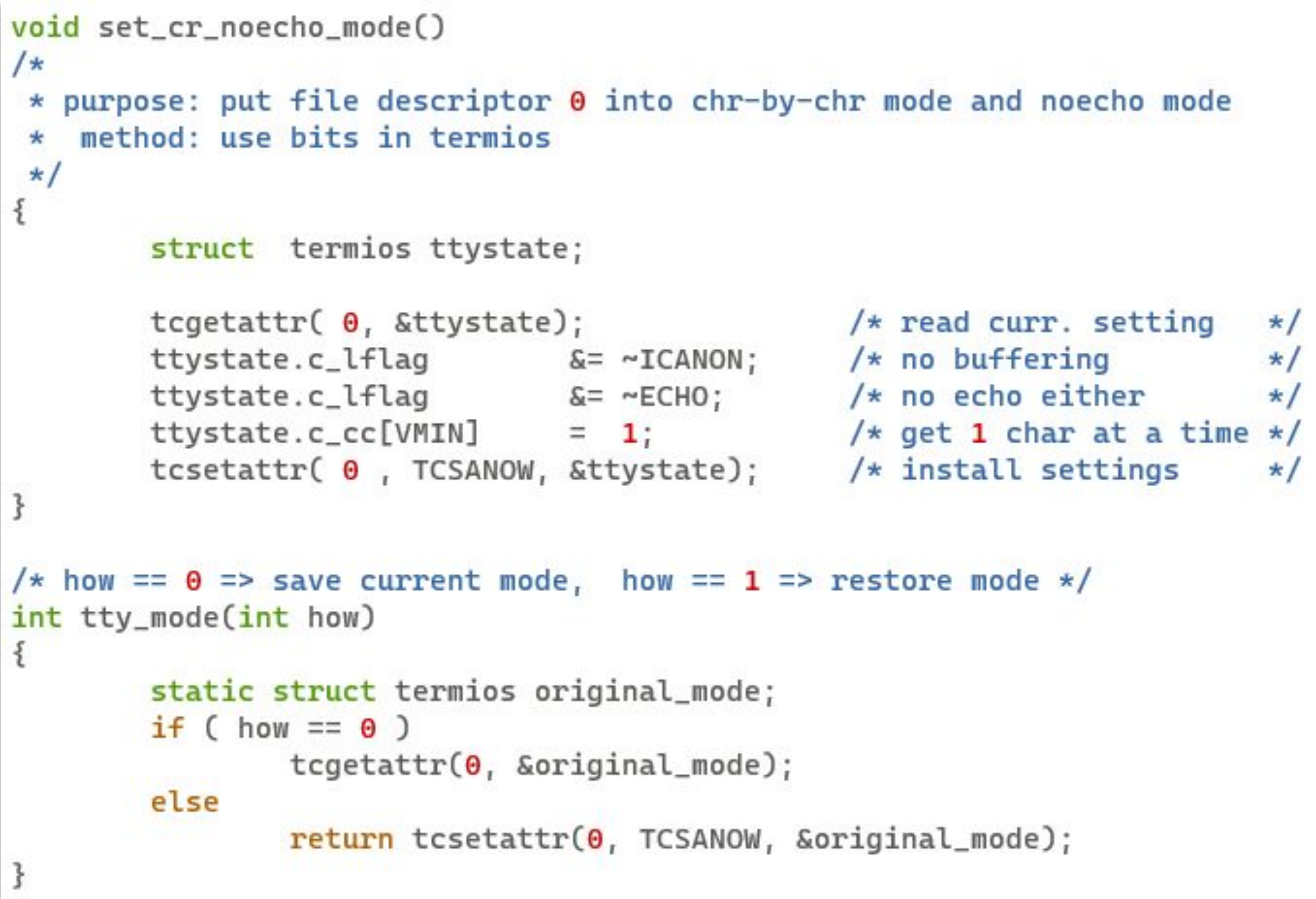


```c
/* play_again2.c
 *      purpose: ask if user wants another transaction
 *       method: set tty into char-by-char mode and no-echo mode
 *               read char, return result
 *      returns: 0=>yes, 1=>no
 *       better: timeout if user walks away
 *
 */
#include        <stdio.h>
#include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

#define QUESTION        "Do you want another transaction"

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

int main()
{
        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;
                }
        }
}
```

Writing a User Program: `play_again2` (cont.)



```c
void 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 */
int tty_mode(int how)
{
        static struct termios original_mode;
        if ( how == 0 )
                tcgetattr(0, &original_mode);
        else
                return tcsetattr(0, TCSANOW, &original_mode);
}
```

Writing a User Program: `play_again3`

- Non-blocking mode
  - Blocking mode:
    - e.g., `getchar()` or `read()` → Wait for input
  - How to set non-blocking
    - Use `O_NDELAY` in `open()` or `fcntl()`
  - New `play_again`
    - Timeout feature
    - Telling the terminal driver NOT to wait
    - No input found then sleep for few (2) seconds and look again for input
    - After three attempts, then give up

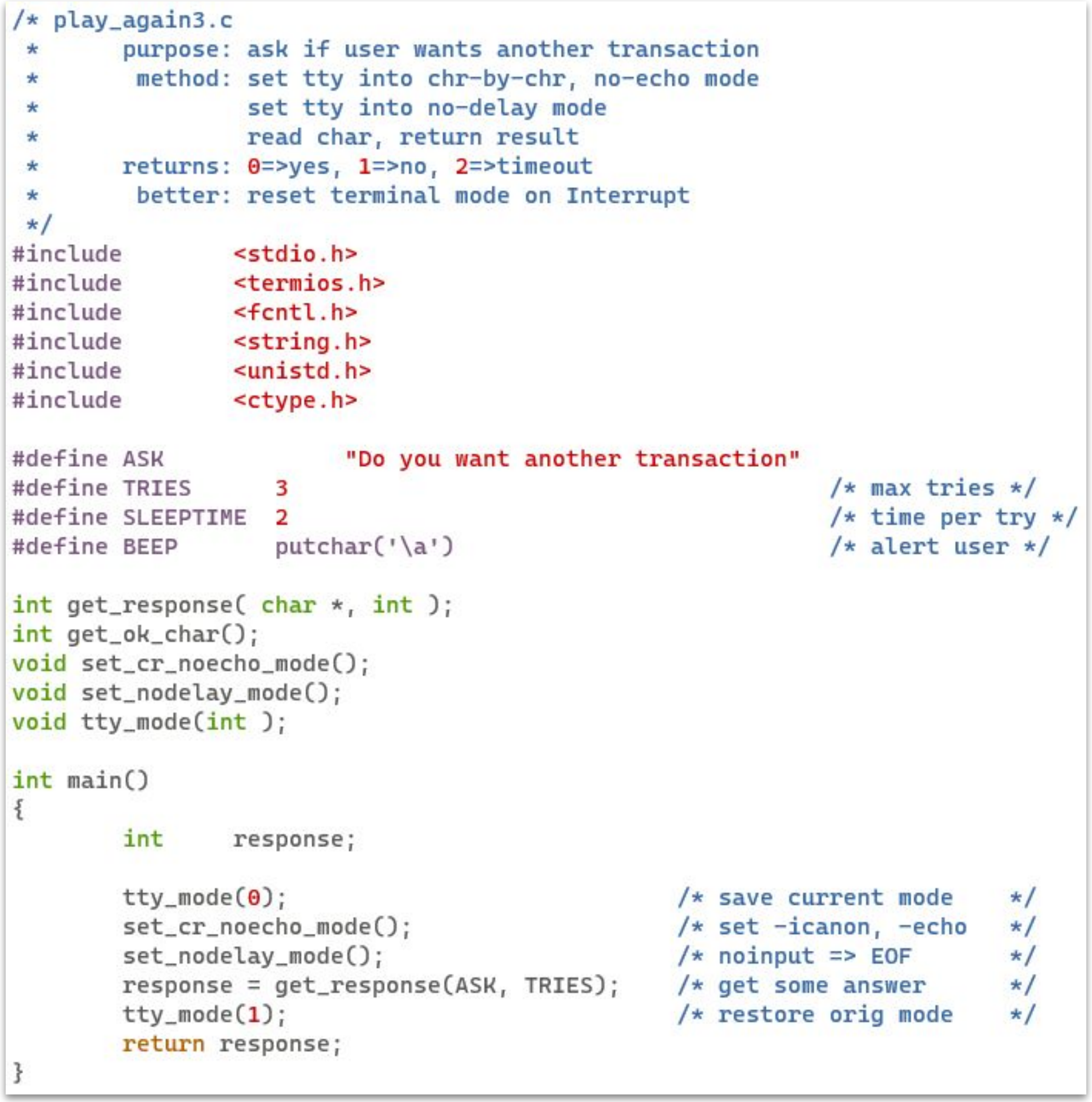
Writing a User Program: `play_again3` (cont.)

- Idea: if no input
  - wait.. then ask again

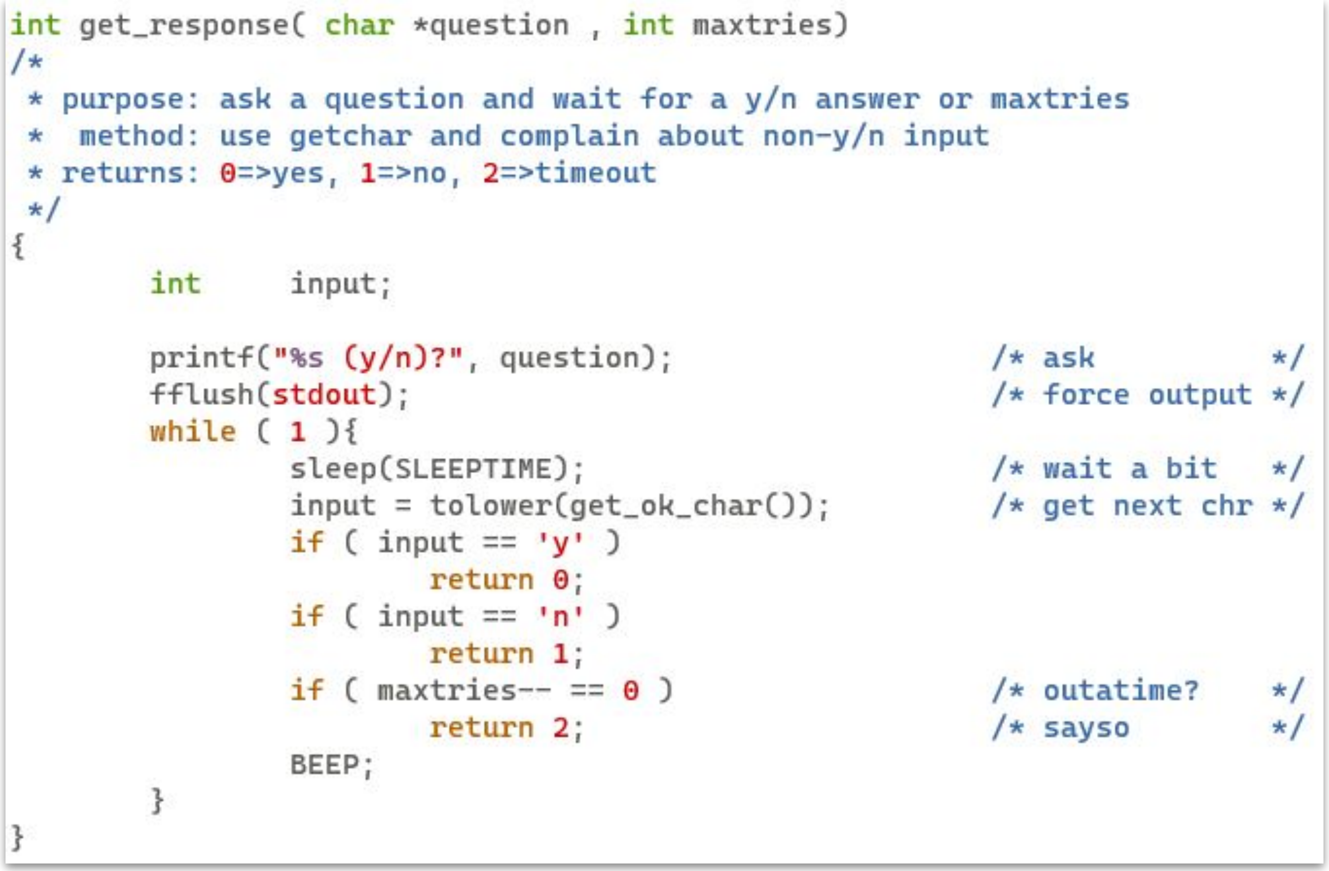
    …

    give up
- How: put the fd in ‘non-blocking’ mode
  - i.e., do not wait (block) for input
  - read return 0 if no chars available
- Note: read() usually waits for input
- Fact
  - Non-blocking input is an attribute of a file descriptor and may be set for any open file - disk files AND devices

Writing a User Program: `play_again3` (cont.)

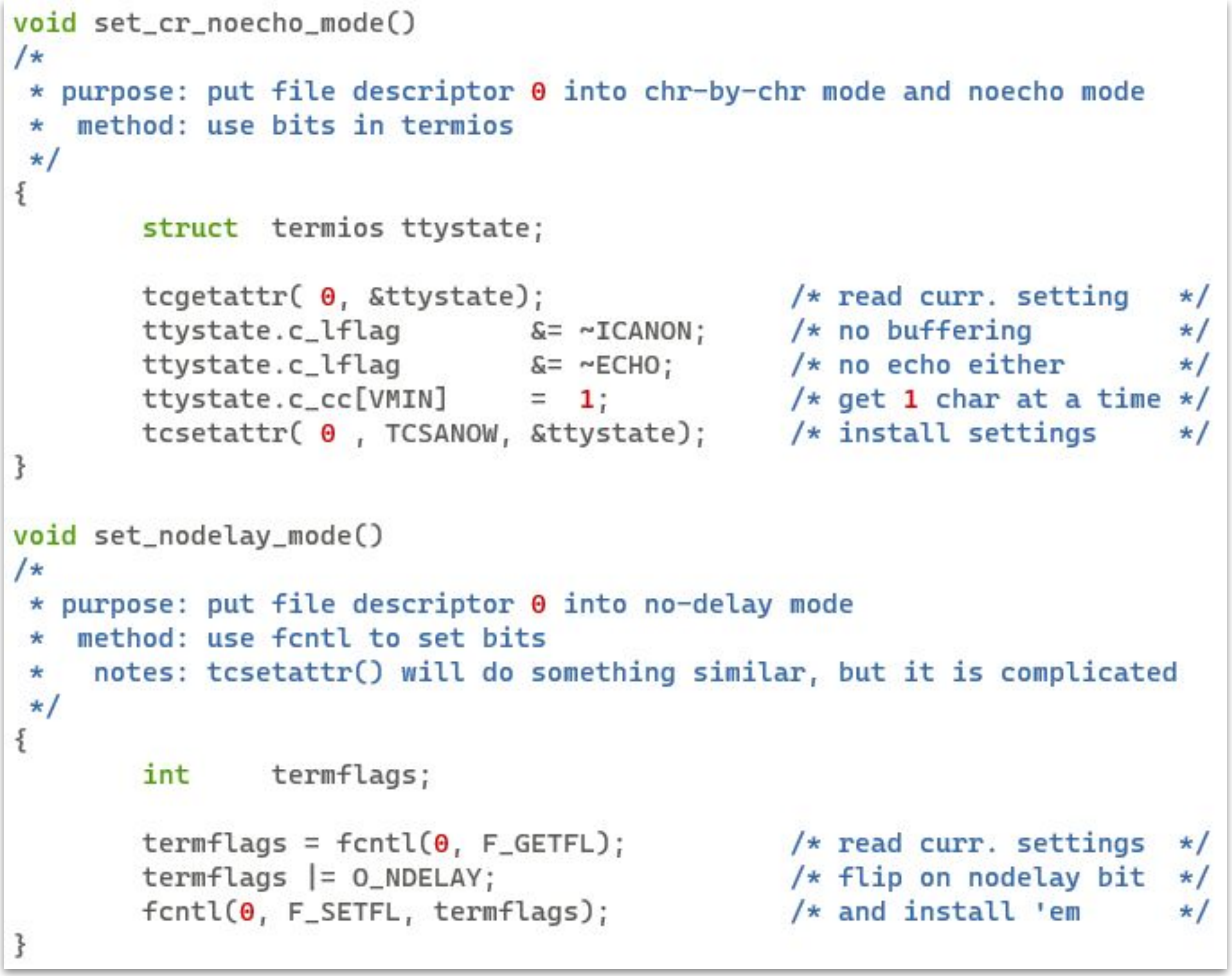


```
/* play_again3.c
 *      purpose: ask if user wants another transaction
 *       method: set tty into chr-by-chr, no-echo mode
 *               set tty into no-delay mode
 *               read char, return result
 *      returns: 0=>yes, 1=>no, 2=>timeout
 *       better: reset terminal mode on Interrupt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

int get_response( char *, int );
int get_ok_char();
void set_cr_noecho_mode();
void set_nodelay_mode();
void tty_mode(int );

int 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;
}
```

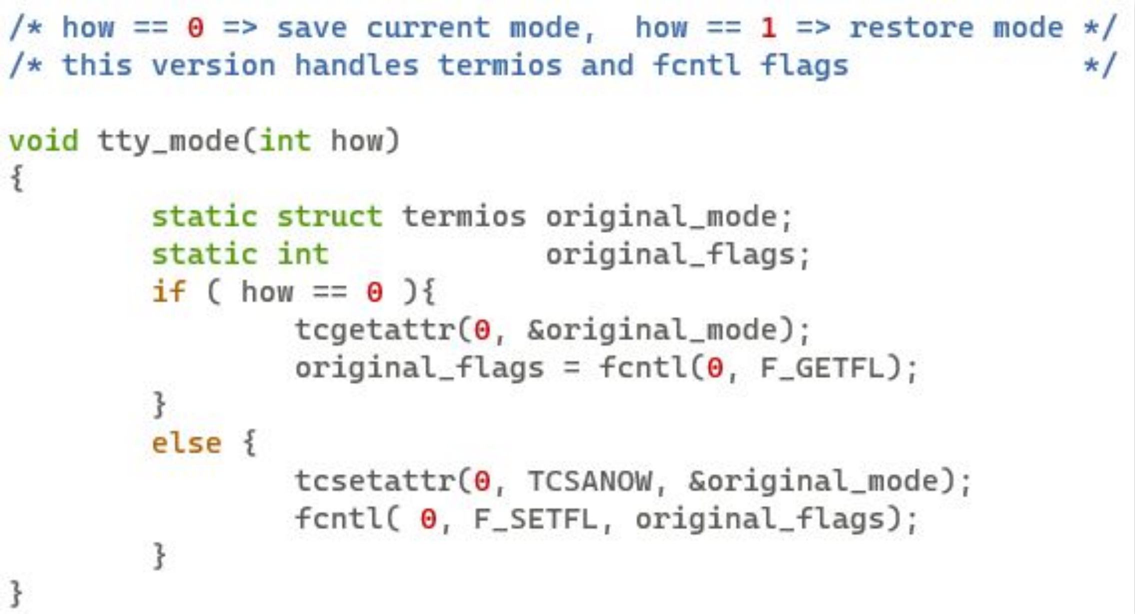


```
int get_response( char *question , int maxtries)
/*
 * purpose: ask a question and wait for a y/n answer or maxtries
 *  method: use getchar and complain about non-y/n input
 * returns: 0=>yes, 1=>no, 2=>timeout
 */
{
        int     input;

        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;
        }
}
```

Writing a User Program: `play_again3` (cont.)



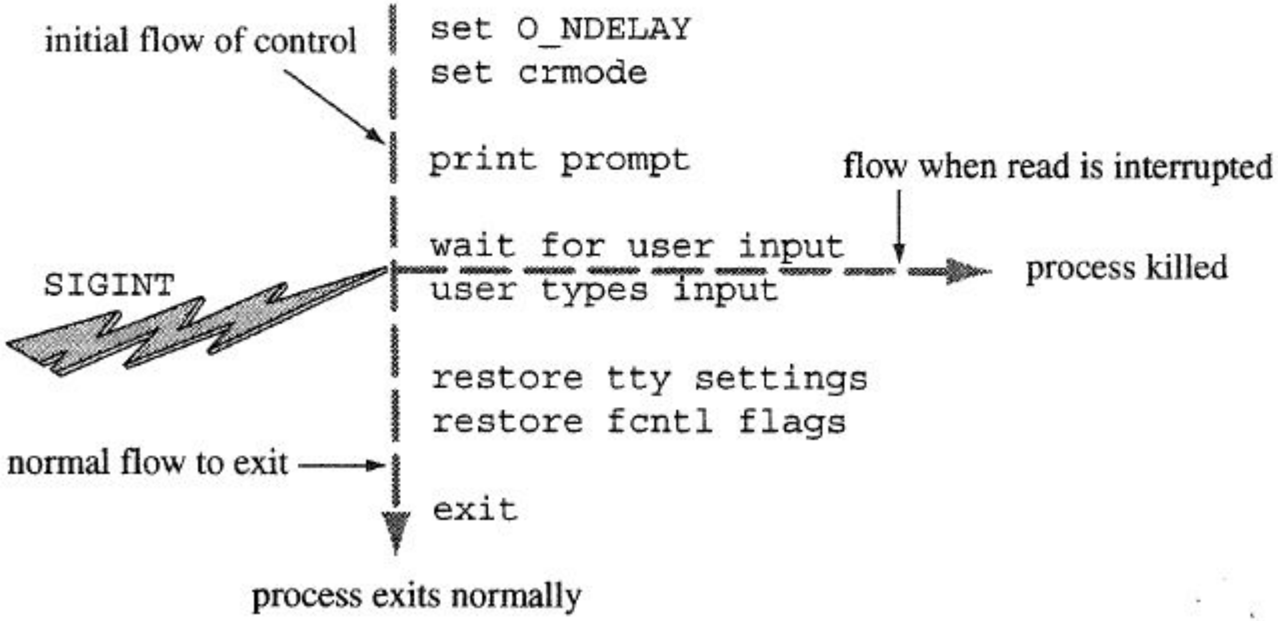
```c
void set_cr_noecho_mode()
/*
 * purpose: put file descriptor 0 into chr-by-chr mode and noecho mode
 *  method: use bits in termios
 */
{
        struct  termios ttystate;

        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     */
}

void 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      */
}
```



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

void 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);
        }
}
```

Signals

- Ctrl-C: Kills a running process



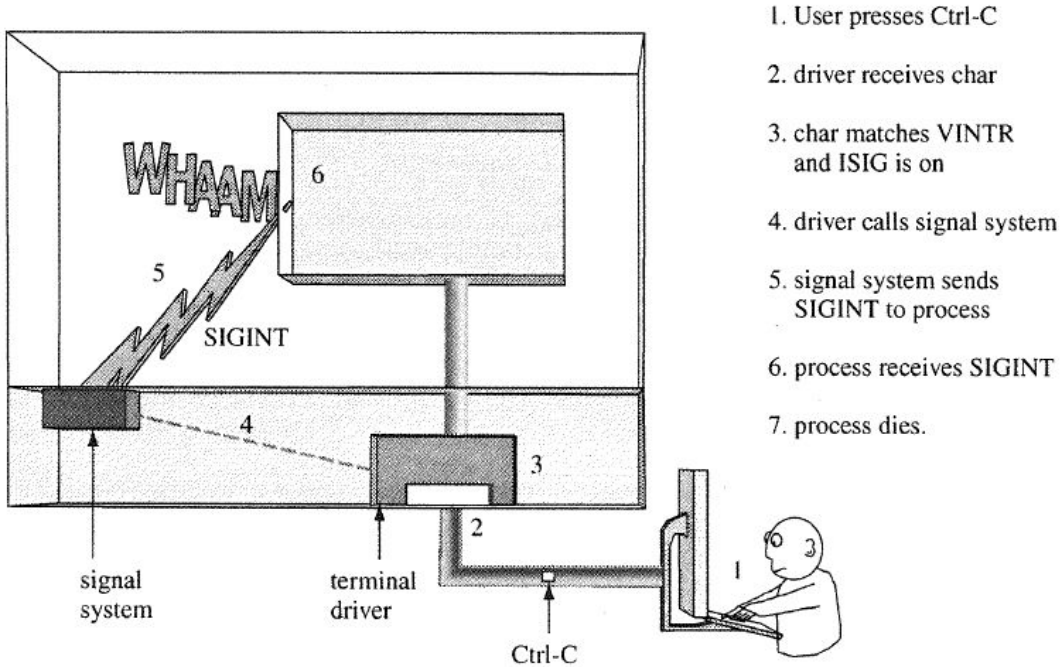
< Ctrl-C kills a program >

Signals (cont.)

- What does Ctrl-C do?
  - The Ctrl-C key *interrupts* a running program.
    - Generates a signal!
- Signal: defined as a one-word message
  - A kernel mechanism of showing how the kernel “controls” processes
  - Simple (by **numbers**) but very powerful and strong
    - e.g., go, stop, out, green light
- Each signal has its own numerical code.
  - Interrupt signal generated by Ctrl-C is No. 2: SIGINT

Signals (cont.)

- How does Ctrl-C do?
  - ‘VINTR’
    - Interrupt character (INTR)
    - Send a SIGINT signal
  - ‘ISIG’
    - When any of the characters INTR, QUIT, SUSP, or DSUSP are received, generate the corresponding signal
  - Refer to `termios`



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.

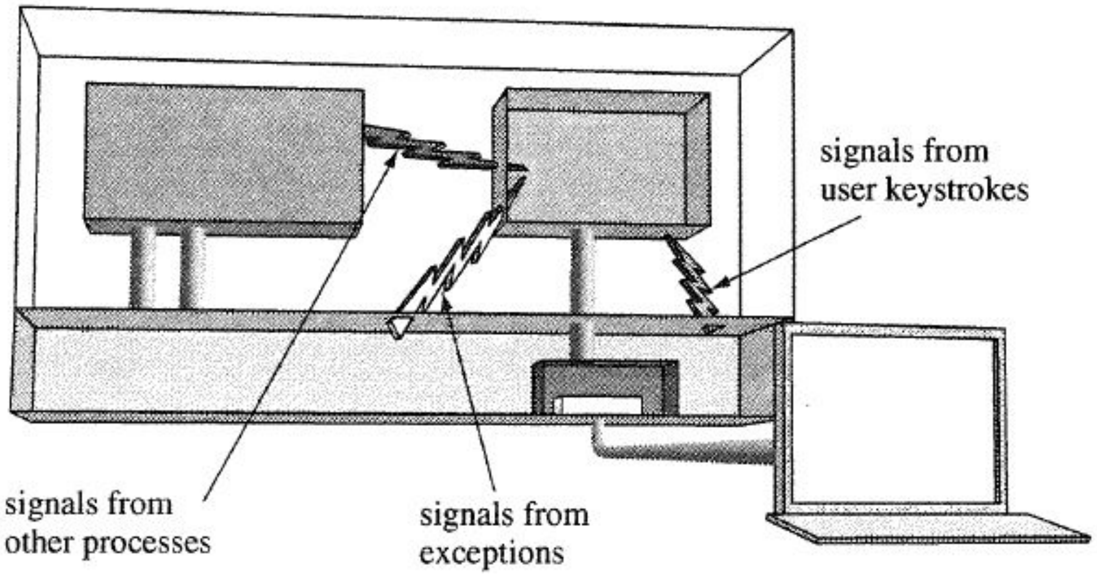
< How Ctrl-C works >

Signals (cont.)

- Where do signals come from?
  - Users
    - Press Ctrl-C, Ctrl-\, …
  - Kernel
    - A segmentation violation,

      a floating point exception,

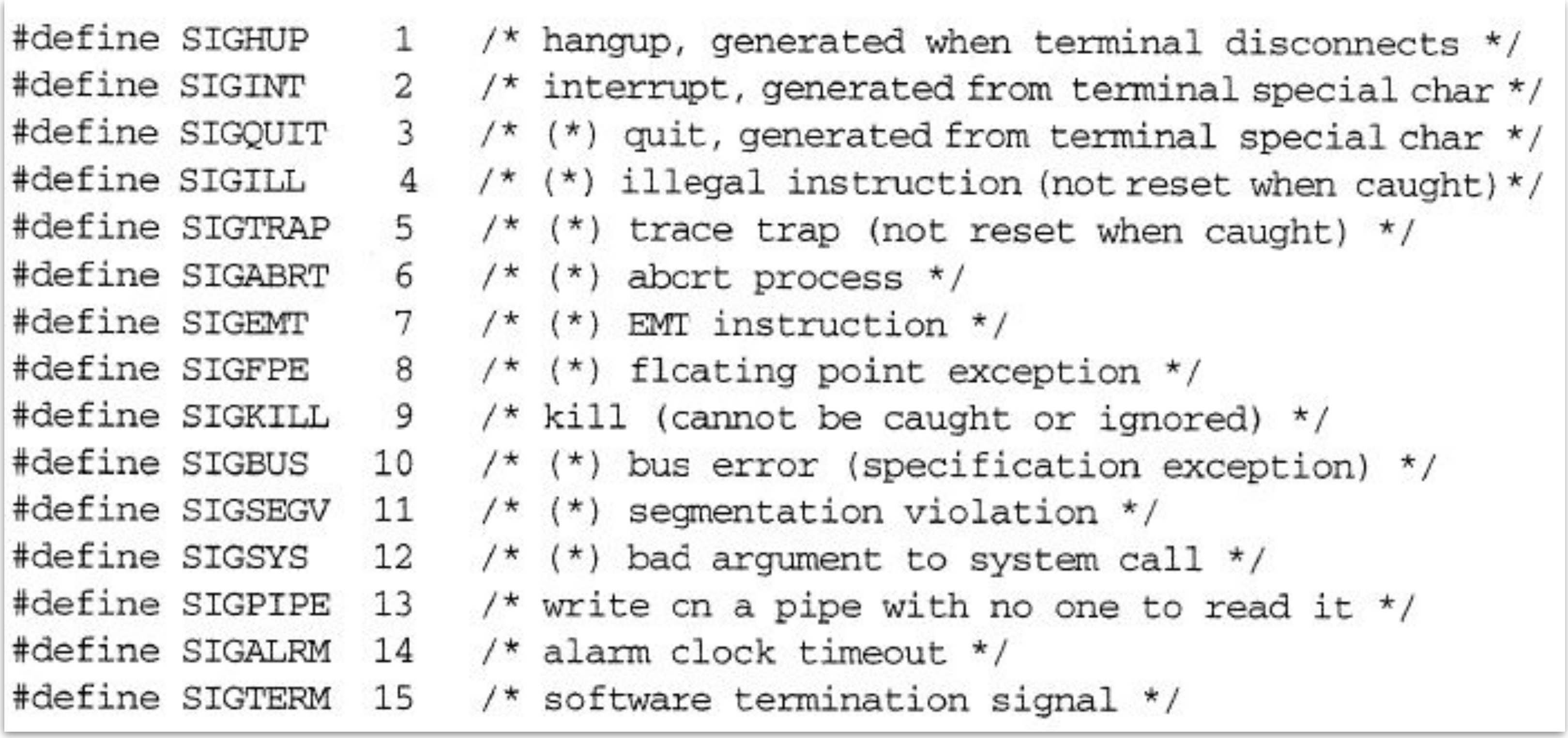
      or an illegal opcode
  - Other processes
    - One process can send a signal to another process
      - e.g., `kill` system call



< Three sources of signals >

Signals (cont.)

- List of signals
  - “Typically” defined in `/usr/include/signal.h`



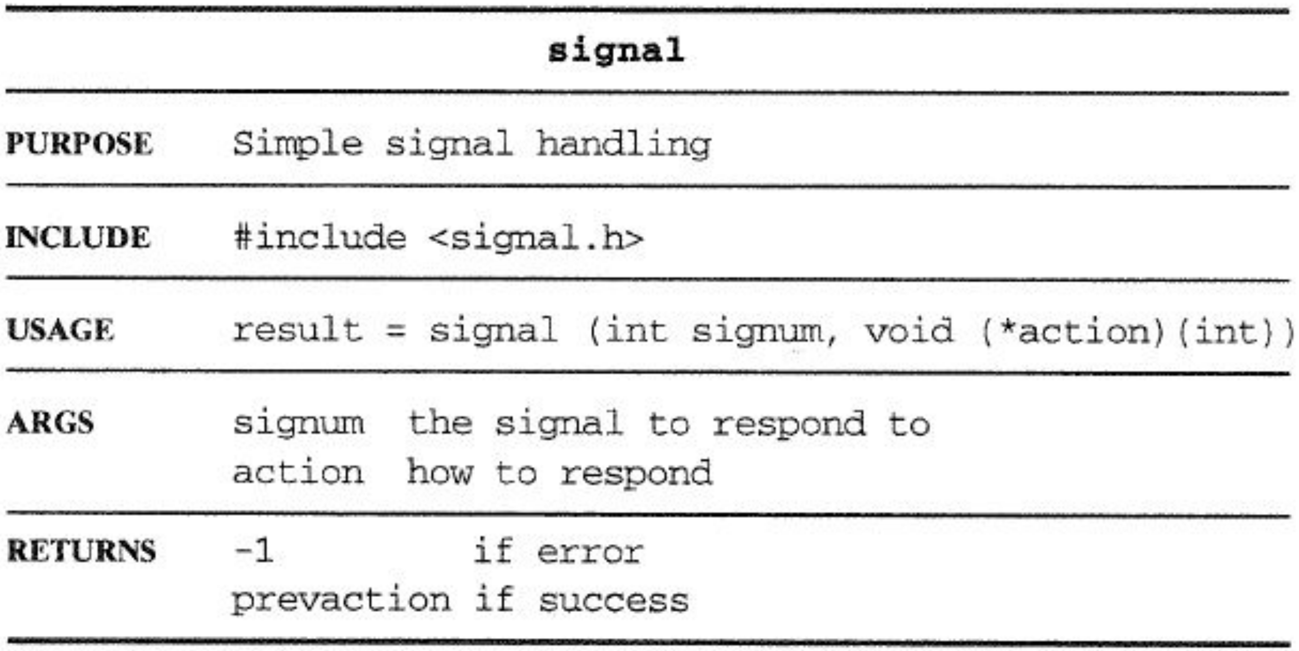
```
#define SIGHUP    1   /* hangup, generated when terminal disconnects */
#define SIGINT    2   /* interrupt, generated from terminal special char */
#define SIGQUIT   3   /* (*) quit, generated from terminal special char */
#define SIGILL    4   /* (*) illegal instruction (not reset when caught)*/
#define SIGTRAP   5   /* (*) trace trap (not reset when caught) */
#define SIGABRT   6   /* (*) abcrt process */
#define SIGEMT    7   /* (*) EMT instruction */
#define SIGFPE    8   /* (*) flcating 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 cn a pipe with no one to read it */
#define SIGALRM  14   /* alarm clock timeout */
#define SIGTERM  15   /* software termination signal */
```

Signals (cont.)

- When a signal comes
  - A process can tell the kernel, by using the `signal` system call, how it wants to respond to a signal
  - Process have 3 choices:
    - Accept the default action (usually death)
      - `signal(SIGINT, SIG_DFL) // reset signal to its default action`
    - Ignore
      - “Hey kernel, I want to ignore SIGINT”
      - `signal(SIGINT, SIG_IGN)// ignore the signal`
    - Call a function (say, f), called signal handler
      - `signal(SIGINT, f)`

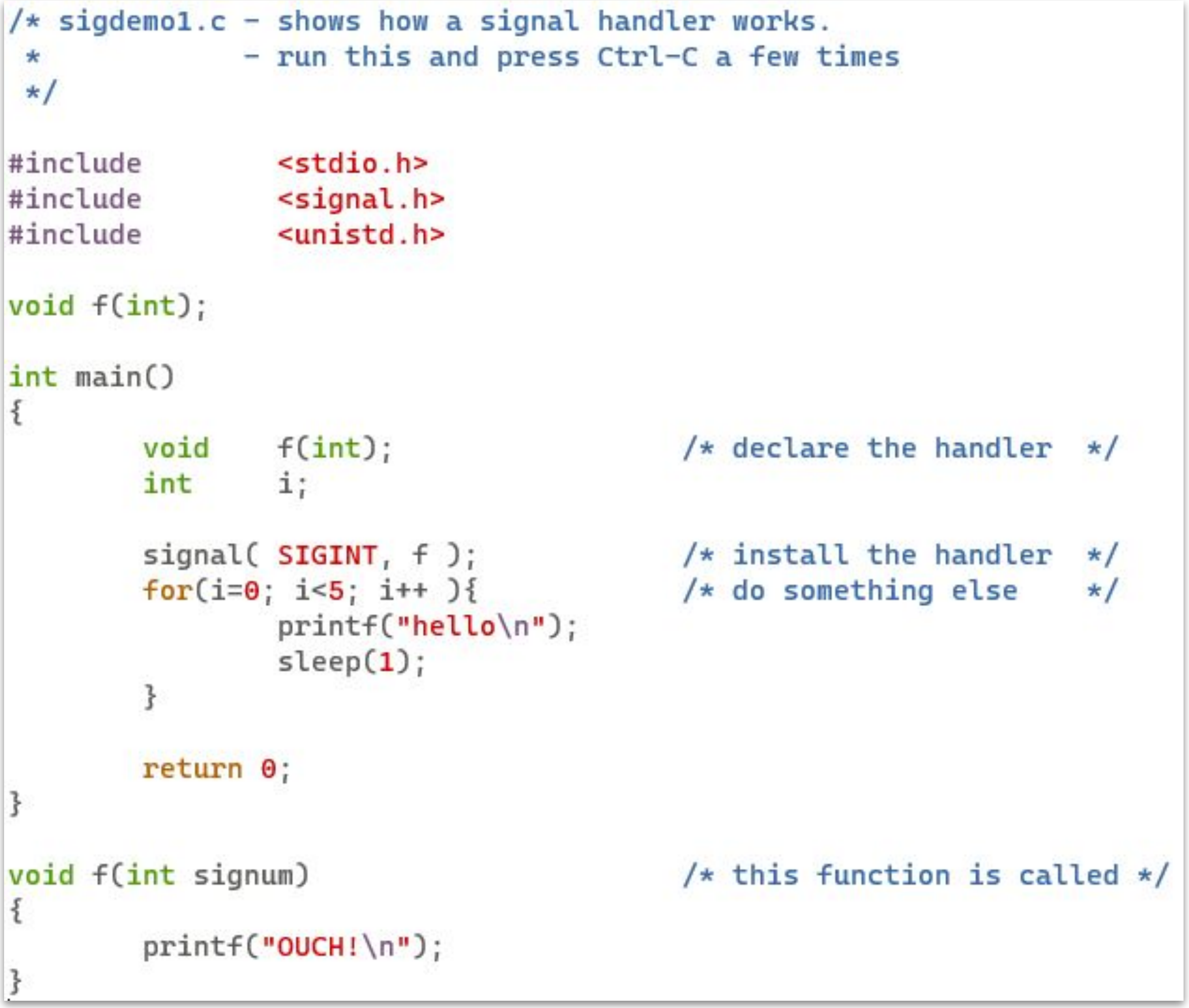
Signals (cont.)

- How to call a signal handler
  - `signal(`*`signum`*`, `*`function_name`*`)`



	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

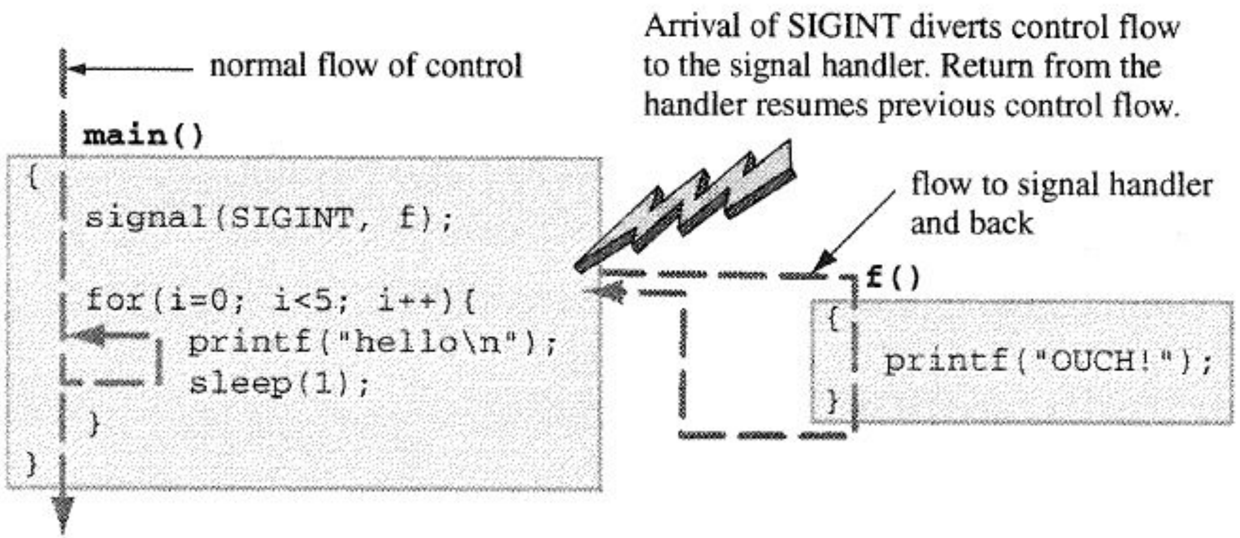
Signal Handling: `sigdemo1`



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

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

void f(int);

int 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);
        }

        return 0;
}

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

Signal Handling: `sigdemo1` (cont.)

- How `sigdemo1` works
  - There are two independent flows of control

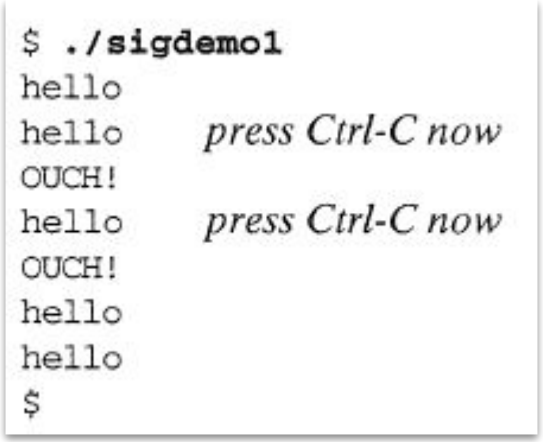


```
main()
{
    signal(SIGINT, f);

    for(i=0; i<5; i++){
        printf("hello\n");
        sleep(1);
    }
}
```

```
f()
{
    printf("OUCH!");
}
```

normal flow of control
Arrival of SIGINT diverts control flow to the signal handler. Return from the handler resumes previous control flow.
flow to signal handler and back

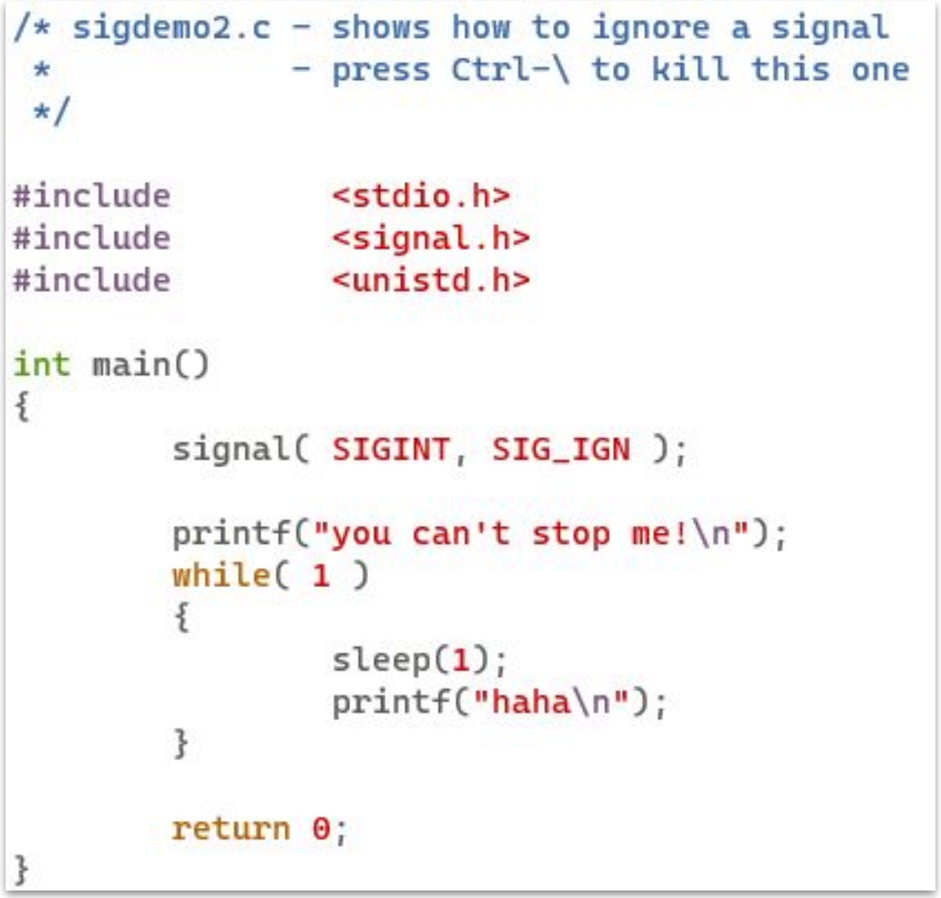
< A signal causes a subroutine call >



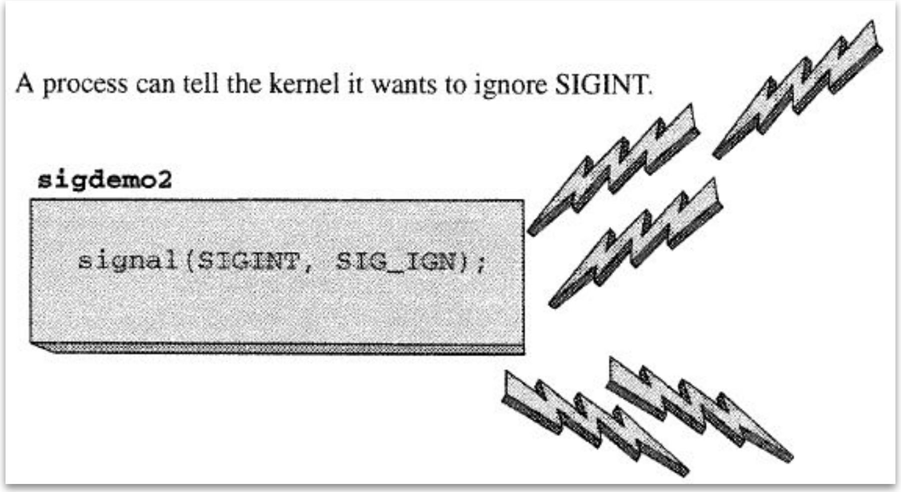
```
$ ./sigdemo1
hello
hello     press Ctrl-C now
OUCH!
hello     press Ctrl-C now
OUCH!
hello
hello
$
```

Signal Handling: sigdemo2

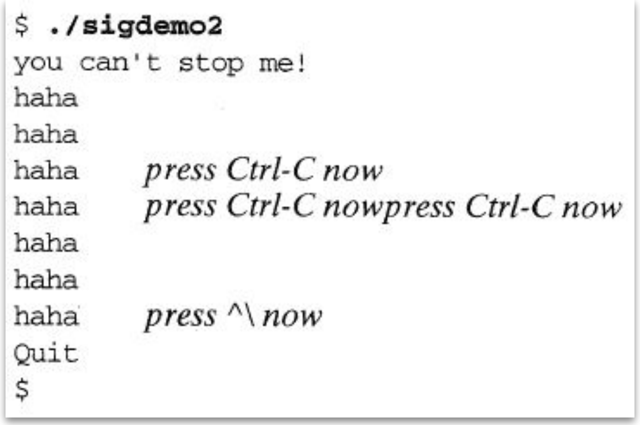
- Ignoring a signal
  - Ctrl-\ : SIGQUIT



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

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

int main()
{
        signal( SIGINT, SIG_IGN );

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

        return 0;
}
```



< The effect of signal (SIGINT, SIG_IGN) >



```
$ ./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
$
```

Summary

- Software tools: read stdin or files / write to stdout
  - Software tools view input and output as byte streams
  - Most processes automatically have 3 fd’s open
    - 0 (stdin), 1 (stdout), 2 (stderr)
    - The program does not need to call open for these
- User programs
  - Designed to be used by a human at a keyboard and screen
- A signal is a one-word message
  - e.g., green light, stop sign, umpire gesture, etc.

Summary (cont.)

- Device-specific programs have to control the connection to the device
  - Terminal is the most common and popular service
- A terminal driver has many settings
  - A collection of settings: a mode of the terminal driver
- Keys users press fall into the following three categories
  - Regular data: delivered through the driver
  - Editing functions (invoked by the keys)
    - e.g., erase key: removes the previous char from the line buffer and sends the codes to the terminal screen to remove that from the display
  - Process control functions: e.g., Ctrl-C key
- A signal is a short message from the kernel to a process
  - A process tells the kernel how to react upon receipt of a signal

Appendix

Make

* Most slides from CS352 by courtesy of Prof. Saumya Debray at U of Arizona and Prof. Suh at KNU

Structuring Large Applications

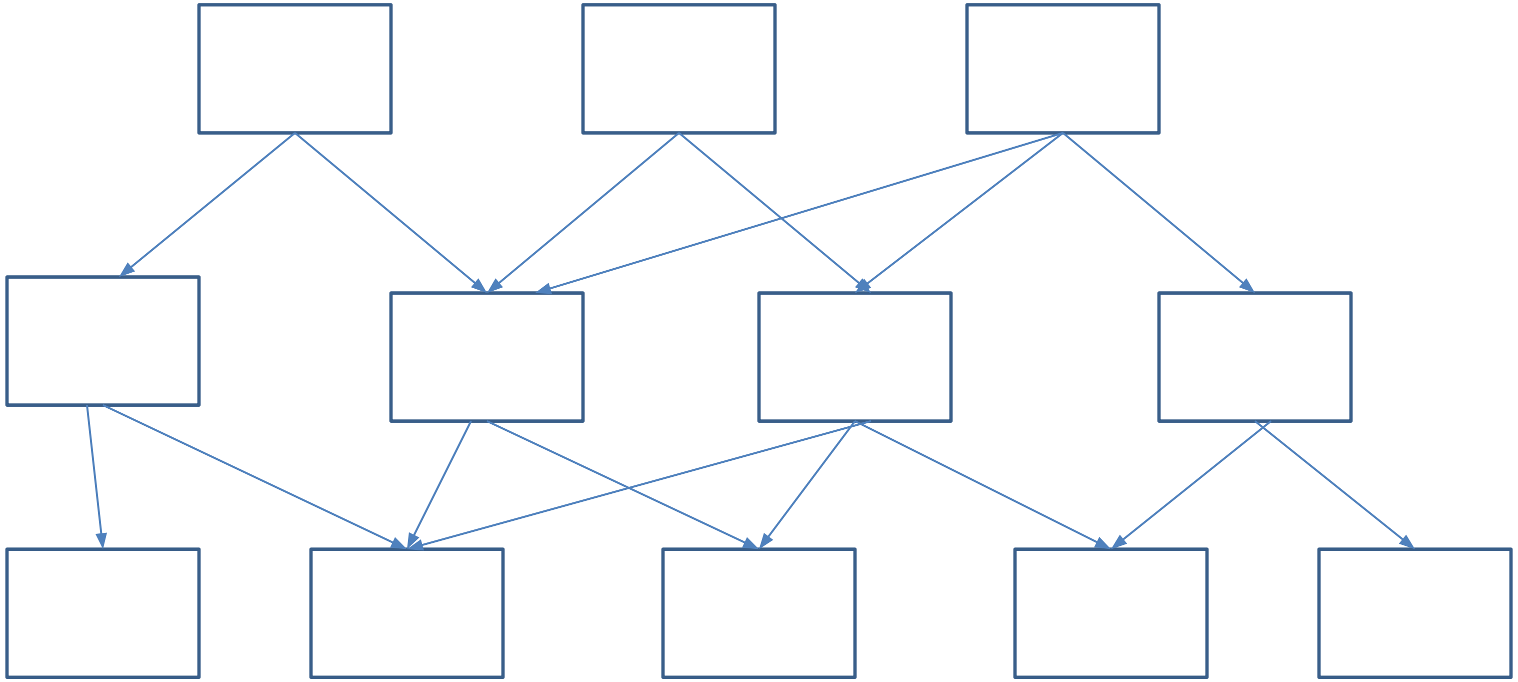
- So far, many of our programs have involved a single source file
  - Obviously impractical for large(r) programs
  - Even where practical, may not be good from a design perspective
- If an application is broken up into multiple files, we need to manage the build process:
  - How do we (re)compile the various different files that make up the application(s)?

Structuring Large Applications (cont.)

- When one file is edited, other files may need to be recompiled
  - Changes to typedefs or macros in header files
  - Changes to types of shared variables
- Applications can contain a lot of files
  - e.g., Linux kernel source code: about 5,000 files (totaling 15+M LOC)
- Re-compiling all files whenever any file is changed can be VERY time-consuming

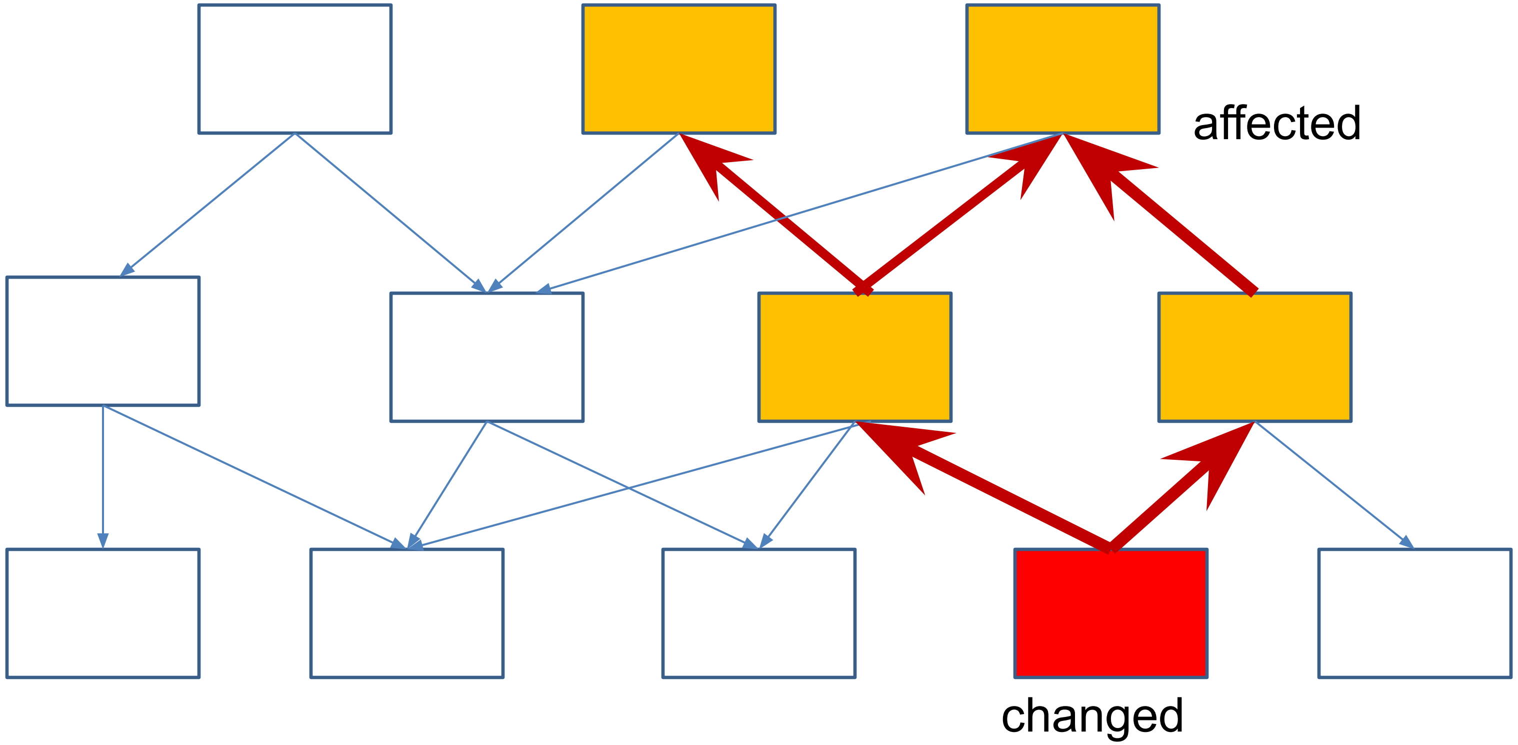
Structuring Large Applications (cont.)

- Obvious idea: only recompile those files that need to be recompiled



Structuring Large Applications (cont.)

- Obvious idea: only recompile those files that need to be recompiled



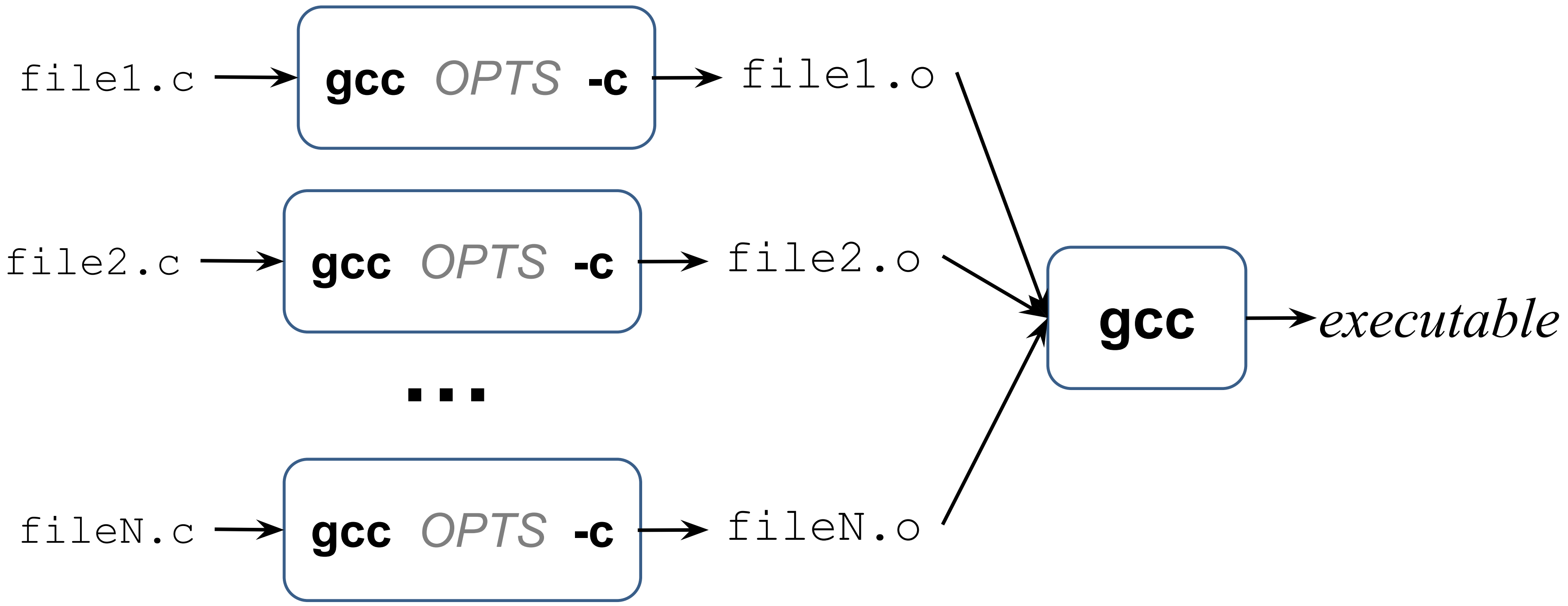
Structuring Large Applications (cont.)

- “Smart recompilation” : issues
  - Need to be able to express, or keep track of dependencies between files
    - “Dependency” ≈ which files are (might be) affected by a change to a file
  - Need to make sure that all (and only) affected files are recompiled
    - Doing this manually is tedious (지루한) and error-prone (오류 발생이 쉬운)
    - WANT an ***automated*** solution!!!
- `make`
  - a tool to automate recompilation of parts of a project based on a file of dependencies (“`make` *file*”)

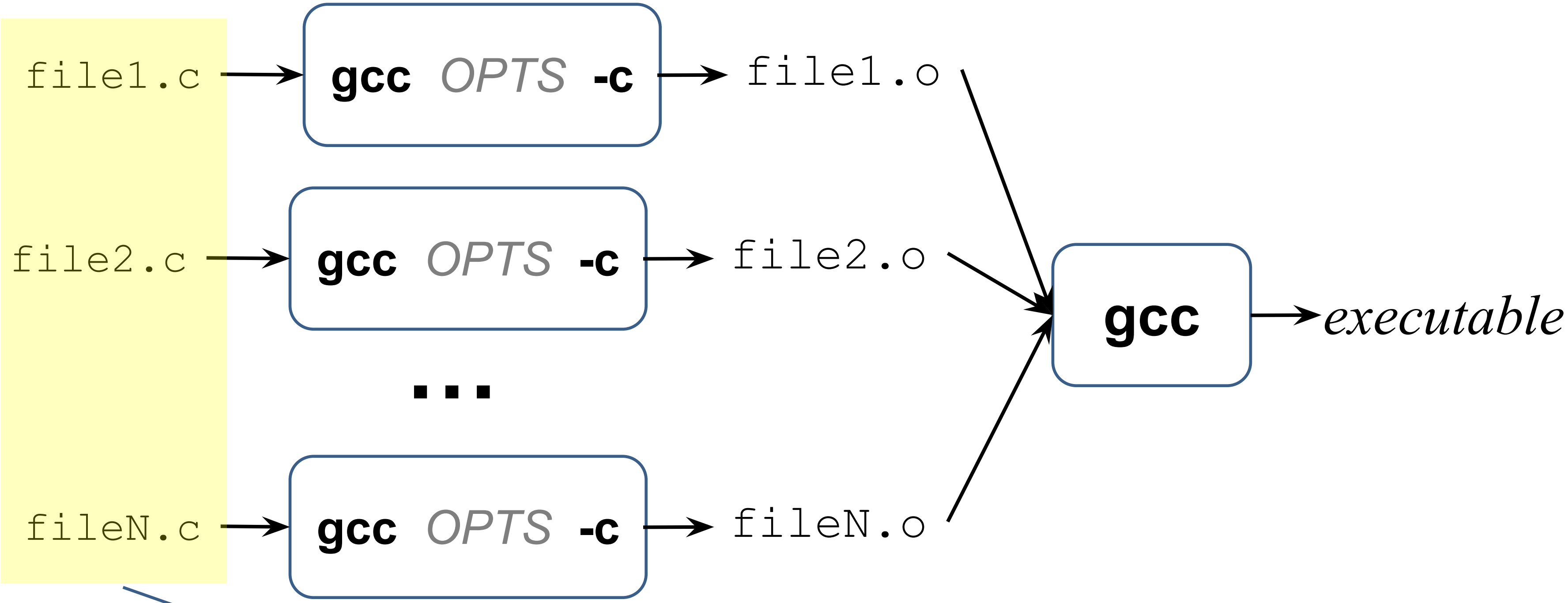
What is `make`?

- `make(1)` is a command generator and build utility
  - Using a description file (usually `Makefile`) it creates
- A sequence of commands for execution by the shell
  - used to sort out dependency relations among files
  - avoids having to rebuild the entire project after modification of a single source file
  - performs selective rebuilds following a dependency graph
  - allows simplification of rules through use of macros and suffixes, some of which are internally defined
  - different versions of make(1) (BSD make, GNU make, Sys V make, ...) may differ in e.g.:
    - variable assignment and expansion/substitution
    - including other files
    - flow control (for-loops, conditionals etc.)

Compiling Multi-File Programs



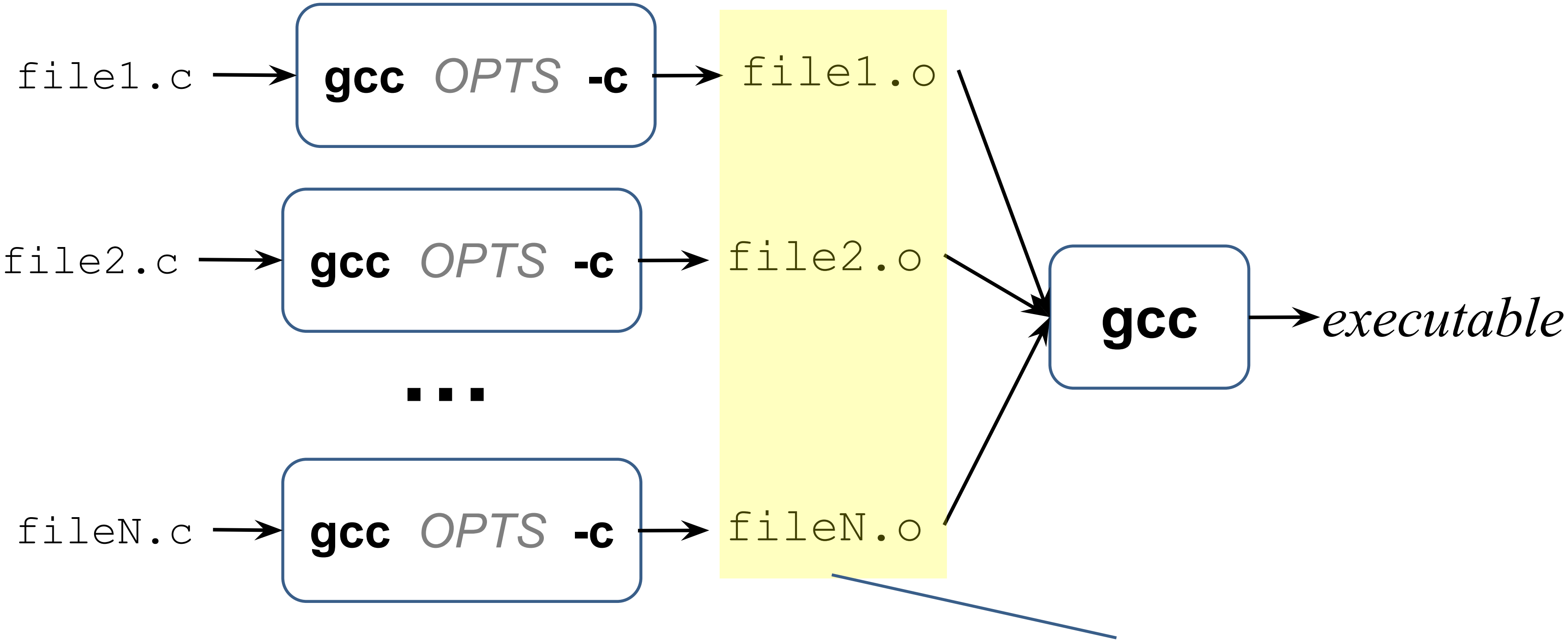
Compiling Multi-File Programs (cont.)



source files

Only one of these files contains `main()`

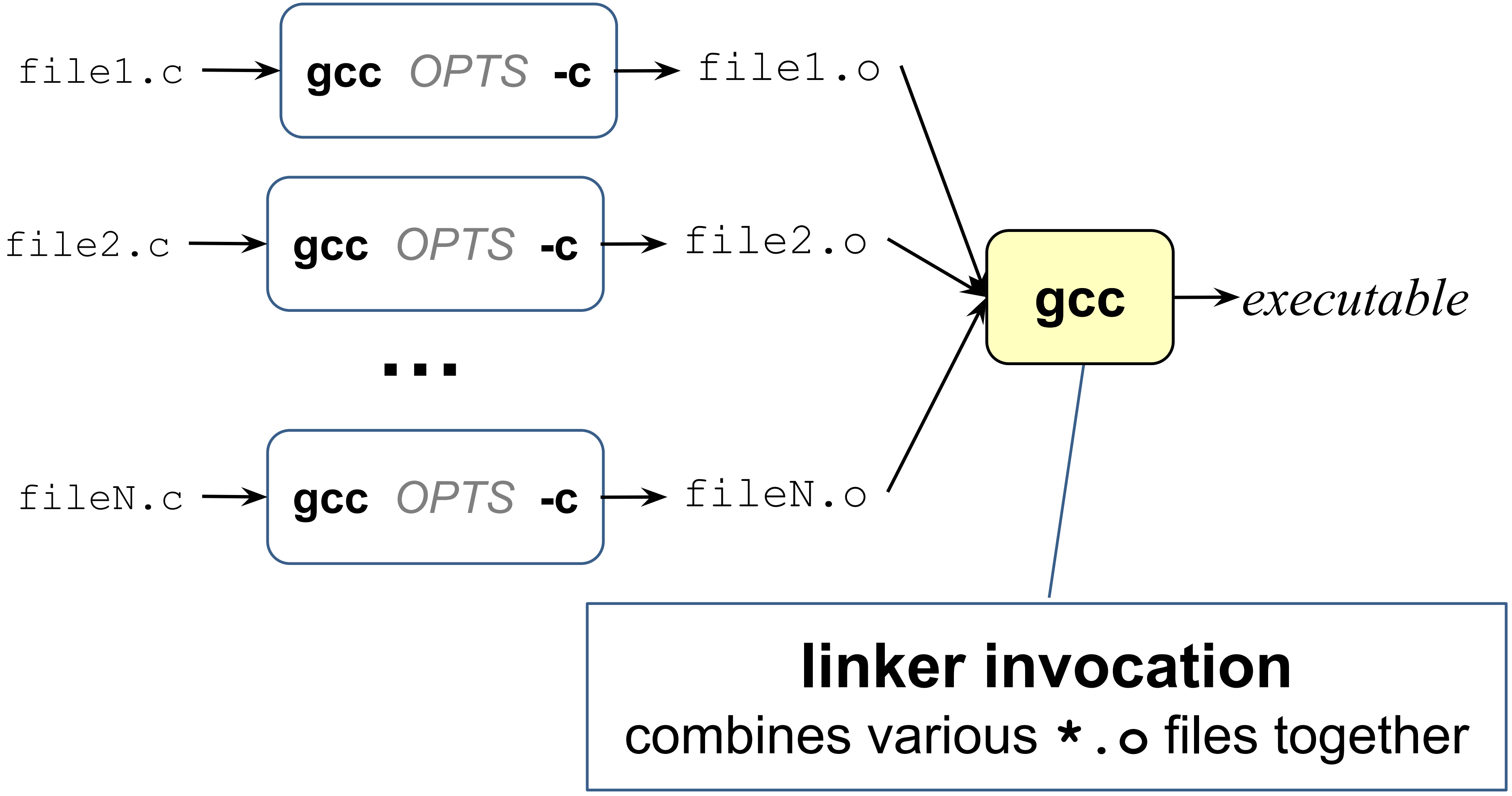
Compiling Multi-File Programs (cont.)



object files

machine code, but not executable

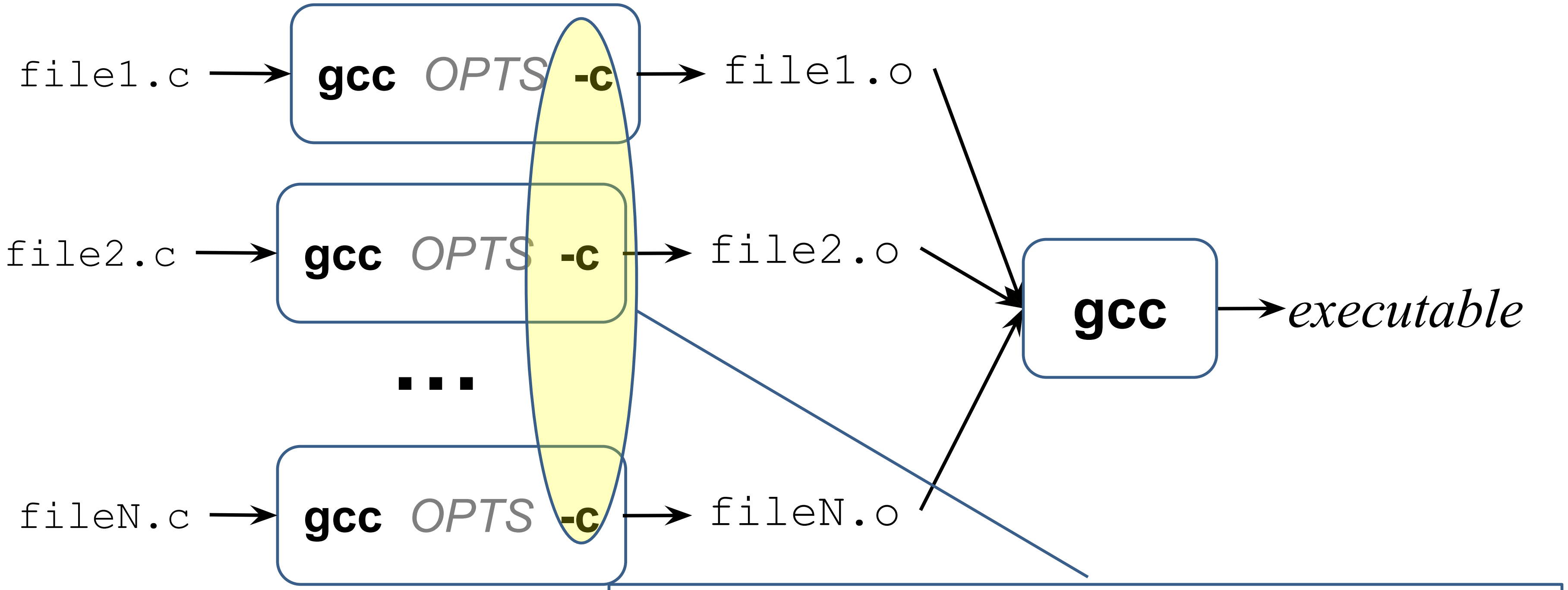
Compiling Multi-File Programs (cont.)



linker invocation

combines various `*.o` files together

Compiling Multi-File Programs (cont.)



file1.c → gcc *OPTS* -c → file1.o

file2.c → gcc *OPTS* -c → file2.o

...

fileN.c → gcc *OPTS* -c → fileN.o

file1.o, file2.o, fileN.o → gcc → *executable*

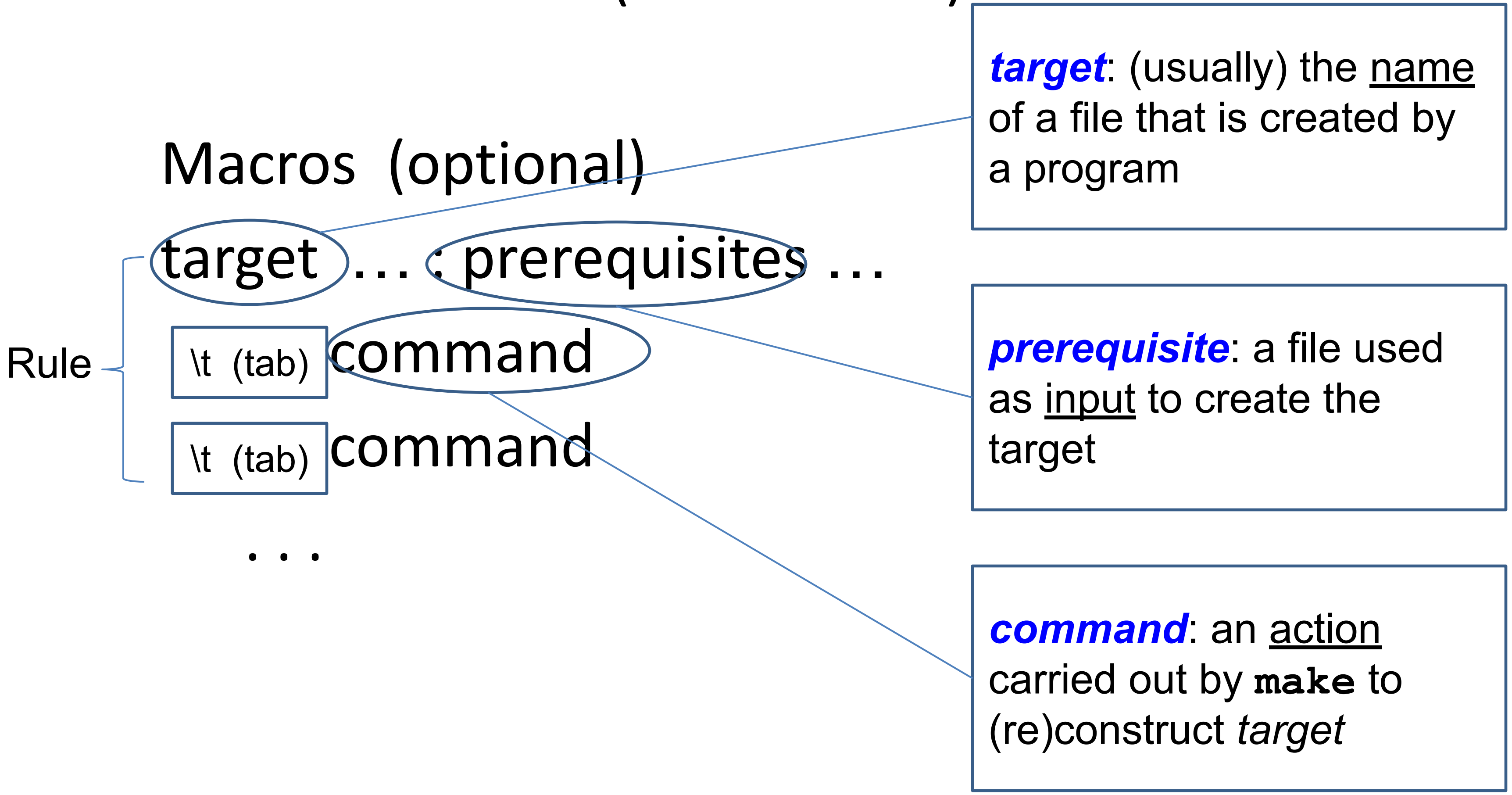
`gcc   -c`

compile to a linkable object

don't worry about `main()`

Makefile: Structure

Structure of a makefile (`Makefile`):



Macros (optional)

```
target … : prerequisites …
\t (tab) command
\t (tab) command
. . .
```

Rule

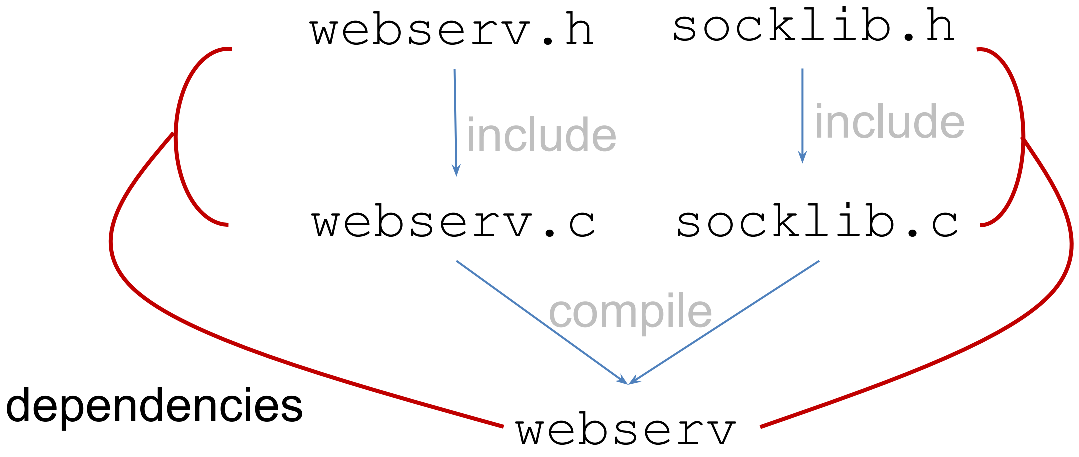
target: (usually) the name of a file that is created by a program

prerequisite: a file used as input to create the target

command: an action carried out by **`make`** to (re)construct *target*

Makefile: An Elementary Example

- Dependency structure:



webserv.h socklib.h

include include

webserv.c socklib.c

compile

dependencies

webserv

A Makefile file:

```
webserv: webserv.c  webserv.h socklib.c  socklib.h
	gcc -Wall webserv.c socklib.c
```

must be a tab!

Creating a `Makefile` file

- 1. What are the targets?
    - Figure out which files are ***created*** from other files and which need to be ***re-created*** when any of those files change
- 2. For each target, say `foo`:
    - What are the files which, if changed, would require us to re-create `foo`?
        - These are the prerequisites for `foo` (let’s say *bar_1* ... *bar_n*)
- 3. What commands do we use to (re-)create `foo`?
    - say: *cmd_1* ... *cmd_m*

Creating a `Makefile` file (cont.)

- The resulting rule for `foo` is:

$$\begin{array}{l}
\texttt{foo: bar}_{1}\quad\texttt{bar}_{2}\quad\texttt{...}\ \texttt{bar}_{n}\\
\boxed{\text{tab}}\ \texttt{cmd}_{1}\\
\boxed{\text{tab}}\ \texttt{cmd}_{2}\\
\qquad\ \texttt{...}\\
\boxed{\text{tab}}\ \texttt{cmd}_{m}
\end{array}$$

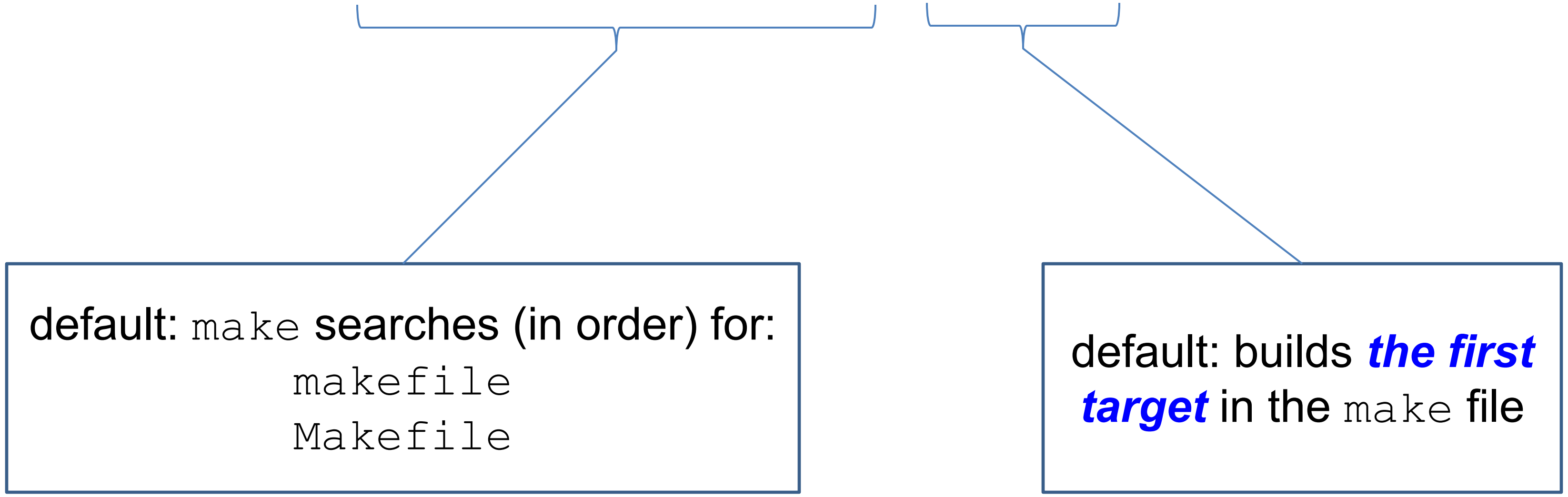
or:

$$\begin{array}{l}
\texttt{foo: bar}_{1}\quad\texttt{bar}_{2}\quad\texttt{...}\ \texttt{bar}_{n}\\
\boxed{\text{tab}}\ \texttt{cmd}_{1}\texttt{; cmd}_{2}\texttt{; ... cmd}_{m}
\end{array}$$

How to use make?

- Invocation:

make [-f *makeFileName*] [*target*]

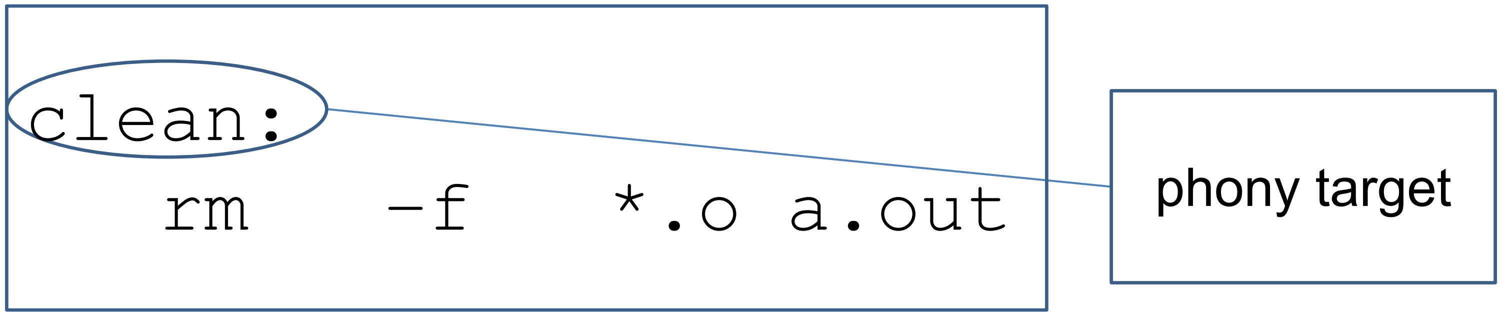


default: make searches (in order) for:
makefile
Makefile

default: builds ***the first target*** in the make file

`Makefile`: Phony Targets

- A *phony target* is one that is not the name of a file
  - used to run a recipe (i.e., a set of commands) to be executed when an explicit request is made



```
clean:
      rm   -f    *.o a.out
```

phony target

  - “`make clean`” will remove `a.out` and `*.o` files

`Makefile`: Phony Targets (cont.)

```
clean:
            rm    -f    *.o   a.out
```

- This won’t work if we create a file named “`clean`” by accident
- Fix:

```
.PHONY:  clean


 clean:
    rm  *.o  a.out
```

cleanup actions will be executed even if there is a file named “`clean`”

`Makefile`: Macros

- Makes `make` files easier to write, modify
  - Define: `Name` = replacement list
  - Use: `$(Name)`
- Example

```
CC = gcc
OPTLEV = -O2       # optimization level
CFLAGS = -Wall -g -D DEBUG $(OPTLEV) -c
. . .
file1.o  :  file1.c  hdrfile1.h
   $(CC)   $(CFLAGS)  file1.c
```

Makefile: Macros (cont.)

- "gcc –D" defines a macro to be used by the preprocessor

```
// myfile.c
#include <stdio.h>

void main(){
    #ifdef DEBUG
       printf("Debug run\n");
    #else
       printf("Release
run\n");
    #endif
}
```

```
$ gcc -D DEBUG myfile.c -o myfile
$ ./myfile
Debug run
$
```

How `make` Works

- When invoked, begins processing the appropriate target
- For each **target**, considers the prerequisites it depends on:

  ```
  target : file1  file2  …
  ```

  - Checks (recursively) whether each of *file*$_i$ (1) exists and (2) is more recent than the files that *file*$_i$ depends on;
    - if not, executes the associated command(s) to update *file*$_i$
  - Checks whether target exists and is more recent than *file*$_i$
    - if not, executes the commands associated with *target*
- Commands associated with each rule had better be concatenated by “;”
  - e.g., *cmd*$_1$; *cmd*$_2$; *cmd*$_3$; …
  - If a command returns an error (with a nonzero exit value), `make` abandons that rule; to ignore errors in a command, precede with ‘-’

How `make` Works (cont.)

`Makefile`

$\texttt{file}_a\texttt{:}\ \texttt{file}_b\quad\texttt{file}_c$

$\quad\texttt{cmd}_a$

$\texttt{file}_b\texttt{:}\ \texttt{file}_e\quad\texttt{file}_d$

$\quad\texttt{cmd}_b$

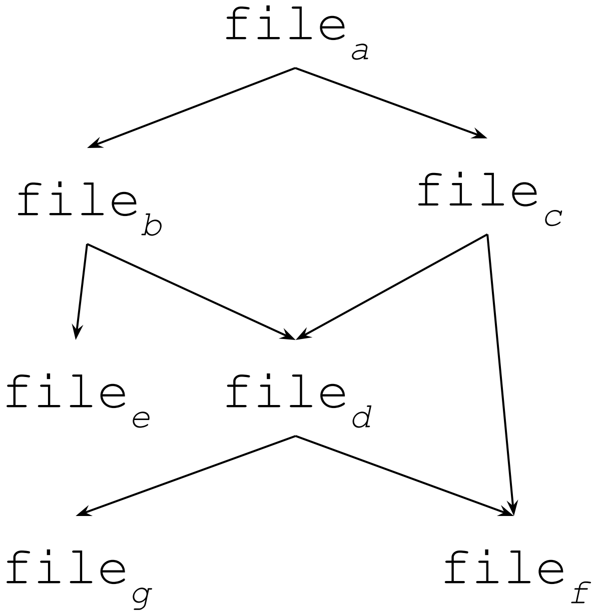
$\texttt{file}_c\texttt{:}\ \texttt{file}_d\quad\texttt{file}_f$

$\quad\texttt{cmd}_c$

$\texttt{file}_d\texttt{:}\ \texttt{file}_f\quad\texttt{file}_g$

$\quad\texttt{cmd}_d$

Dependence structure

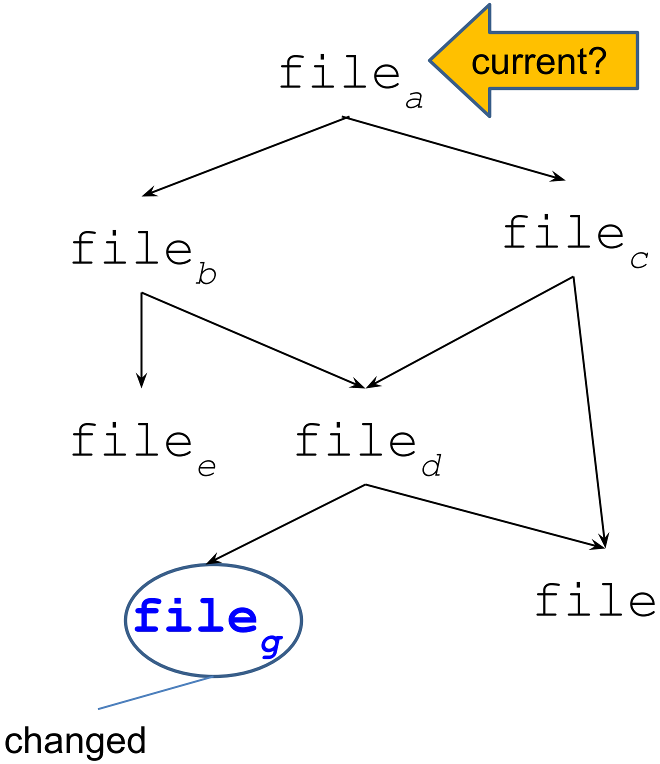


How `make` Works (cont.)

`Makefile`

```
▶ file_a: file_b    file_c
      cmd_a
  file_b: file_e   file_d
      cmd_b
  file_c: file_d   file_f
      cmd_c
  file_d:  file_f  file_g
      cmd_d
```

`make` execution

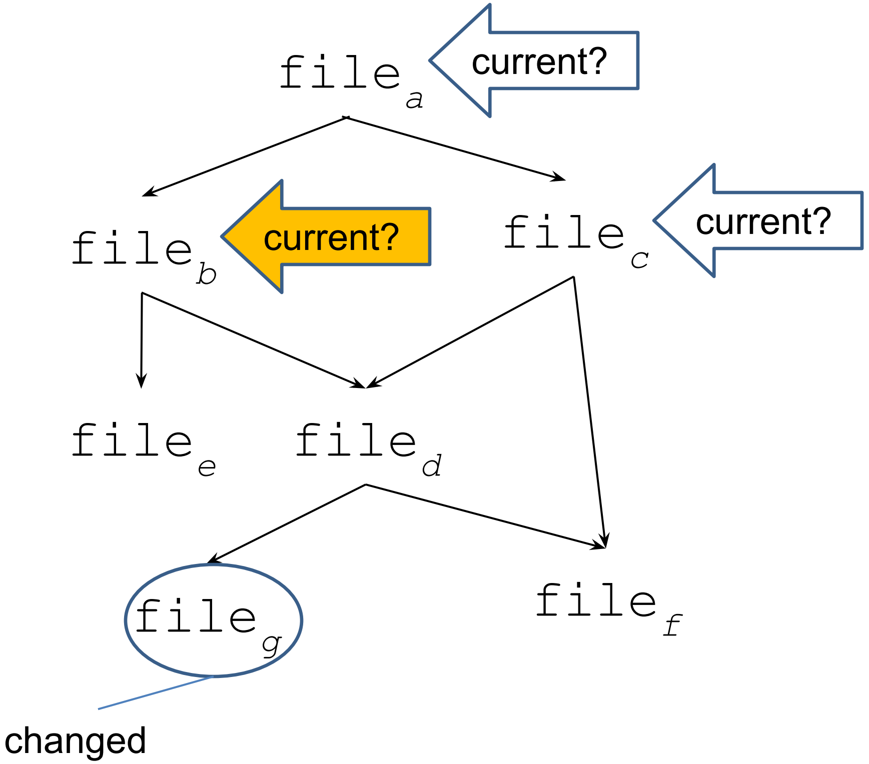


How `make` Works (cont.)

`Makefile`

```
file_a: file_b    file_c
    cmd_a
▶ file_b: file_e   file_d
    cmd_b
file_c: file_d   file_f
    cmd_c
file_d:  file_f  file_g
    cmd_d
```

`make` execution

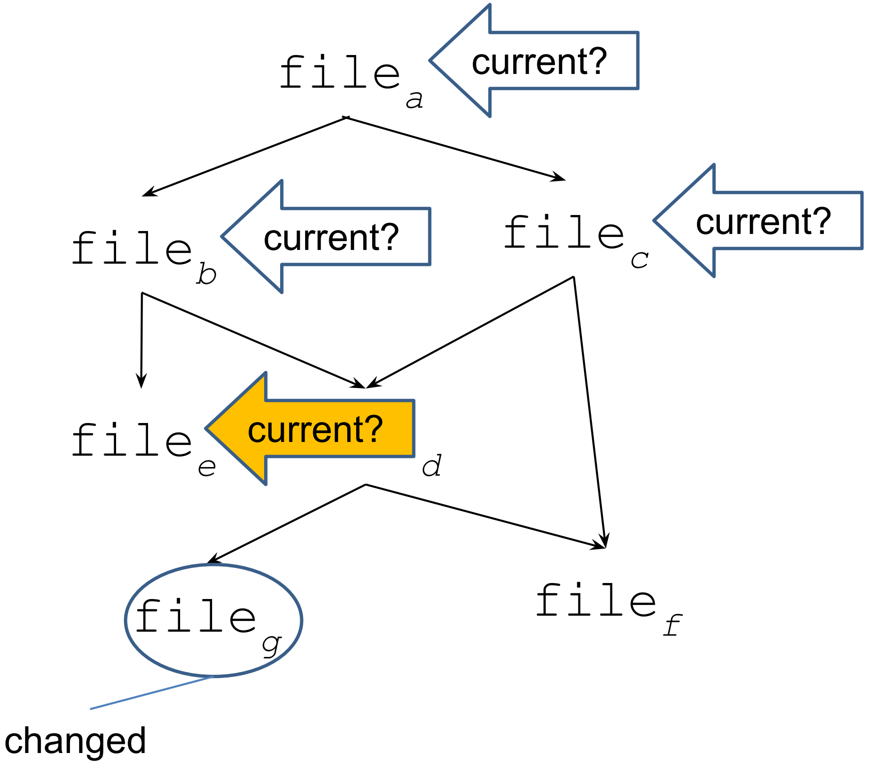


How make Works (cont.)

Makefile

file$_a$: file$_b$ file$_c$
    cmd$_a$
▶ file$_b$: file$_e$ file$_d$
    cmd$_b$
file$_c$: file$_d$ file$_f$
    cmd$_c$
file$_d$: file$_f$ file$_g$
    cmd$_d$

make execution



How `make` Works (cont.)

`Makefile`

$\texttt{file}_a$: $\texttt{file}_b$ $\texttt{file}_c$

  $\texttt{cmd}_a$

$\texttt{file}_b$: $\texttt{file}_e$ $\texttt{file}_d$

  $\texttt{cmd}_b$

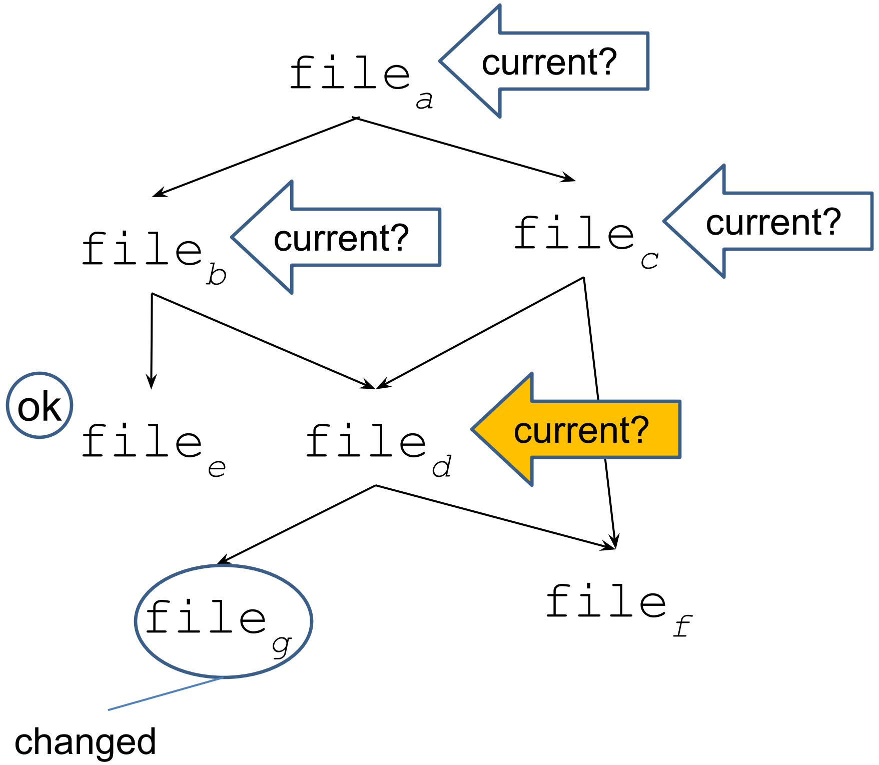
$\texttt{file}_c$: $\texttt{file}_d$ $\texttt{file}_f$

  $\texttt{cmd}_c$

▶ $\texttt{file}_d$: $\texttt{file}_f$ $\texttt{file}_g$

  $\texttt{cmd}_d$

`make` execution

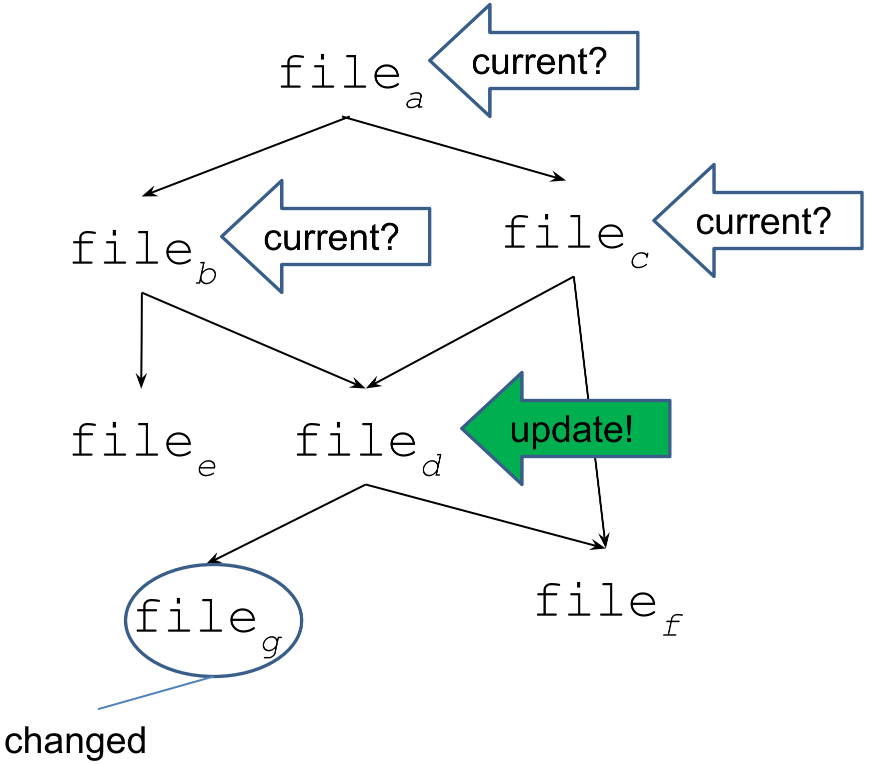


How `make` Works (cont.)

`Makefile`

$\texttt{file}_a\texttt{:}\ \texttt{file}_b\ \ \texttt{file}_c$
$\quad\texttt{cmd}_a$
$\texttt{file}_b\texttt{:}\ \texttt{file}_e\ \ \texttt{file}_d$
$\quad\texttt{cmd}_b$
$\texttt{file}_c\texttt{:}\ \texttt{file}_d\ \ \texttt{file}_f$
$\quad\texttt{cmd}_c$
$\texttt{file}_d\texttt{:}\ \texttt{file}_f\ \ \texttt{file}_g$
$\quad\texttt{cmd}_d$

`make` execution

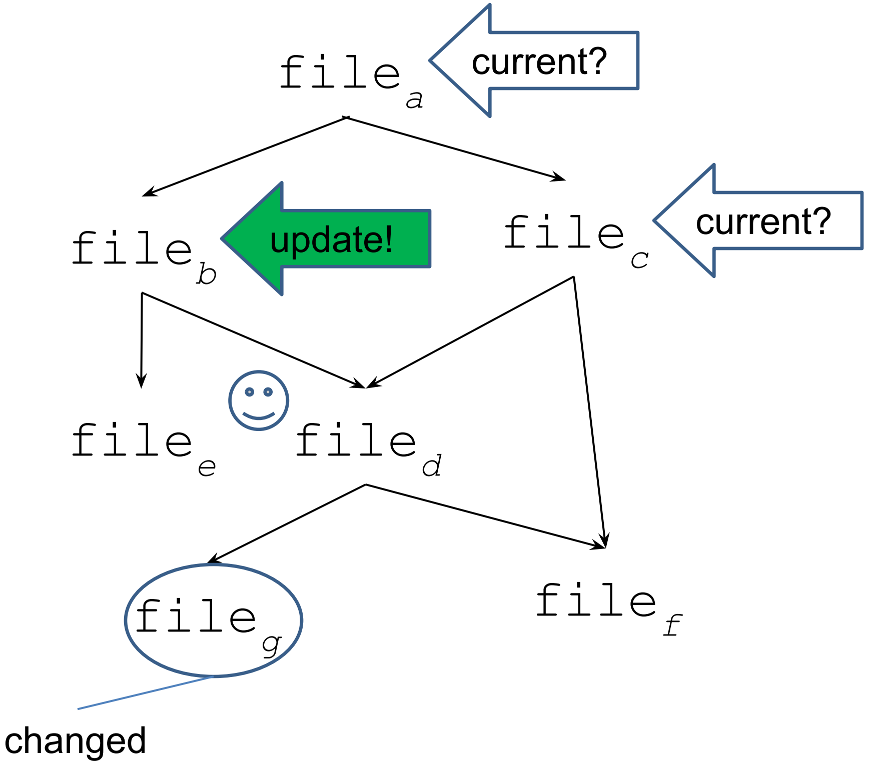


How make Works (cont.)

Makefile

```
file_a: file_b   file_c
    cmd_a
file_b: file_e   file_d
    cmd_b
file_c: file_d   file_f
    cmd_c
file_d:  file_f  file_g
    cmd_d
```

make execution

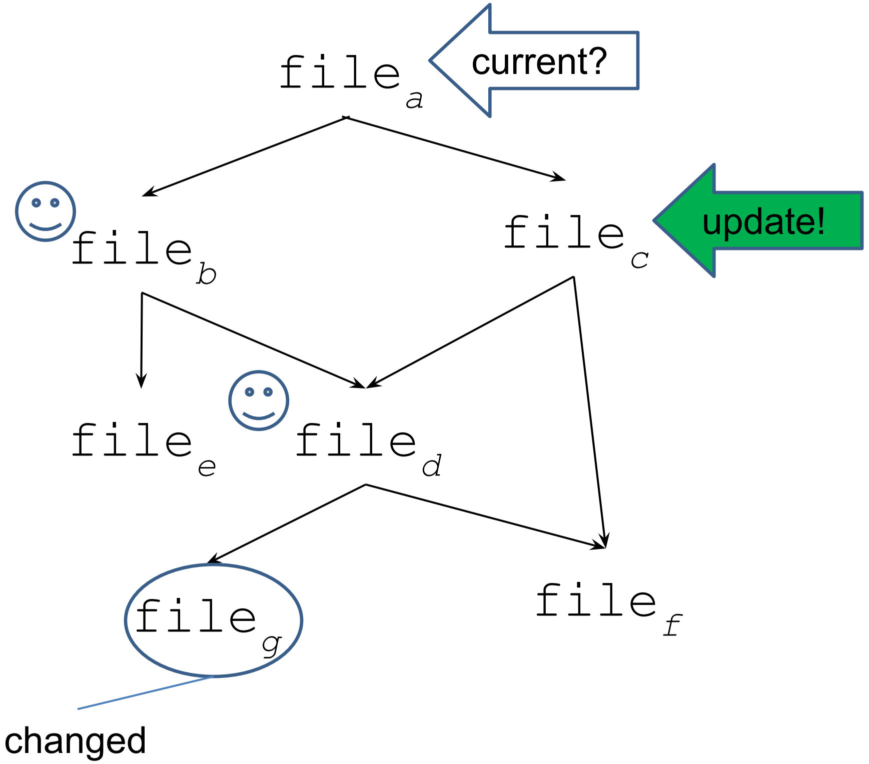


How make Works (cont.)

Makefile

$\texttt{file}_a$: $\texttt{file}_b$ $\texttt{file}_c$
    $\texttt{cmd}_a$
$\texttt{file}_b$: $\texttt{file}_e$ $\texttt{file}_d$
    $\texttt{cmd}_b$
$\texttt{file}_c$: $\texttt{file}_d$ $\texttt{file}_f$
    $\texttt{cmd}_c$
$\texttt{file}_d$: $\texttt{file}_f$ $\texttt{file}_g$
    $\texttt{cmd}_d$

make execution

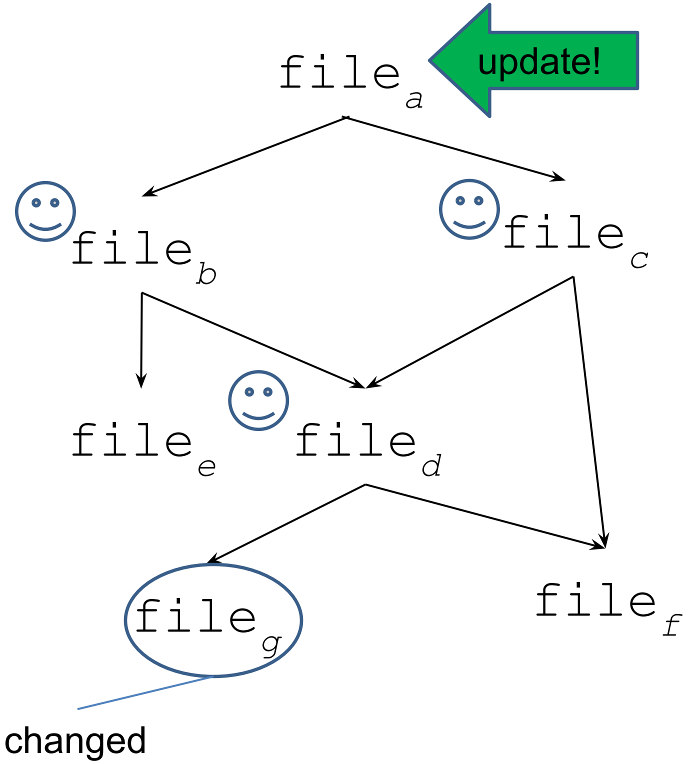


How `make` Works (cont.)

`Makefile`

$\texttt{file}_a$: $\texttt{file}_b$ $\texttt{file}_c$
    $\texttt{cmd}_a$
$\texttt{file}_b$: $\texttt{file}_e$ $\texttt{file}_d$
    $\texttt{cmd}_b$
$\texttt{file}_c$: $\texttt{file}_d$ $\texttt{file}_f$
    $\texttt{cmd}_c$
$\texttt{file}_d$: $\texttt{file}_f$ $\texttt{file}_g$
    $\texttt{cmd}_d$

`make` execution



How `make` Works (cont.)

`Makefile`

$\texttt{file}_a\texttt{:}\ \texttt{file}_b\quad \texttt{file}_c$

$\quad\texttt{cmd}_a$

$\texttt{file}_b\texttt{:}\ \texttt{file}_e\quad \texttt{file}_d$

$\quad\texttt{cmd}_b$

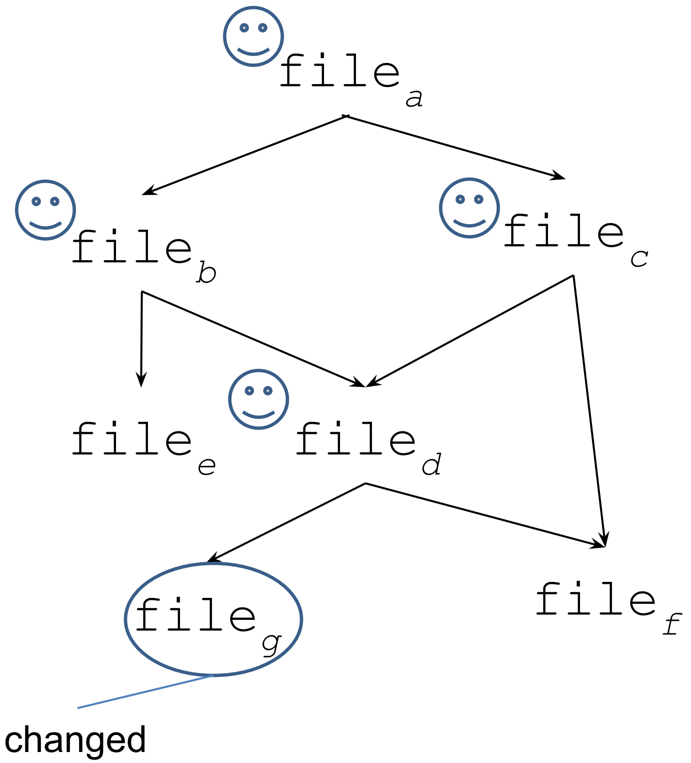
$\texttt{file}_c\texttt{:}\ \texttt{file}_d\quad \texttt{file}_f$

$\quad\texttt{cmd}_c$

$\texttt{file}_d\texttt{:}\ \texttt{file}_f\quad \texttt{file}_g$

$\quad\texttt{cmd}_d$

`make` execution



Topics Not Covered

- `make` has a lot of functionality we won’t get to cover,
  - e.g., implicit rules, implicit variables, conditional parts of make files, recursively running make in subdirectories

- See online `make` tutorials for more information

Summary

- Typically, large applications are written by multiple source files
  - Recompiling all files due to a single update is ridiculous
- `make` is a tool designed to automate a building process for such large programs
  - Recompiles those files that need to be recompiled due to a change(s)
  - Its targets must be determined
    - Phony targets (e.g., `clean, install,` etc) may be specified to avoid a conflict and to improve performance
  - Searches by default for `makefile` or `Makefile`
    - Any target that it first meets will be first processed
  - Its macros would help makefiles to be written and modified more easily
  - When invoked, it begins processing the appropriate target