CS 111
Operating Systems Principles

Section 1E

Friday 2-4 PM

Tengyu Liu

Your TA

- Tengyu Liu
  - 4th year Ph.D. focusing on Computer Vision and Artificial Intelligence
  - 24.jason.5@gmail.com , 424-443-4763
- Three TAs in total

Name	Discussion Section Time	Discussion Section Link	Office Hour Time	Office Hour Link
Tianxiang Li	F 10-11:50 AM	https://ccle.ucla.edu/mod/zoom/view.php?id=3564595	Monday 4-6 PM	https://ucla.zoom.us/j/5471431322?pwd=bjhlOFpGZktubFJJeE9NdEhyNHF3QT09
Alexandre Tiard	F 12-1:50 PM	https://ccle.ucla.edu/mod/zoom/view.php?id=3565435	Tuesday 9:30-11:30AM	https://ucla.zoom.us/j/98415880816?pwd=QlVlOXV6bWlDelY5VmJNb0xlYUhqZz09
Tengyu Liu	F 2-3:50 PM	https://ccle.ucla.edu/mod/zoom/view.php?id=3559397	Sunday 7-9 PM	https://ucla.zoom.us/j/3666922249

Logistics

- Discussion section 1A
  - Time: Every Friday 2-4 PM
  - Zoom link can be found on CCLE site info and piazza pinned post
  - Content: Help you navigate through your projects
  - Interruption is expected and welcomed

Logistics – QA

Piazza

- Use Piazza so that your classmates can
  - Answer your question
  - Benefit from your post
- … unless it is inappropriate to do so
  - Avoid sharing your solution, even if it’s buggy
- Ask clarification questions and questions on abstraction level
- Check for existing questions before asking it
  - We won’t answer duplicated questions

Office Hour

- OH will be one-on-one
- If I am helping another person, you will be put in a "waiting room"
- This will be very inefficient, but is necessary to prevent plagiarism
- Ask your question on Piazza if possible.
- I can help you debug your code, but I will not write your code for you.

Logistics – Projects

- 10 projects in total
  - 0: Warm up
  - 1 A/B : I/O, Inter-process communication, Networking
  - 2 A/B : Concurrency, parallelism
  - 3 A/B : Explore/Diagnose filesystem
  - 4 A/B/C : IoT
- Projects are due on Wednesdays
- Projects worth 53% of your grade

- Late penalty is 2^{N-1}% for N late days
  - 1 day late $= 2^0 = 1$ pt off
  - 2 days late $= 2^1 = 2$ pts off
  - …
  - 7 days late $= 2^6 = 64$ pts off
  - 8 days late, you get nothing
- Ignore "Slip Days" if they are mentioned
  - They're an artefact of another late policy that we will not apply this quarter

Logistics – Exponential Late Penalty

- Exponential late penalty is lenient at first but extremely harsh for over 4 days
- Designed to accommodate emergencies, not procrastination
- One-minute past midnight is counted as 1 day late. So please plan to submit on time so network delays can cost you 1 extra point at most.
- Assume each project takes roughly 3-5 days to finish

	Release date	Due date	Start date	# Days taken	Submission date	Late penalty	1-minute late penalty
A	Wed	Wed (+1)	Wed	7	Tue (+1)	0	0
B	Wed	Wed (+1)	Fri	5	Tue (+1)	0	0
C	Wed	Wed (+1)	Sun	5	Thu (+1)	1	2
D	Wed	Wed (+1)	Wed (+1)	3	Fri (+1)	2	4
E	Wed	Wed (+1)	Wed (+1)	5	Sun (+1)	8	16
F	Wed	Wed (+1)	Wed (+1)	7	Tue (+2)	32	64

Projects – General Comments

- Preparing
  - Read the specs
  - Read manual pages
  - They tend to be complicated, but reading them thoroughly is an important skill
- Submission
  - You will need to submit
    - Readme
      - Be careful with the syntax ("NAME:" is NOT the same as "name:")
      - Include sources you used at the bottom of the readme
      - Include a short description of the files (can be a single line)
    - Makefile
      - If you are unfamiliar with Makefiles, learn it now!
  - Test your code before submission
  - Re-download your submission to a different directory to make sure
    - It was submitted
    - The contents are correct

Projects – Plagiarism

- *Plagiarism includes, but is not limited to, the use of <u>another person's work</u> (including words, ideas, designs, or data) without giving appropriate attribution or citation. – UCLA Student Conduct Code*
- If we suspect an act of plagiarism, we are obligated to report to the Dean of Students.
- We use software and manual review to detect plagiarism
- Although we absolutely hate to do this, we had to report several infractions to the dean each quarter.

Projects – Connection

- Connect to UCLA VPN
  - https://www.it.ucla.edu/bol/services/virtual-private-network-vpn-clients
- Once you are connected to VPN, connect to Linux server
  - SEASNET Account
    - If you don't have one, apply at https://www.seas.ucla.edu/acctapp/
    - Once you have your account, follow https://www.seasnet.ucla.edu/lnxsrv/
  - Connect to Linux server by
    - ssh <username>@lnxsrv09.seas.ucla.edu
- Make sure you test your code on lnxsrv09
  - A submission that compiles and runs on Mac/Windows/Ubuntu does not mean it works on lnxsrv09

- Running your code on lnxsrv09
  - Many versions of GCC on lnxsrv09
  - Put `/usr/local/cs/gcc-9.3.0/bin` at the beginning of your PATH variable
  - Each time you log out, the PATH variable restores itself
  - Add this line to `~/.profile` or `~/.bash_profile`
    - `PATH=/usr/local/cs/gcc-9.3.0/bin:$PATH`
  - Then log out and log back in
    - `~/.profile` and `~/.bash_profile` is executed when you log in to your account in the server
    - Putting PATH=… in those files sets the PATH variable to its appropriate value each time you log in
  - Double check that the appropriate gcc is used by typing
    - `which gcc`

Project 0

Warm Up

Project 0 – Goal

- Write a program that
  - copies its standard input to its standard output
  - If no errors (other than EOF) are encountered, exit(2) with a return code of 0.
  - Four optional arguments
    - --input=filename
      - Use specified file as standard input
    - --output=filename
      - Use specified file as standard output
    - --segfault
      - Force a segfault
    - --catch
      - Use a signal handler to catch the segfault

- Deliverables – a single compressed tarball
  - .tar.gz
  - A single C source module that compiles with no errors or warnings
  - A Makefile to build the program and the tarball
    - (default) – build the executable
    - check – runs a quick smoke-test
      - Describe your smoke tests in your README file
    - clean – delete all the files created by Makefile
      - Executables, tarball, ect.
    - dist – build the submission tarball
  - Two screenshots from gdb
  - A README file

Project 0 – Table of Content

- Tools
  - tar
  - gdb
- File descriptors
- I/O Redirection
- `getopt_long(3)`
- `signal(2)`

Project 0 – Tools

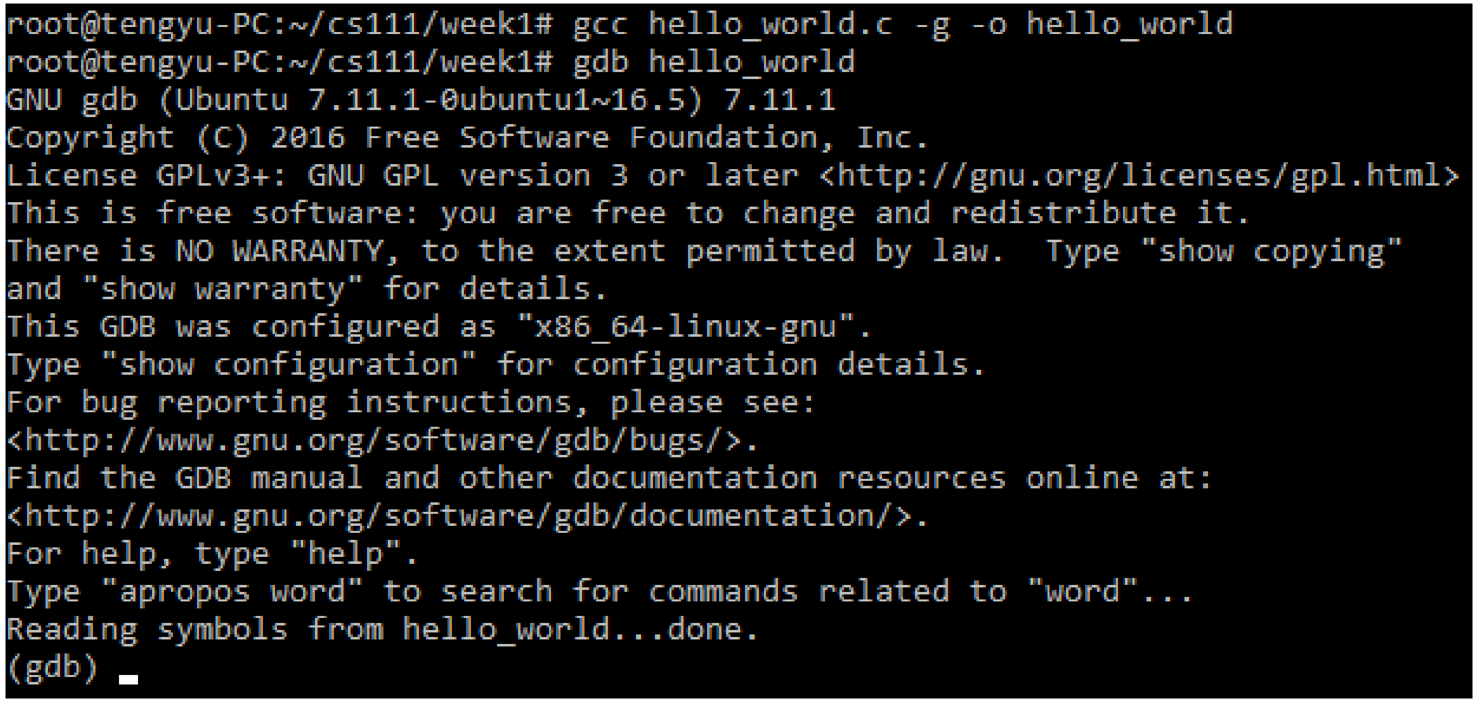
- For this project, you will need
  - Linux
  - gcc, libc, make, gdb
  - tar
- Install gcc, libc, make and gdb with
  - `sudo apt-get install build-essential`

Project 0 – Tools

- `tar`
    - Compression: `tar -czvf` example.tar.gz file1.pdf file2.png source.c
    - Extraction: `tar -xzvf` example.tar.gz
    - What are `-czvf` and `-xzvf`?
        - `-c`: Create an archive
        - `-z`: Compress the archive with gzip
        - `-v`: Verbose. Display progress while creating the archive. This is optional.
        - `-f`: Allows you to specify the filename of the archive
        - `-x`: Extract from an archive

Project 0 – Tools

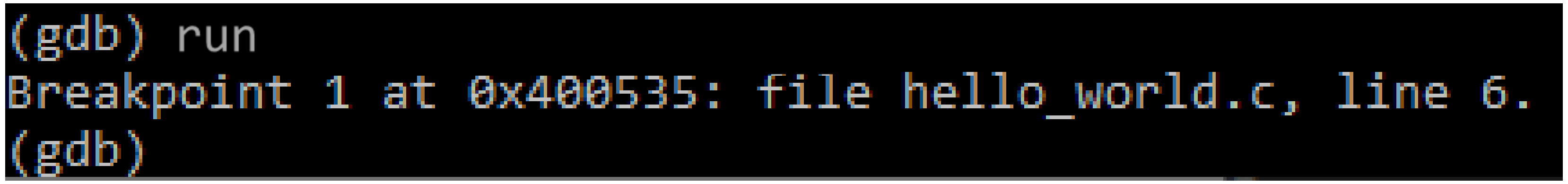
- gdb
  - Terminal-based debugger
  - May seem scary at first, but as powerful as any IDE-based debugger



```
root@tengyu-PC:~/cs111/week1# gcc hello_world.c -g -o hello_world
root@tengyu-PC:~/cs111/week1# gdb hello_world
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from hello_world...done.
(gdb) _
```

gdb – GNU Debugger

- Running the program from gdb with `run`

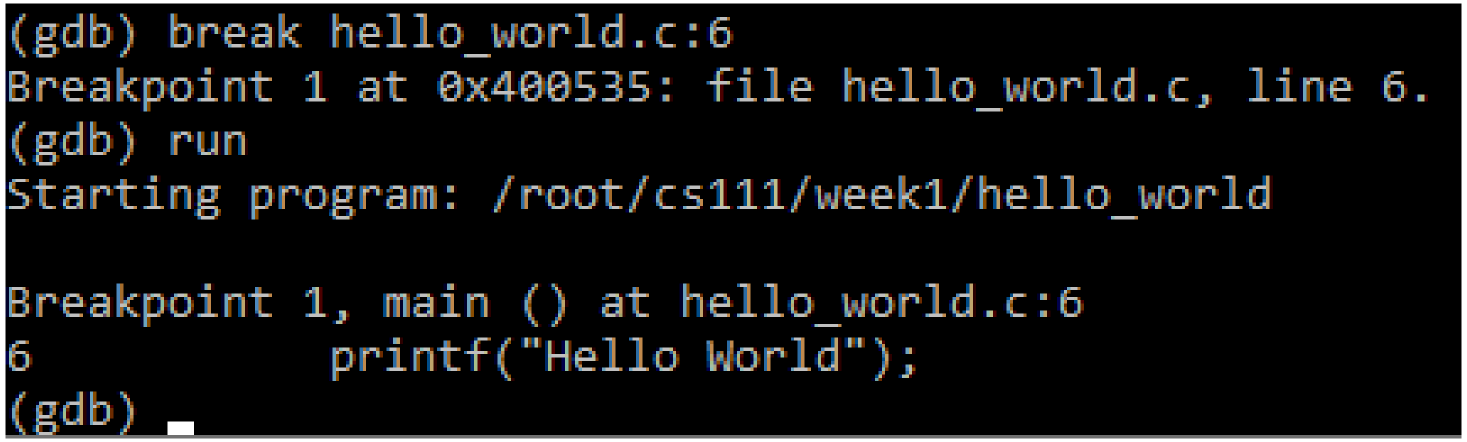


```
(gdb) run
Breakpoint 1 at 0x400535: file hello_world.c, line 6.
(gdb)
```

- You can pass arguments as you normally would from command line

gdb – GNU Debugger

- Setting breakpoints with break

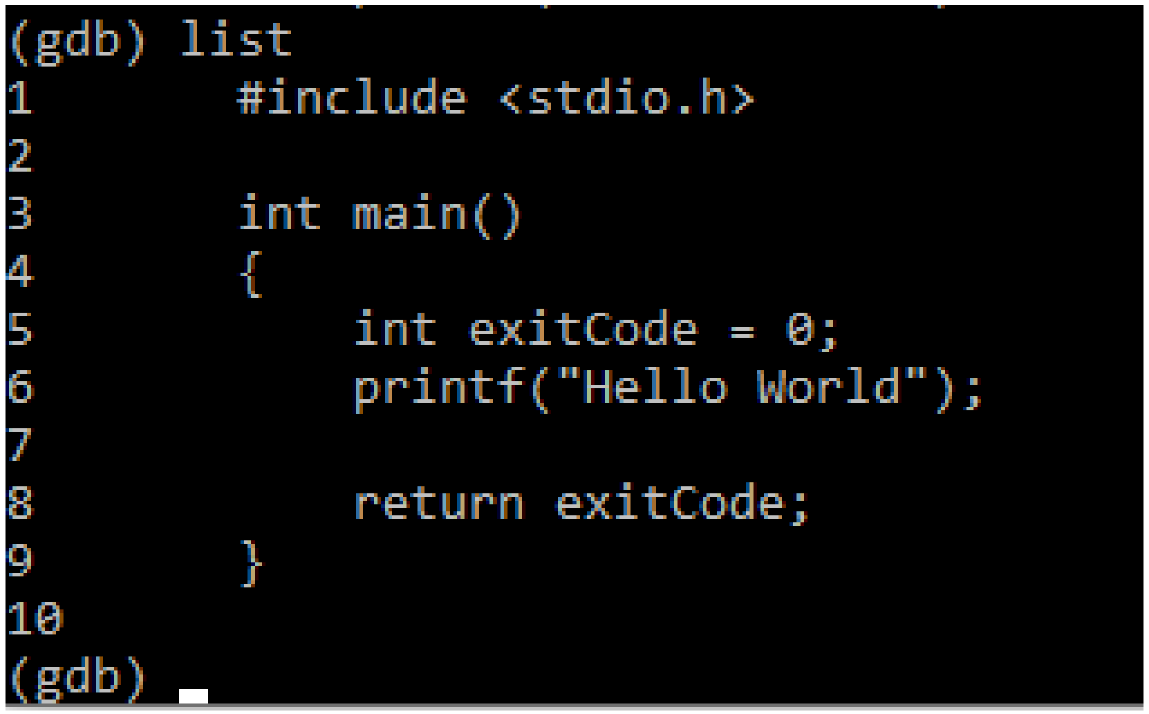


```
(gdb) break hello_world.c:6
Breakpoint 1 at 0x400535: file hello_world.c, line 6.
(gdb) run
Starting program: /root/cs111/week1/hello_world

Breakpoint 1, main () at hello_world.c:6
6           printf("Hello World");
(gdb) _
```

- Program pauses at the breakpoint

gdb – GNU Debugger

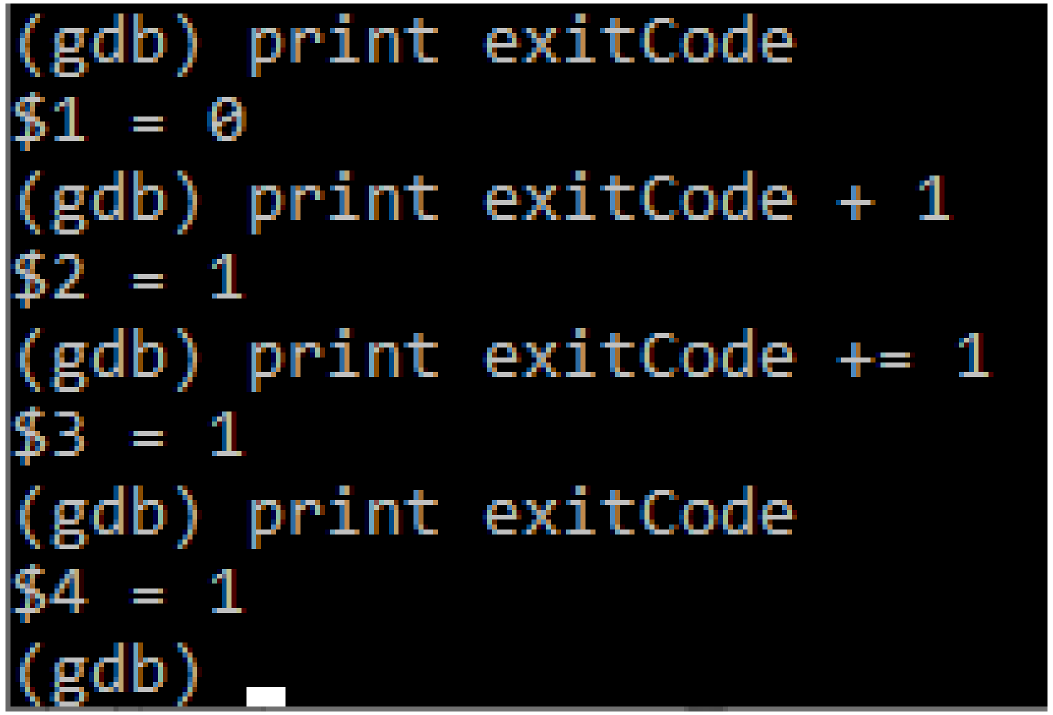
- `list` shows the code surrounding the current line of execution



```
(gdb) list
1       #include <stdio.h>
2
3       int main()
4       {
5           int exitCode = 0;
6           printf("Hello World");
7
8           return exitCode;
9       }
10
(gdb) _
```

gdb – GNU Debugger

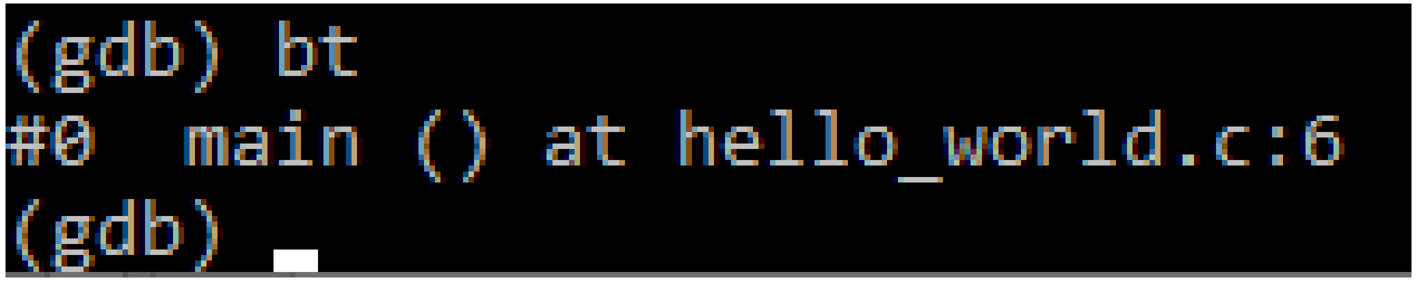
- `print` evaluates expression at current state
- Most commonly used to print the value of a variable



```
(gdb) print exitCode
$1 = 0
(gdb) print exitCode + 1
$2 = 1
(gdb) print exitCode += 1
$3 = 1
(gdb) print exitCode
$4 = 1
(gdb)
```

gdb – GNU Debugger

- `bt` shows the backtrace, or call stack of the current execution



```
(gdb) bt
#0  main () at hello_world.c:6
(gdb) 
```

- `#0` refers to main's frame, and we can move around the call stack using `frame n`, where `n` is the frame we want to move to
  - In this example, we only have one function, but if we have multiple, we can jump around to each function in the call stack

gdb – GNU Debugger

- Other common commands
  - `continue`: continue the execution until next breakpoint or the program exits
  - `step`: execute and step over current line
  - `clear <line number>`: clear breakpoint at the given line
  - `watch <variable>`: similar to `break`, pauses the program whenever a watched variable's value is modified
    - What if `<variable>` is a pointer?

Makefile review

- A Makefile is a collection of rules
- Each rule has a target, a list of dependencies, and a set of commands
- Use `make <target>` to execute a rule

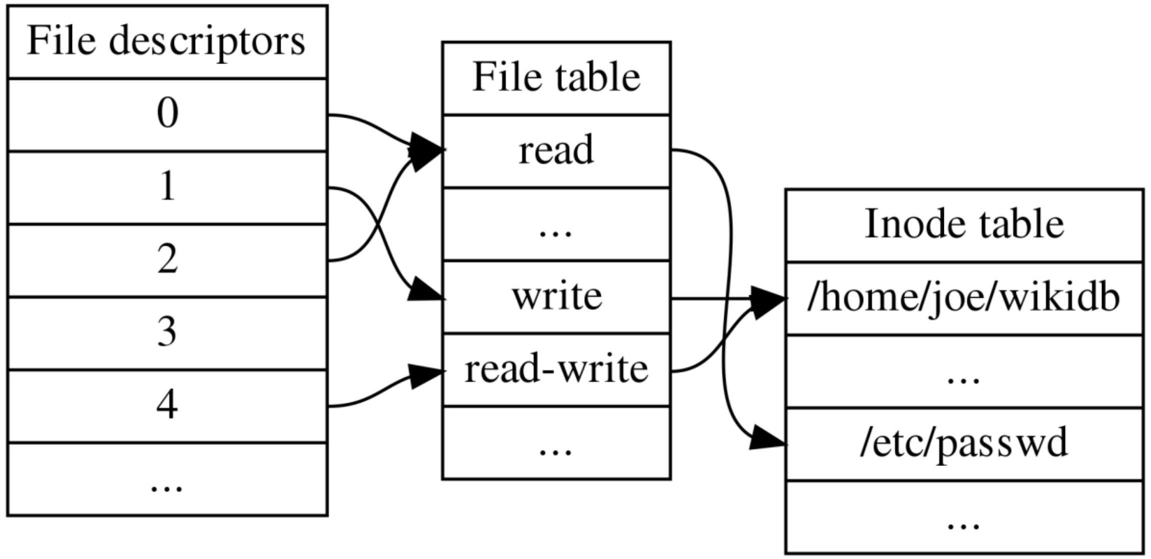
-
  ```
  <target>: <dependency_1>, <dependency_2>, …
  [tab] command_1
  [tab] command_2
  …
  ```
- If a target does not exist, or at least one of its dependencies is newer than target, then the commands will run.
- There can be zero, one, or many dependencies for a rule.

Makefile skeleton for project 0

```
lab0: lab0.c
      gcc …

clean:
      rm -f …

dist: lab0.c …
      tar …

check: lab0 …
      …
```

File Descriptor

- An integer
- Used by file access API (read, write)
- Normally every process has access to
  - 0: standard input (read only)
  - 1: standard output (write only)
  - 2: standard error (write ony)
- File descriptors are process-wide
- File table and inode table are system-wide

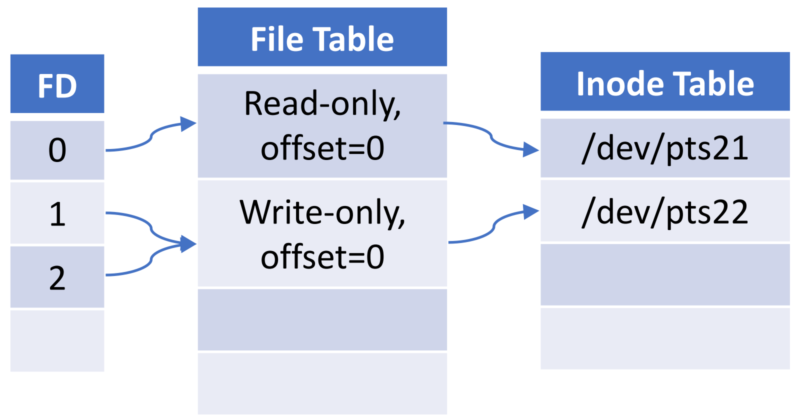


Basic FD Operations

- open(2): input path name, output fd
- close(2): closes file descriptor, so it can be reused
- dup(2): duplicates file descriptor to lowest available fd

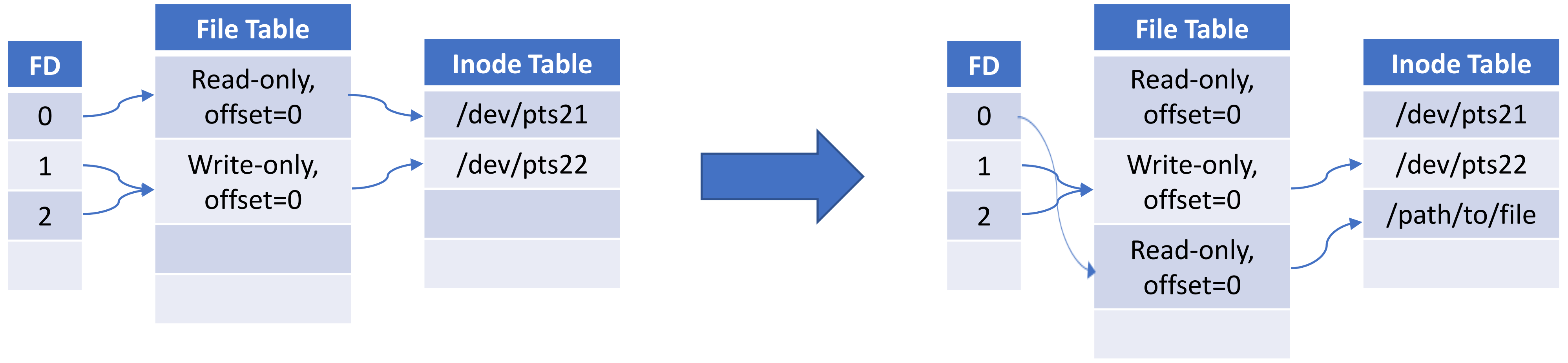
Example: Input Redirection

- Goal: instead of reading from stdin, read from another source



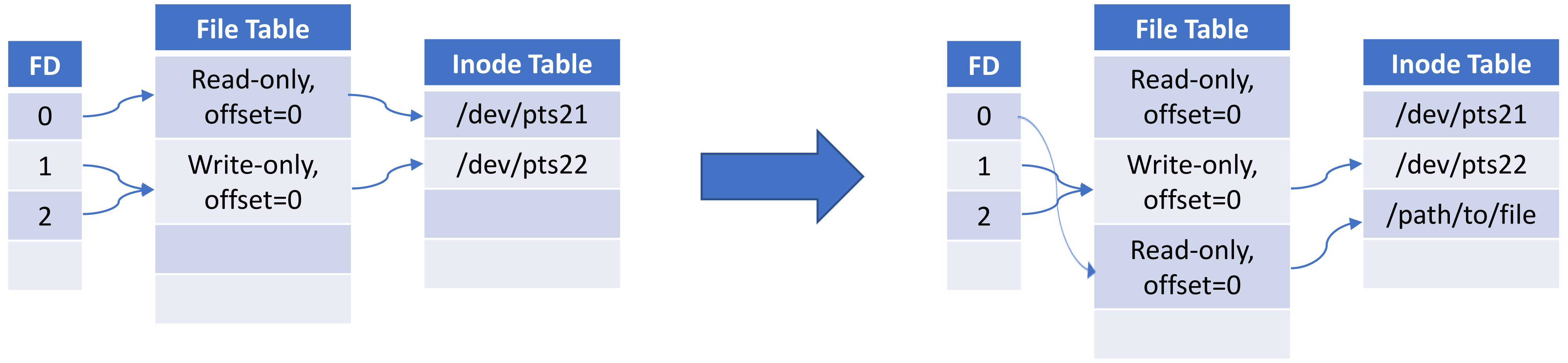
Example: Input Redirection

- Goal: instead of reading from stdin, read from another source



Example: Input Redirection

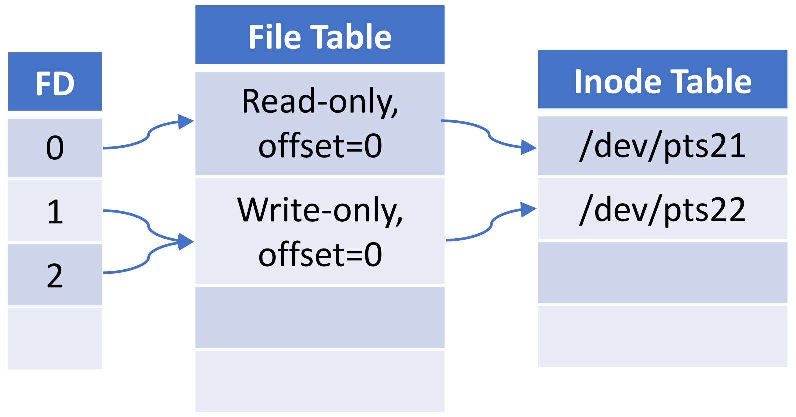
- Goal: instead of reading from stdin, read from another source



```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

Example: Input Redirection

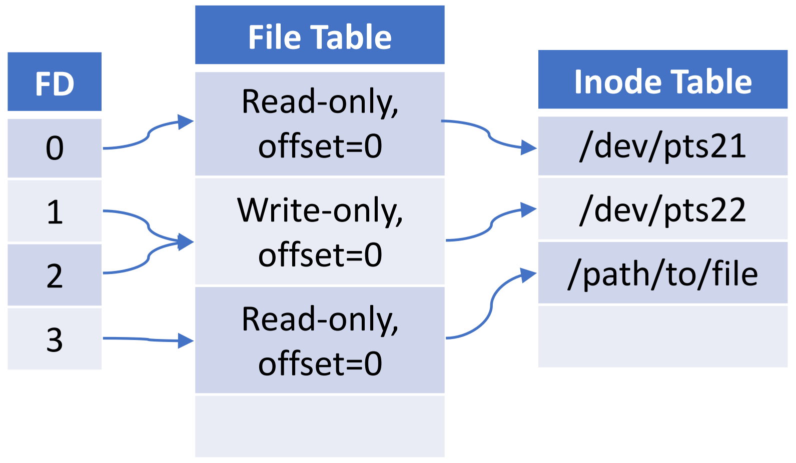
- Goal: instead of reading from stdin, read from another source



```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

Example: Input Redirection

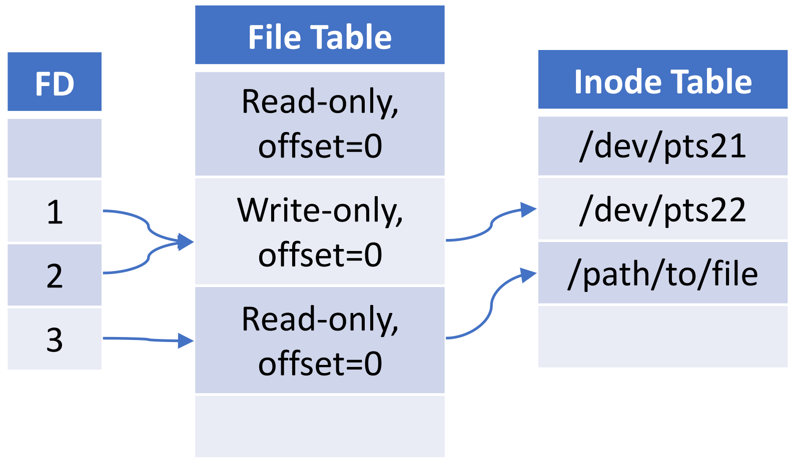
- Goal: instead of reading from stdin, read from another source



```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

Example: Input Redirection

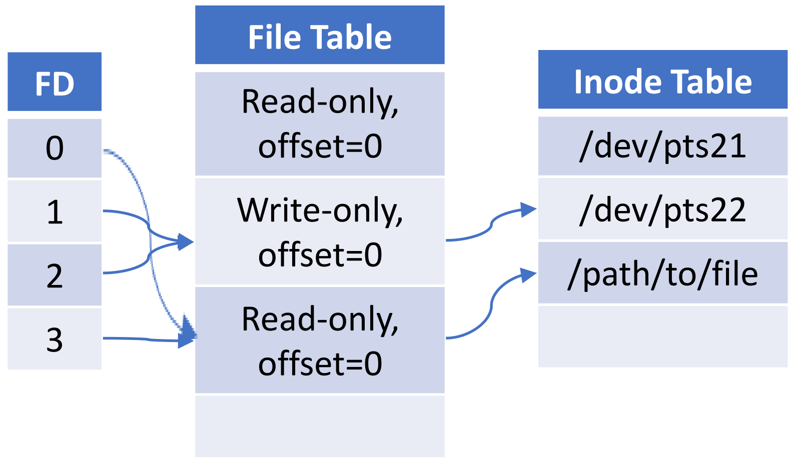
- Goal: instead of reading from stdin, read from another source



```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

Example: Input Redirection

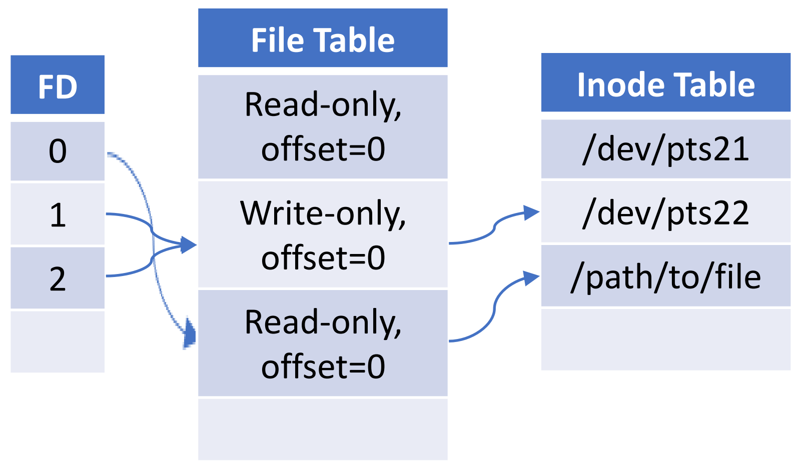
- Goal: instead of reading from stdin, read from another source



```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

Example: Input Redirection

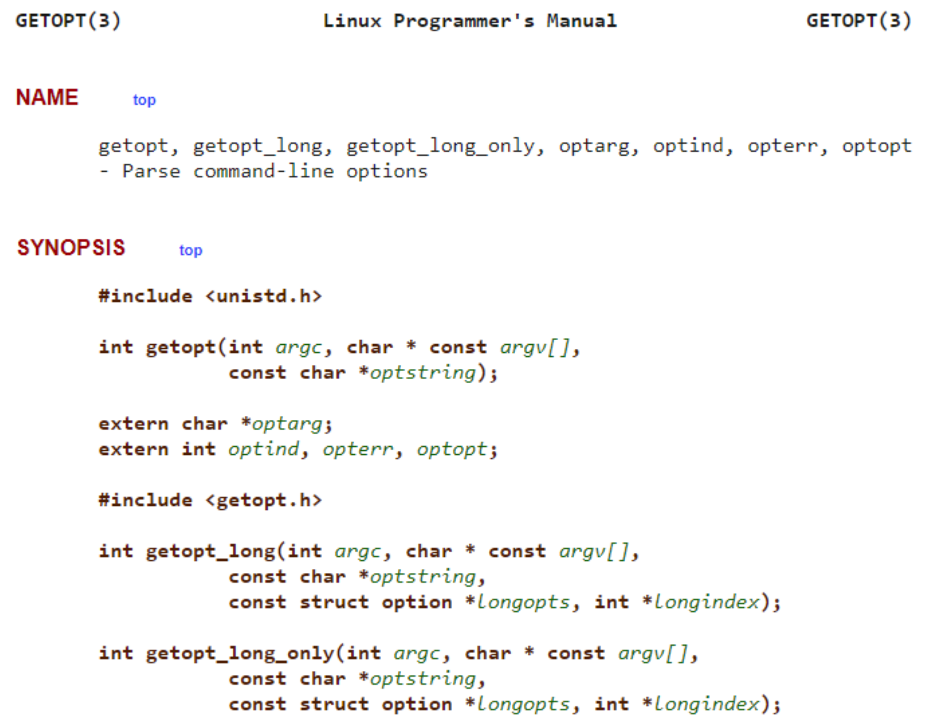
- Goal: instead of reading from stdin, read from another source



```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

Parsing command line arguments

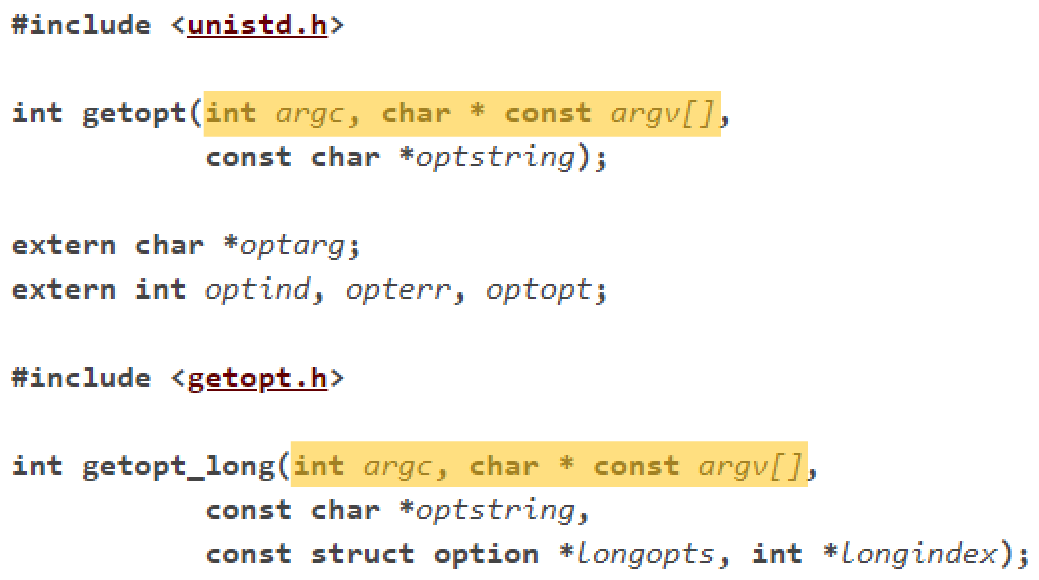
- `getopt_long(3)`
  - You will see numbers in parenthesis in manuals
  - (1): User commands
  - (2): System calls
  - (3): C library functions
  - …



```
GETOPT(3)                 Linux Programmer's Manual                GETOPT(3)

NAME       top

       getopt, getopt_long, getopt_long_only, optarg, optind, opterr, optopt
       - Parse command-line options

SYNOPSIS       top

       #include <unistd.h>

       int getopt(int argc, char * const argv[],
                  const char *optstring);

       extern char *optarg;
       extern int optind, opterr, optopt;

       #include <getopt.h>

       int getopt_long(int argc, char * const argv[],
                  const char *optstring,
                  const struct option *longopts, int *longindex);

       int getopt_long_only(int argc, char * const argv[],
                  const char *optstring,
                  const struct option *longopts, int *longindex);
```

getopt_long(3)

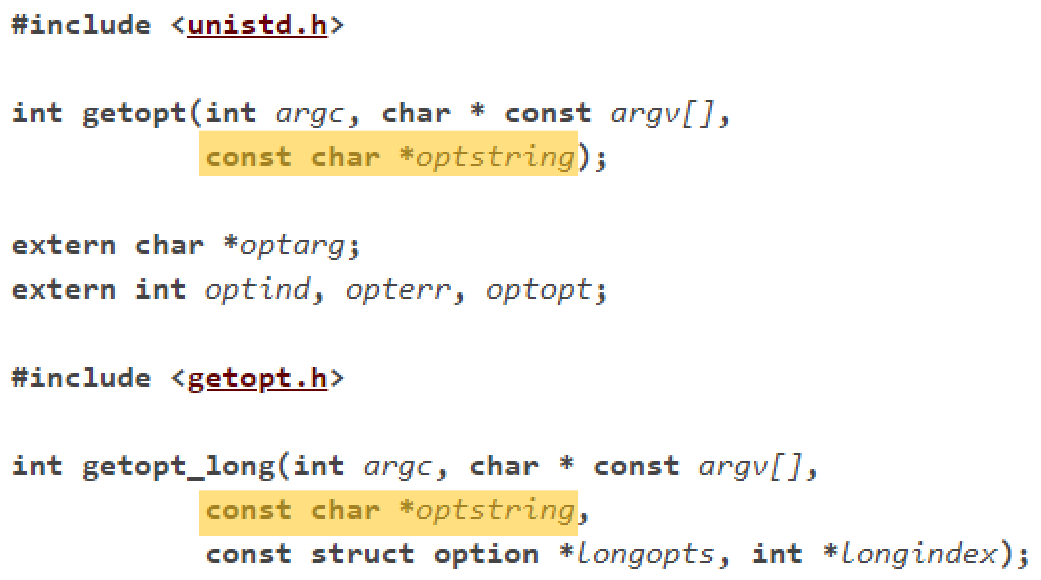
- `int argc, char * const argv[]`
  - The same parameters that are fed into main()
  - `argc` gives the argument count
  - `argv` gives the array of arguments



```
#include <unistd.h>

int getopt(int argc, char * const argv[],
           const char *optstring);

extern char *optarg;
extern int optind, opterr, optopt;

#include <getopt.h>

int getopt_long(int argc, char * const argv[],
           const char *optstring,
           const struct option *longopts, int *longindex);
```

getopt_long(3)

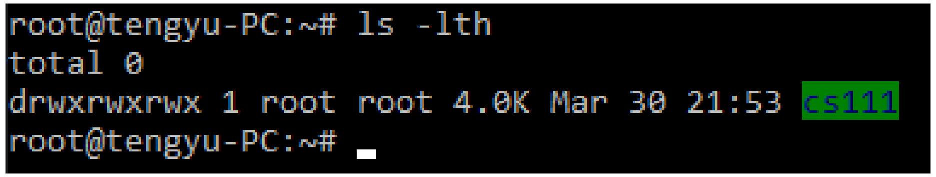
- `const char *optstring`
  - A string describes expected <u>short</u> options
  - Short options: options whose names are single characters



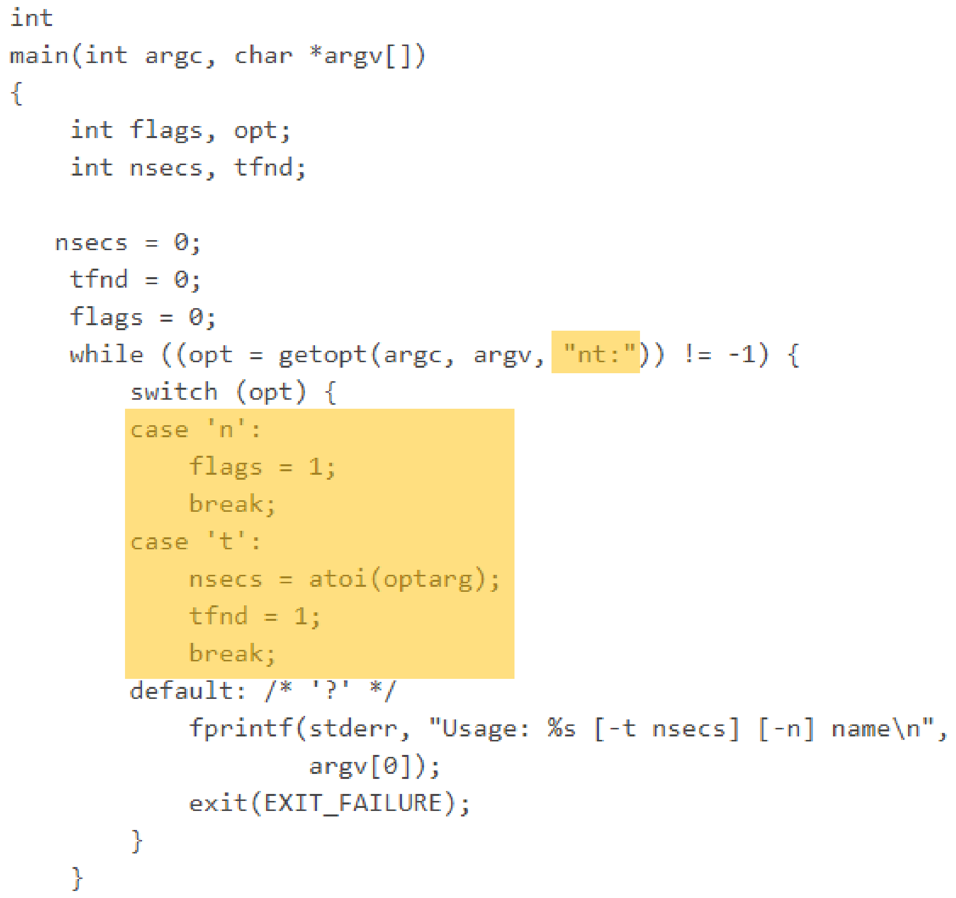
```
#include <unistd.h>

int getopt(int argc, char * const argv[],
           const char *optstring);

extern char *optarg;
extern int optind, opterr, optopt;

#include <getopt.h>

int getopt_long(int argc, char * const argv[],
           const char *optstring,
           const struct option *longopts, int *longindex);
```

getopt_long(3)

- const char *optstring
  - A string describes expected short options
  - Short options: options whose names are single characters



```
root@tengyu-PC:~# ls -lth
total 0
drwxrwxrwx 1 root root 4.0K Mar 30 21:53 cs111
root@tengyu-PC:~# _
```



```
int
main(int argc, char *argv[])
{
    int flags, opt;
    int nsecs, tfnd;

   nsecs = 0;
    tfnd = 0;
    flags = 0;
    while ((opt = getopt(argc, argv, "nt:")) != -1) {
        switch (opt) {
        case 'n':
            flags = 1;
            break;
        case 't':
            nsecs = atoi(optarg);
            tfnd = 1;
            break;
        default: /* '?' */
            fprintf(stderr, "Usage: %s [-t nsecs] [-n] name\n",
                    argv[0]);
            exit(EXIT_FAILURE);
        }
    }
```

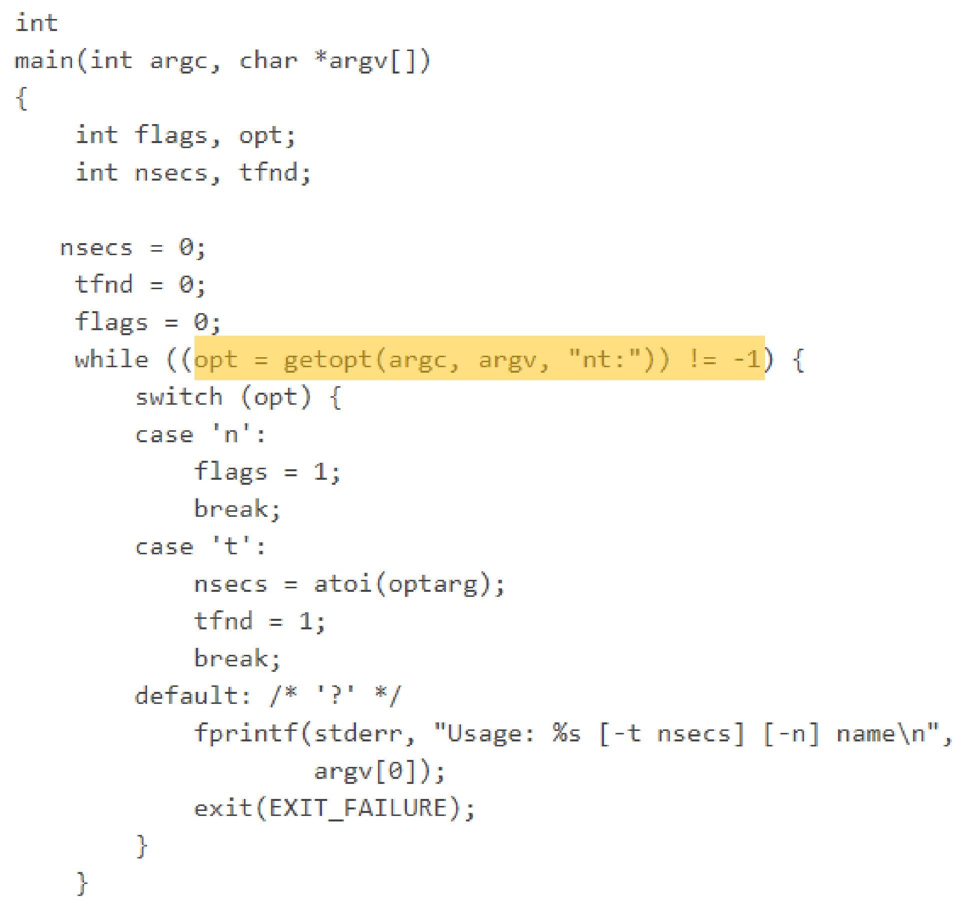
getopt_long(3)

- `const char *optstring`
  - A string describes expected short options
  - Short options: options whose names are single characters
  - Each time `getopt` is called, it parses and returns the name of the next argument
    - Current argument index can be accessed at the external variable `optind`
    - You can rewind parsing (although I don’t see a reason to do so) by modifying `optind`
    - Hence `getopt` is usually called in a while loop

```
int
main(int argc, char *argv[])
{
    int flags, opt;
    int nsecs, tfnd;

   nsecs = 0;
    tfnd = 0;
    flags = 0;
    while ((opt = getopt(argc, argv, "nt:")) != -1) {
        switch (opt) {
        case 'n':
            flags = 1;
            break;
        case 't':
            nsecs = atoi(optarg);
            tfnd = 1;
            break;
        default: /* '?' */
            fprintf(stderr, "Usage: %s [-t nsecs] [-n] name\n",
                    argv[0]);
            exit(EXIT_FAILURE);
        }
    }
```

getopt_long(3)

- `const char *optstring`
  - A string describes expected short options
  - Short options: options whose names are single characters
  - Each time `getopt` is called, it parses and returns the name of the next argument
    - When all arguments are parsed, `getopt` returns -1



```
int
main(int argc, char *argv[])
{
    int flags, opt;
    int nsecs, tfnd;

   nsecs = 0;
    tfnd = 0;
    flags = 0;
    while ((opt = getopt(argc, argv, "nt:")) != -1) {
        switch (opt) {
        case 'n':
            flags = 1;
            break;
        case 't':
            nsecs = atoi(optarg);
            tfnd = 1;
            break;
        default: /* '?' */
            fprintf(stderr, "Usage: %s [-t nsecs] [-n] name\n",
                    argv[0]);
            exit(EXIT_FAILURE);
        }
    }
```

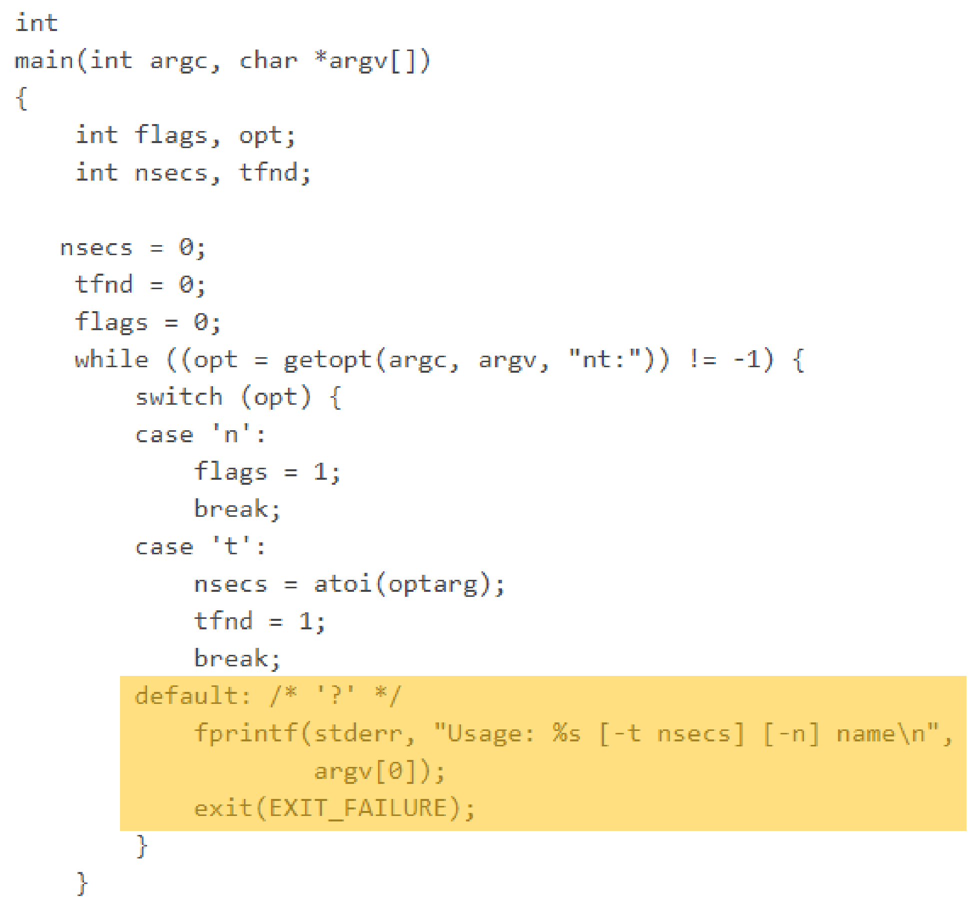
getopt_long(3)

- `const char *optstring`
  - A string describes expected <u>short</u> options
  - Short options: options whose names are single characters
  - If an option expects an argument, append `":"` after the character
    - The argument can be accessed at the external variable `optarg`

```
int
main(int argc, char *argv[])
{
    int flags, opt;
    int nsecs, tfnd;

    nsecs = 0;
    tfnd = 0;
    flags = 0;
    while ((opt = getopt(argc, argv, "nt:")) != -1) {
        switch (opt) {
        case 'n':
            flags = 1;
            break;
        case 't':
            nsecs = atoi(optarg);
            tfnd = 1;
            break;
        default: /* '?' */
            fprintf(stderr, "Usage: %s [-t nsecs] [-n] name\n",
                    argv[0]);
            exit(EXIT_FAILURE);
        }
    }
```

getopt_long(3)

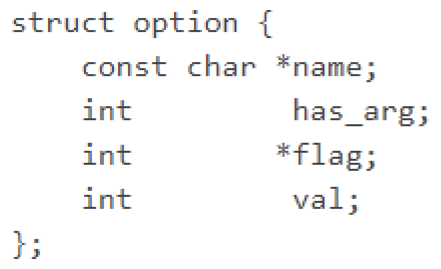
- const char *optstring
  - A string describes expected short options
  - Short options: options whose names are single characters
  - If parsing fails (for example, seeing an unknown argument), getopt returns '?'



```
int
main(int argc, char *argv[])
{
    int flags, opt;
    int nsecs, tfnd;

   nsecs = 0;
    tfnd = 0;
    flags = 0;
    while ((opt = getopt(argc, argv, "nt:")) != -1) {
        switch (opt) {
        case 'n':
            flags = 1;
            break;
        case 't':
            nsecs = atoi(optarg);
            tfnd = 1;
            break;
        default: /* '?' */
            fprintf(stderr, "Usage: %s [-t nsecs] [-n] name\n",
                    argv[0]);
            exit(EXIT_FAILURE);
        }
    }
```

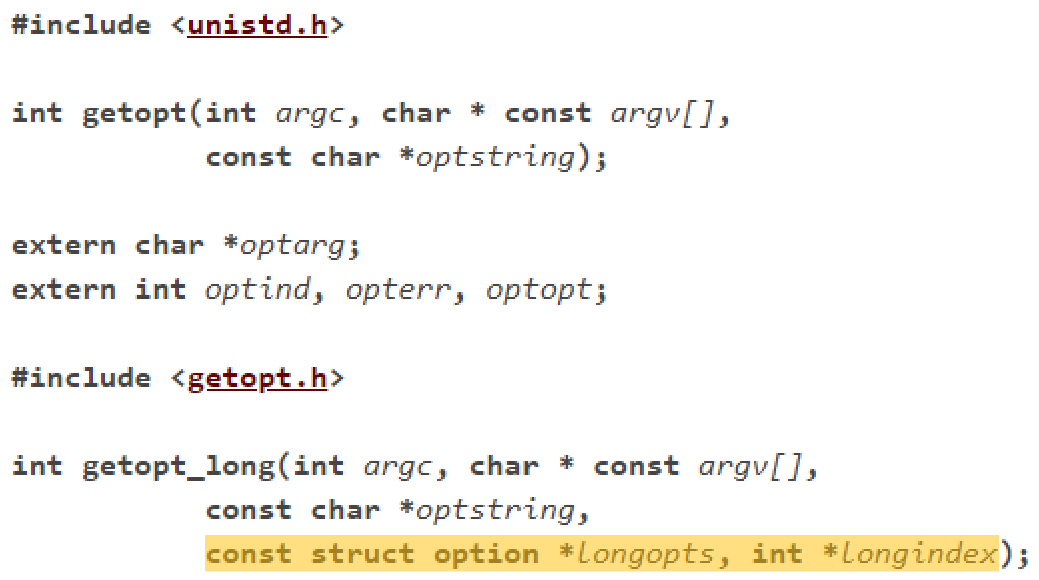
getopt_long(3)

- `const struct option *longopts, int *longindex`
  - `longopts` is an array of `struct option` which describes the expected options



```
struct option {
    const char *name;
    int         has_arg;
    int        *flag;
    int         val;
};
```

  - `longindex` stores the index of current option



```
#include <unistd.h>

int getopt(int argc, char * const argv[],
           const char *optstring);

extern char *optarg;
extern int optind, opterr, optopt;

#include <getopt.h>

int getopt_long(int argc, char * const argv[],
           const char *optstring,
           const struct option *longopts, int *longindex);
```

getopt_long(3)

```
struct option {
    const char *name;
    int         has_arg;
    int        *flag;
    int         val;
};
```

- `const struct option *longopts, int *longindex`
  - `longopts` is an array of `struct option` which describes the expected options
  - `longindex` stores the index of current option in `longopts`
  - If `longopts` is parsed, `getopt` returns `val`
  - `longopts` ends in `{0,0,0,0}`, similar to NULL

```
while(1) {
    int option_index = 0;
    static struct option long_options[] = {
        {"add",     required_argument, 0,  0 },
        {"append",  no_argument,       0,  0 },
        {"delete",  required_argument, 0,  0 },
        {"verbose", no_argument,       0,  0 },
        {"create",  required_argument, 0, 'c'},
        {"file",    required_argument, 0,  0 },
        {0,         0,                 0,  0 }};
    c = getopt_long(argc, argv, "ab:0", longoptions, &option_index);
    if (c == -1) break;
    switch (c) {
        case 0:
            printf("option %s", long_options[option_index].name);
            if (optarg) printf(" with arg %s" optarg);
            printf("\n");
            break;
        case 'a':
            printf("option a\n");
            break;

        case 'b':
            printf("option b with value '%s'\n", optarg);
            break;
        case '?’:
            break;
        default:
            printf("?? getopt returned character code 0%o ??\n", c);
        }
    }
```

getopt_long(3)

- What happens when we run with arguments
  - `./a.out --add 1 --append 2 -a -b 2`

Iteration	c	long_options[option_index].name	optarg

```
while(1) {
    int option_index = 0;
    static struct option long_options[] = {
        {"add",     required_argument, 0,  0 },
        {"append",  no_argument,       0,  0 },
        {"delete",  required_argument, 0,  0 },
        {"verbose", no_argument,       0,  0 },
        {"create",  required_argument, 0, 'c'},
        {"file",    required_argument, 0,  0 },
        {0,         0,                 0,  0 }};
    c = getopt_long(argc, argv, "ab:0", longoptions, &option_index);
    if (c == -1) break;
    switch (c) {
        case 0:
            printf("option %s", long_options[option_index].name);
            if (optarg) printf(" with arg %s" optarg);
            printf("\n");
            break;
        case 'a':
            printf("option a\n");
            break;

        case 'b':
            printf("option b with value '%s'\n", optarg);
            break;
        case '?':
            break;
        default:
            printf("?? getopt returned character code 0%o ??\n", c);
        }
    }
```

getopt_long(3)

- What happens when we run with arguments
  - `./a.out --add 1 --append 2 -a -b 2`

Iteration	c	long_options[option_index].name	optarg
0	0	"add"	1

```
while(1) {
    int option_index = 0;
    static struct option long_options[] = {
        {"add",     required_argument, 0,  0 },
        {"append",  no_argument,       0,  0 },
        {"delete",  required_argument, 0,  0 },
        {"verbose", no_argument,       0,  0 },
        {"create",  required_argument, 0, 'c'},
        {"file",    required_argument, 0,  0 },
        {0,         0,                 0,  0 }};
    c = getopt_long(argc, argv, "ab:0", longoptions, &option_index);
    if (c == -1) break;
    switch (c) {
        case 0:
            printf("option %s", long_options[option_index].name);
            if (optarg) printf(" with arg %s" optarg);
            printf("\n");
            break;
        case 'a':
            printf("option a\n");
            break;

        case 'b':
            printf("option b with value '%s'\n", optarg);
            break;
        case '?':
            break;
        default:
            printf("?? getopt returned character code 0%o ??\n", c);
        }
    }
```

getopt_long(3)

- What happens when we run with arguments
  - `./a.out --add 1 --append 2 -a -b 2`

Iteration	c	long_options[option_index].name	optarg
0	0	"add"	1
1	0	"append"	2

```
while(1) {
    int option_index = 0;
    static struct option long_options[] = {
        {"add",     required_argument, 0,  0 },
        {"append",  no_argument,       0,  0 },
        {"delete",  required_argument, 0,  0 },
        {"verbose", no_argument,       0,  0 },
        {"create",  required_argument, 0, 'c'},
        {"file",    required_argument, 0,  0 },
        {0,         0,                 0,  0 }};
    c = getopt_long(argc, argv, "ab:0", longoptions, &option_index);
    if (c == -1) break;
    switch (c) {
        case 0:
            printf("option %s", long_options[option_index].name);
            if (optarg) printf(" with arg %s" optarg);
            printf("\n");
            break;
        case 'a':
            printf("option a\n");
            break;

        case 'b':
            printf("option b with value '%s'\n", optarg);
            break;
        case '?’:
            break;
        default:
            printf("?? getopt returned character code 0%o ??\n", c);
        }
    }
```

getopt_long(3)

- What happens when we run with arguments
  - `./a.out --add 1 --append 2 -a -b 2`

Iteration	c	long_options[option_index].name	optarg
0	0	"add"	1
1	0	"append"	2
2	'a'	/	/

```
while(1) {
    int option_index = 0;
    static struct option long_options[] = {
        {"add",     required_argument, 0,  0 },
        {"append",  no_argument,       0,  0 },
        {"delete",  required_argument, 0,  0 },
        {"verbose", no_argument,       0,  0 },
        {"create",  required_argument, 0, 'c'},
        {"file",    required_argument, 0,  0 },
        {0,         0,                 0,  0 }};
    c = getopt_long(argc, argv, "ab:0", longoptions, &option_index);
    if (c == -1) break;
    switch (c) {
        case 0:
            printf("option %s", long_options[option_index].name);
            if (optarg) printf(" with arg %s" optarg);
            printf("\n");
            break;
        case 'a':
            printf("option a\n");
            break;

        case 'b':
            printf("option b with value '%s'\n", optarg);
            break;
        case '?’:
            break;
        default:
            printf("?? getopt returned character code 0%o ??\n", c);
        }
    }
```

getopt_long(3)

- What happens when we run with arguments
  - `./a.out --add 1 --append 2 -a -b 2`

Iteration	c	long_options[option_index].name	optarg
0	0	"add"	1
1	0	"append"	2
2	'a'	/	/
3	'b'	/	2

```
while(1) {
    int option_index = 0;
    static struct option long_options[] = {
        {"add",     required_argument, 0,  0 },
        {"append",  no_argument,       0,  0 },
        {"delete",  required_argument, 0,  0 },
        {"verbose", no_argument,       0,  0 },
        {"create",  required_argument, 0, 'c'},
        {"file",    required_argument, 0,  0 },
        {0,         0,                 0,  0 }};
    c = getopt_long(argc, argv, "ab:0", longoptions, &option_index);
    if (c == -1) break;
    switch (c) {
        case 0:
            printf("option %s", long_options[option_index].name);
            if (optarg) printf(" with arg %s" optarg);
            printf("\n");
            break;
        case 'a':
            printf("option a\n");
            break;

        case 'b':
            printf("option b with value '%s'\n", optarg);
            break;
        case '?':
            break;
        default:
            printf("?? getopt returned character code 0%o ??\n", c);
        }
    }
```

getopt_long(3)

- What happens when we run with arguments
  - `./a.out --add 1 --append 2 -a -b 2`

Iteration	c	long_options[option_index].name	optarg
0	0	"add"	1
1	0	"append"	/
2	'a'	/	/
3	'b'	/	2
4	-1	/	/

```
while(1) {
    int option_index = 0;
    static struct option long_options[] = {
        {"add",     required_argument, 0,  0 },
        {"append",  no_argument,       0,  0 },
        {"delete",  required_argument, 0,  0 },
        {"verbose", no_argument,       0,  0 },
        {"create",  required_argument, 0, 'c'},
        {"file",    required_argument, 0,  0 },
        {0,         0,                 0,  0 }};
    c = getopt_long(argc, argv, "ab:0", longoptions, &option_index);
    if (c == -1) break;
    switch (c) {
        case 0:
            printf("option %s", long_options[option_index].name);
            if (optarg) printf(" with arg %s" optarg);
            printf("\n");
            break;
        case 'a':
            printf("option a\n");
            break;

        case 'b':
            printf("option b with value '%s'\n", optarg);
            break;
        case '?':
            break;
        default:
            printf("?? getopt returned character code 0%o ??\n", c);
        }
    }
```

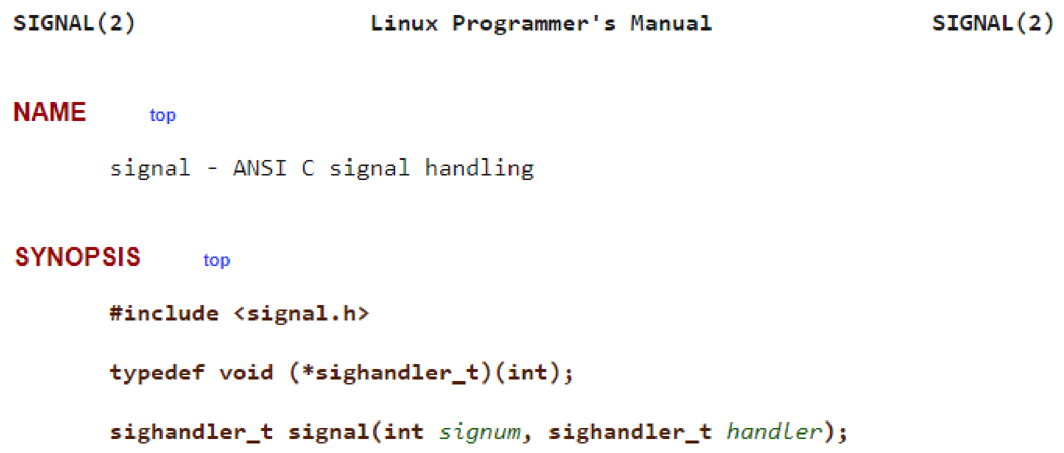
getopt_long(3)

- Best practice
  - First process all arguments and store results in variables
  - Then, operate according to these variables

```
while(1) {
    int option_index = 0;
    static struct option long_options[] = {
        {"add",     required_argument, 0,  0 },
        {"append",  no_argument,       0,  0 },
        {"delete",  required_argument, 0,  0 },
        {"verbose", no_argument,       0,  0 },
        {"create",  required_argument, 0, 'c'},
        {"file",    required_argument, 0,  0 },
        {0,         0,                 0,  0 }};
    c = getopt_long(argc, argv, "ab:0", longoptions, &option_index);
    if (c == -1) break;
    switch (c) {
        case 0:
            printf("option %s", long_options[option_index].name);
            if (optarg) printf(" with arg %s" optarg);
            printf("\n");
            break;
        case 'a':
            printf("option a\n");
            break;

        case 'b':
            printf("option b with value '%s'\n", optarg);
            break;
        case '?’:
            break;
        default:
            printf("?? getopt returned character code 0%o ??\n", c);
        }
    }
```

Registering a signal handler

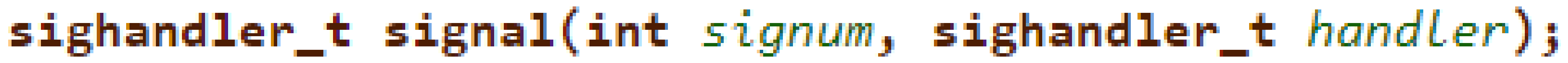
- signal(2)
  - (2): System call
  - Registers a signal handler
  - DOES NOT send a signal
  - To send a signal, call kill(2)
    - kill(2) does not kill a process
    - To do that, you need to send a kill signal (the code is 9)
    - Takeaway: Don't assume the functionality of a sys call by its name. Read the manual!



```
SIGNAL(2)              Linux Programmer's Manual              SIGNAL(2)

NAME      top

       signal - ANSI C signal handling

SYNOPSIS      top

       #include <signal.h>

       typedef void (*sighandler_t)(int);

       sighandler_t signal(int signum, sighandler_t handler);
```

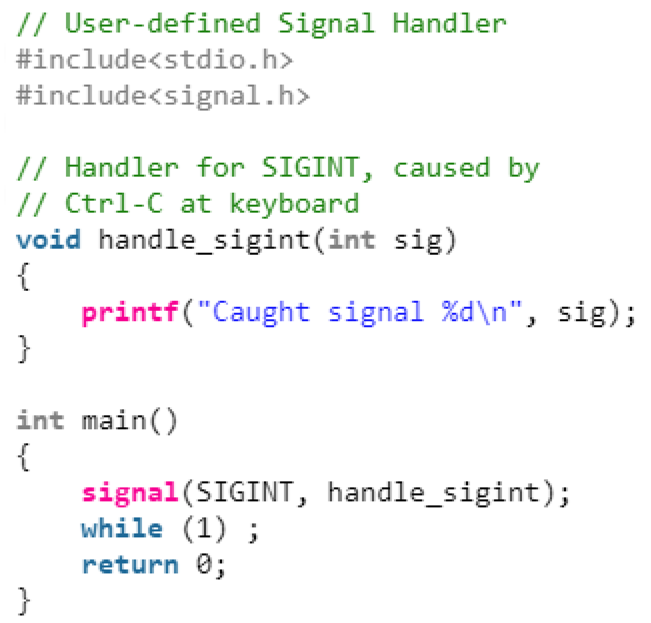
Signal Handler

- The OS has a default handler for each signal (specified by an int)
  - The handler is a subroutine to be executed if that signal is received
- As an application designer, you may want to override the default handler
- signal(2) links a signal (specified by its integer code) to a routine (that you likely wrote yourself)



```
sighandler_t signal(int signum, sighandler_t handler);
```

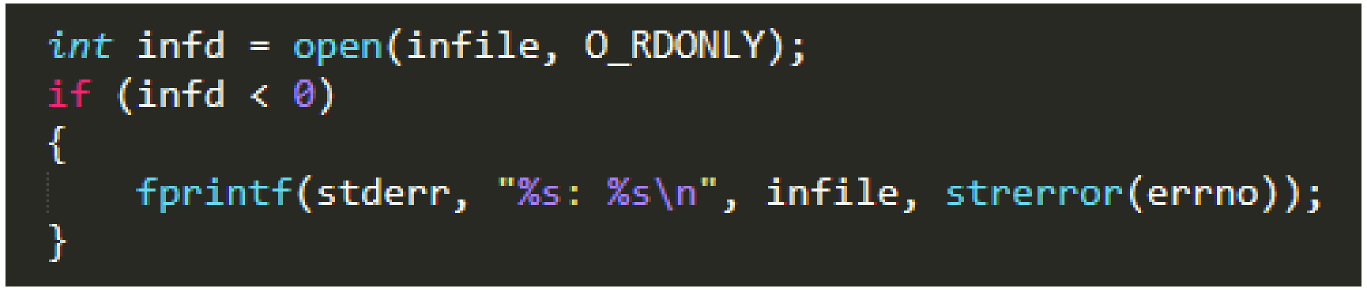
Signal Handler



```c
// User-defined Signal Handler
#include<stdio.h>
#include<signal.h>

// Handler for SIGINT, caused by
// Ctrl-C at keyboard
void handle_sigint(int sig)
{
    printf("Caught signal %d\n", sig);
}

int main()
{
    signal(SIGINT, handle_sigint);
    while (1) ;
    return 0;
}
```

Error Handling

- In some system calls, a global variable `errno` is set when an error occurs
- You may get a string description of the error by invoking
  - `char* strerror(int errno)`



```c
int infd = open(infile, O_RDONLY);
if (infd < 0)
{
    fprintf(stderr, "%s: %s\n", infile, strerror(errno));
}
```