COMPSCI 1XC3 - Computer Science Practice and Experience: Development Basics

Topic 7 - Benchmarking, Profiling and Advanced Bash Commands

Zheng Zheng

McMaster University

Winter 2023

Viewing Processes Using `htop`

Benchmarking using `time`

Profiling using `gprof`

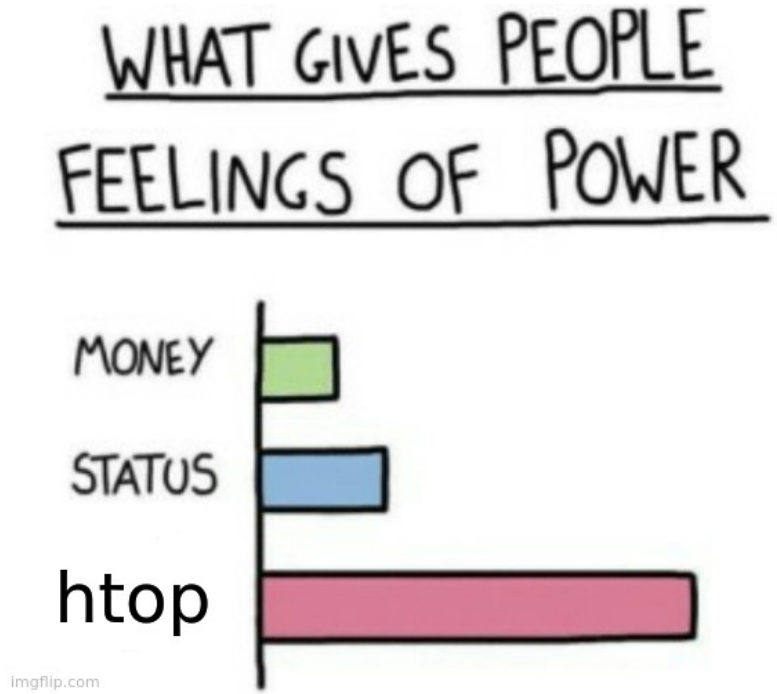
Source Code Annotation using `gcov`

Random Topic: File Permissions in Unix

Errata

Acknowledge

Viewing Processes Using `htop`



"It's like crtl+alt+del for Linux!"

"It's crtl+shift+esc now."

"... of course it is."

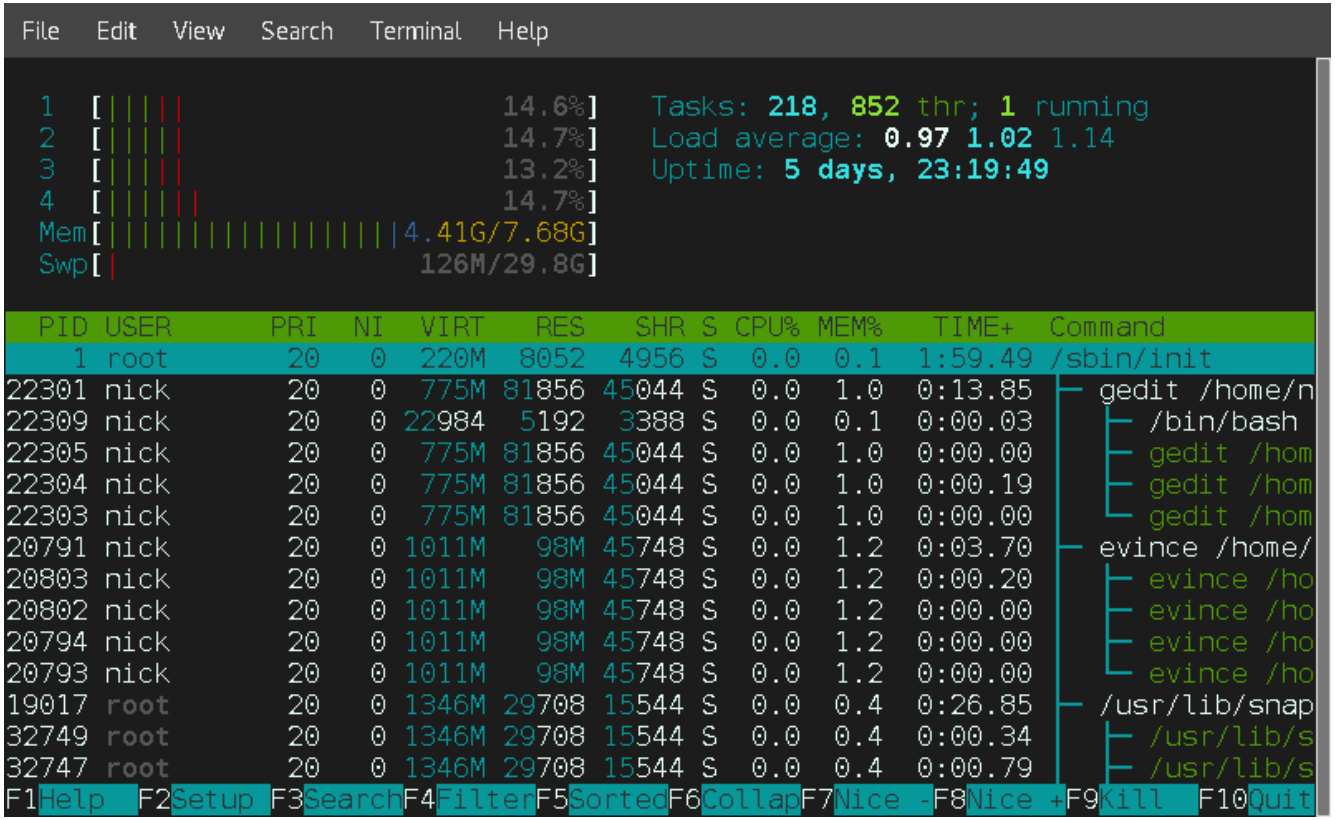
`htop` O' the Morning to Ya!

Modern operating systems manage hundreds, even thousands of concurrently executing programs.

- These are known as **processes**. Each has a unique **process identification number**, or *PID*.
- One program may spawn many separate processes, known as **threads**.
- Process management and scheduling is the subject of such courses as:
  - COMPSCI 3SD3 - Concurrent Systems
  - COMPSCI 4DC3 - Distributed Computing

We can view information about our computers' currently open processes by calling `htop`.

htop in Action



Useful Things to Know!

Among the information displayed is:

- Percent utilization of each CPU core in your computer.
- Memory usage out of total available memory.
- Swap space usage.
- The number of processes and threads currently running.
- How long it's been since you turned off your computer.

For each individual processes, you can easily see...

- Memory and CPU usage.
- Origin within the file system
- Which user is running it

You can use `htop` to *terminate* processes!

Benchmarking!

Benchmarking is the practice of measuring a product's features or performance, and comparing the results to other products of a similar nature.

- The term derives from the 19^{th} century. A benchmark was a channel cut into stone to mount measuring equipment.
- In Hardware, benchmarking involves executing batch operations that push the hardware to its limits.
- In Software, we are typically interested in *time* and *memory* performance.

With `htop`, we can see these parameters for active processes, but what about processes that take a fraction of a second? We need a stopwatch program!

The Nick of `time`

```
$ time <the command you wish to time>
```

After executing this command, and once your program terminates, you'll get a report such as the following:

```
real   2m4.352s
user   0m58.341s
sys  1m49.005s
```

- **real** - the difference between the time the process started and the time it stopped. This includes time the process was waiting for input or other processes.
- **user** - time spent in user-mode code (your code plus libraries). The amount of time the CPU used executing the process.
- **sys** - time spent in system-mode code (system calls and kernel stuff).

That's `time`!

You can also use the GNU version of `time` to track memory usage!

```
$ /usr/bin/time -v <the command you wish to time>
```

Bash has its own version of `time` which it defaults to, so we have to specify the full path of the GNU version.

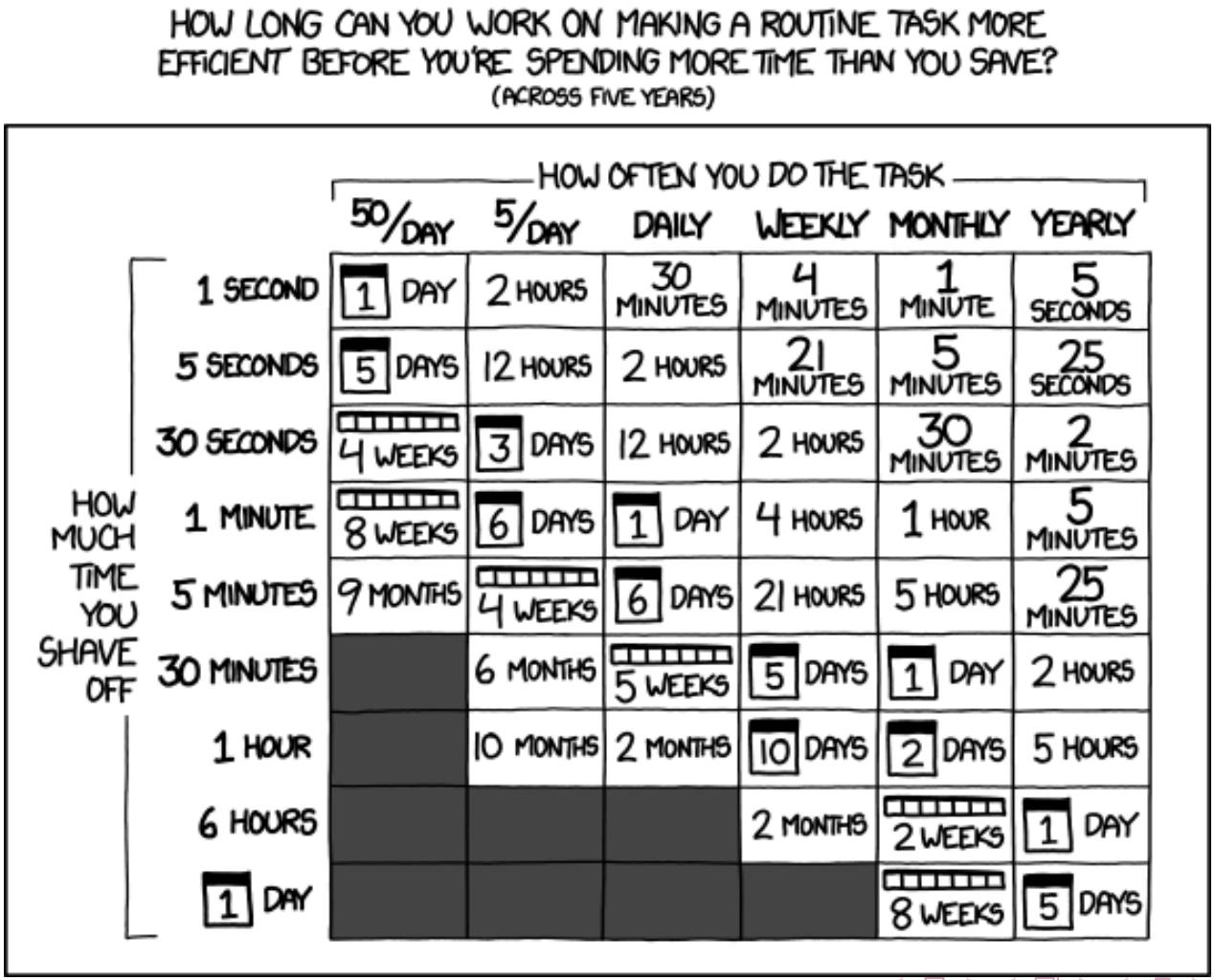
- The `-v` flag specifies "verbose" mode. This produces a lot of statistics on our command's execution:
  - Maximum resident set size - The maximum amount of physical memory your process was allocated by the operating system.
  - Exit status - the integer that your command returned. Non-zero statuses indicate "abnormal exit"
- It also tracks things like filesystem interactions and network usage.

While `time` gives you a quick, rough idea of how many resources your program is using, you'll often need more detail than this.

`time` in Action

```
Game Over, Yeah!
Solitaire Terminated!
        Command being timed: "./solitaire"
        User time (seconds): 0.00
        System time (seconds): 0.00
        Percent of CPU this job got: 0%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:16.00
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 1580
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 62
        Voluntary context switches: 6
        Involuntary context switches: 0
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

Optimization!



Introducing... `gprof`, the GNU profiling utility!

`htop` gives us a good idea of our system load, and `time` is good for fast estimations of runtime and memory usage, but these programs do not tell us anything about our program's *structure*!

- **Profiling** lets us break down our program's structure, and view statistics about the use of individual functions.
- This information can be used to help us optimize our programs.
  - Profiling doesn't tell you *how* to optimize.
  - Profiling does tell you *where* to optimize.

Autobots, Optimize!

Before we profile our code, let's learn how to use gcc's automatic optimization flags!

- `-O` → Standard Optimization.
- `-O2` → Level 2 Optimization.
- `-O3` → Level 3 Optimization.
- `-Os` → Size Optimization.
- `-Ofast` → Speed Optimization.
- `-Og` → Debuggability Optimization.

Notes about Optimization:

- These options will actually re-arrange your source code.
- You should be done debugging before using these.
- Optimization takes time! Especially for larger programs.

Compiling for gprof

gprof requires code to be inserted into your source files by gcc, so it can keep track of things.

1. Compile your program using the -pg option

```
$ gcc -pg example.c -o example
```

2. Execute your program as normal to collect the statistics in the file gmon.out

```
$ ./example
```

3. Run gprof in the following manner:

```
$ gprof example > profilerResults.out
```

4. View the generated file, which has lots of tasty information!

A Flat Profile

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
[...]
```

Interpretting the Flat Profile

The flat profile provides the following information for every function your program uses (yes, even libraries).

- **% time** → The percentage of total runtime the programe spent on each function.
- **cumulative seconds** → The number of seconds spent in this function plus all entries above it in the table
- **self seconds** → Number of seconds spent in this function alone.
- **calls** → Total number of times this function was called.
- **self ms/call** → Average time spent in this function per call.
- **total ms/call** → Averge time spent in this function *and descendents* per call.
- **name** → The name of the function

A Call Graph

```
index % time    self  children    called     name
                                               <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                0.00    0.05       1/1           start [1]
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1           report [3]
-----------------------------------------------
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1         report [3]
                0.00    0.03       8/8           timelocal [6]
                0.00    0.01       1/1           print [9]
                0.00    0.01       9/9           fgets [12]
                0.00    0.00      12/34          strncmp <cycle 1> [40]
                0.00    0.00       8/8           lookup [20]
                0.00    0.00       1/1           fopen [21]
                0.00    0.00       8/8           chewtime [24]
                0.00    0.00       8/16          skipspace [44]
-----------------------------------------------
[4]     59.8    0.01        0.02       8+472     <cycle 2 as a whole> [4]
                0.01        0.02     244+260         offtime <cycle 2> [7]
                0.00        0.00     236+1           tzset <cycle 2> [26]
-----------------------------------------------
```

The call graph gives you an idea of how many times each function is called by every other function in your program.

Interpretting the Call Graph

```
index % time    self  children    called     name
                                               <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                0.00    0.05       1/1           start [1]
[2]    100.0    0.00    0.05       1          main [2]
                0.00    0.05       1/1           report [3]
-----------------------------------------------
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1          report [3]
                0.00    0.03       8/8           timelocal [6]
                0.00    0.01       1/1           print [9]
                0.00    0.01       9/9           fgets [12]
                0.00    0.00      12/34          strncmp <cycle 1> [40]
                0.00    0.00       8/8           lookup [20]
                0.00    0.00       1/1           fopen [21]
                0.00    0.00       8/8           chewtime [24]
                0.00    0.00       8/16          skipspace [44]
-----------------------------------------------
[4]     59.8    0.01        0.02         8+472     <cycle 2 as a whole> [4]
                0.01        0.02       244+260         offtime <cycle 2> [7]
                0.00        0.00       236+1           tzset <cycle 2> [26]
-----------------------------------------------
```

Calling function

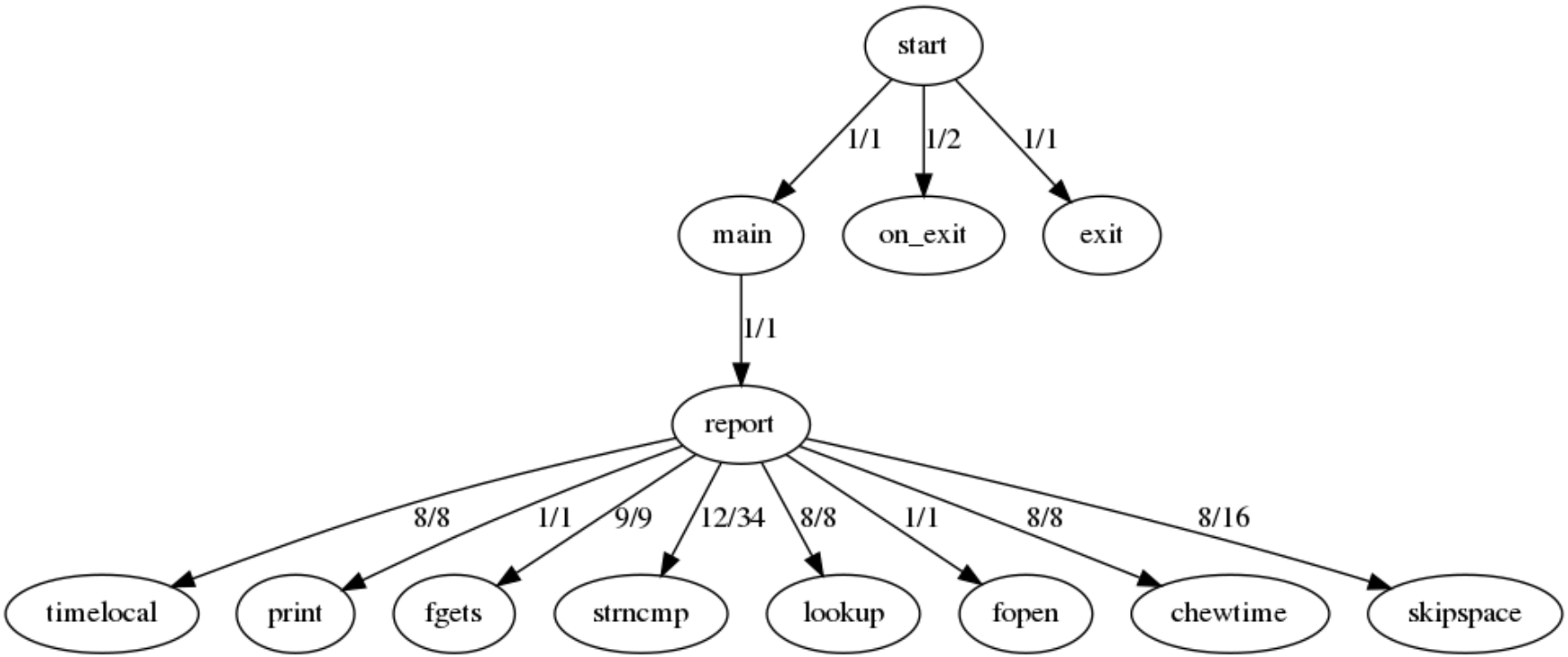
The function this entry pertains to

Functions called by the above function

8 Regular + 472 Recursive Calls

Call Graph Diagram

Taking the above information and arranging it into a call graph diagram yields the following:



Source Code Annotation using gcov

So, now we know how to profile code at the level of *functions*, but *we can go deeper!*

- gcov will annotate your source code itself with:
  - How often each line of code is executed
  - What lines of code are actually executed
  - How much computer time each section of code requires.
- gcov is fantastic to use in combination with a testing, because it will tell you whether your tests have full coverage of the source code!

In order for the results to be meaningful, however, we can't use compiler optimizations, since they change the source code!

Using gcov

To use gcov, we need once again to perform a special compilation:

```
$ gcc -fprofile-arcs -ftest-coverage example.c
```

Take the resulting executable file and execute it. This will produce a `*.gcda` file, which contains the profiling data. Then invoke:

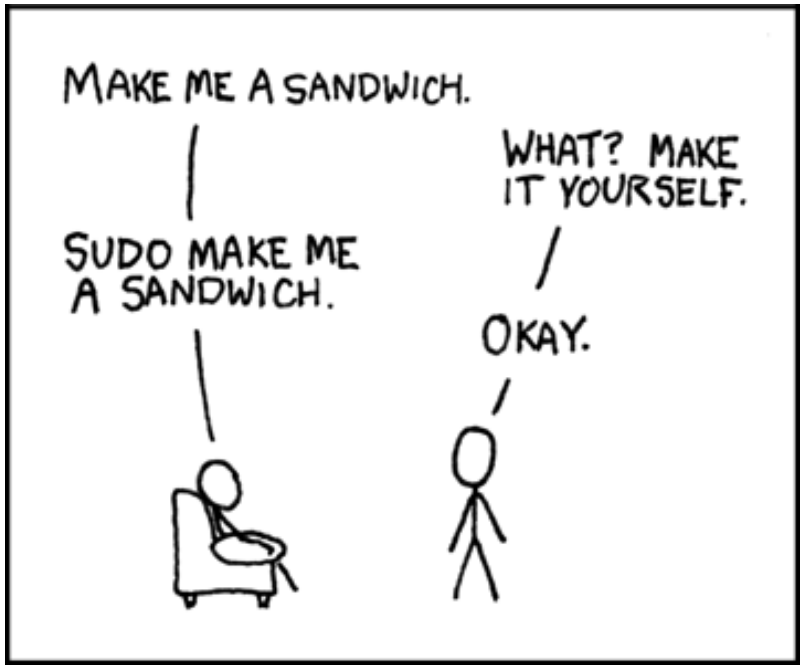
```
$ gcov example.c
```

The profiling results will be stored in `example.c.gcov`.

Interpretting the Output

```
[...]
// <Execution count> : <Line number> : <Source Code>
        5:   123:   int xcheck = x1;
        5:   124:   int ycheck = y1;
        5:   125:   int piecesInWay = 0;
        5:   126:   if (x1 > 7 || x1 < 0) {
        1:   127:     return false;
        4:   128:   } else if (y1 > 7 || y1 < 0) {
    #####:   129:     return false;
        4:   130:   } else if (x2 > 7 || x2 < 0) {
    #####:   131:     return false;
        4:   132:   } else if (y2 > 7 || y2 < 0) {
    #####:   133:     return false;
        -:   134:   }
// "#####" indicates the line was not executed, i.e.,
    not "covered"
[...]
```

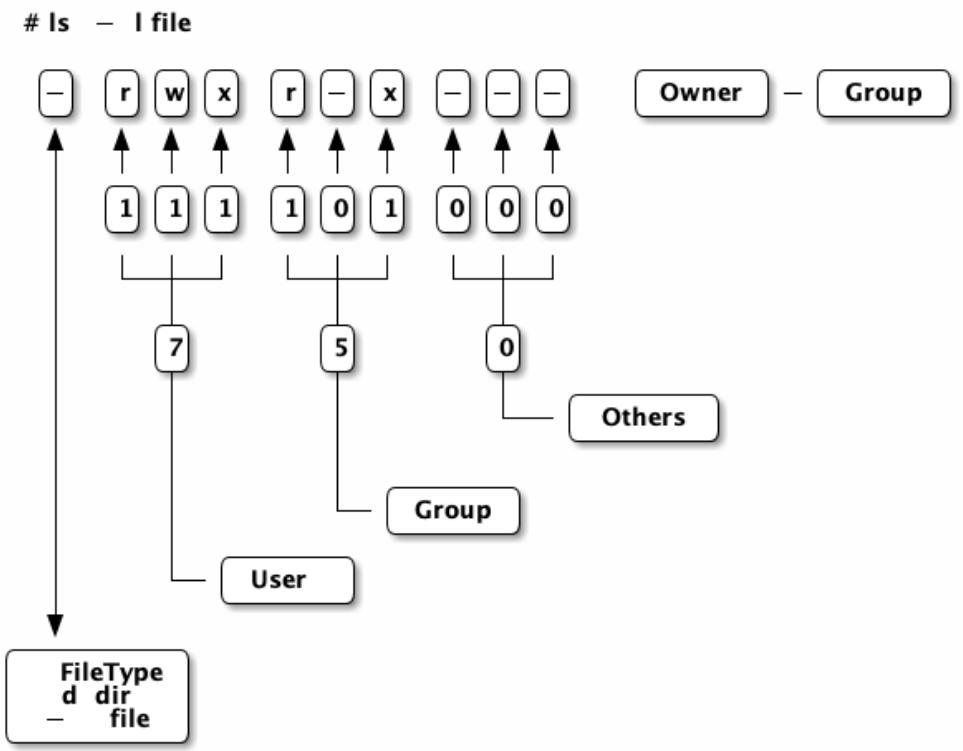
Unix File Permissions



"UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity."
– Dennis Ritchie

Unix File Permissions

Within the Unix file system, 10 bits are allocated per file to record file permissions, and are visible when using `ls -l`



Unix-like systems specify user permissions by:

- **User**
- **Group**
- **Others**

Specifying if a file is :

- **Readable**
- **Writeable**
- **Executable**

Filetype, User, Group and Others

- **File Type** only specifies whether an item is a file or a directory.
- **User** → Each file has an owner, these bit specify the permissions for the owner (this will be you most of the time).
- **Group** → Users may be collected into groups for ease of management.
  - Group permissions apply to the group you are a member of (if any).
  - Think of the difference between *student* and *faculty* access to something like Avenue or Mosaic.

item **Others** → Permissions for everyone not in the first two categories.

It's a bird! It's a Plane! It's Superuser!

If you're trying to do something in Unix/Linux, and you get `Permission denied`, that means you don't have the correct level of privilege for the operation you are trying to perform.

- There is only one way around this: invoking SUPERUSER!

```
$ sudo <command you weren't able to execute>
```

- In Linux, the superuser account (called "root") is omnipotent, with unrestricted access to:
  - commands, files, directories, and resources.
  - This also means the ability to install programs for all users, change system settings, and *compile that kernel!*
- granting and revoking permissions for other users.

The Group Scoop

By default, each user is already in a group containing just itself.

- This group's name is the user's name.
- You can create a group (assuming you have the authority to do so) as follows:

```
$ groupadd <group name>
```

You can add a user to a group with...

```
$ useradd -g <user name> <group name>
```

Modifying File Permissions via chmod

```
$ chmod u+x <file name>
    # add executable permissions to user
$ chmod o+rw <file name>
    # add read and write permission to others
$ chmod -w <file name>
    # remove write permissions from everyone
```

In general, it's

```
$ chmod <person(s)><grant / revoke><permissions> <
    filename>
```

- u → user
- g → group
- o → other

- \+ → grant
- \- → revoke

- r → readable
- w → writeable
- x → executable

Changing Ownership with `chown`

Every file belongs to *both* an owner *and* a group.

- The `chown` command changes file ownership.

```
$ chown 'owner:group' <filename>
```

- Change just the owner by not specifying a group:

```
$ chown 'owner' <filename>
```

- Change just the group by not not specifying an owner and leaving in the colon:

```
$ chown ':group' <filename>
```

IMPORTANT INSTRUCTIONS

`https://files.fosswire.com/2007/08/fwunixref.pdf` →
print this off and tape it up somewhere in your workspace.
Better yet, make your own from the slides in this course!

Acknowledge

The contents of these slides were liberally borrowed (with permission) from slides from the Summer 2021 offering of 1XC3 (by Dr. Nicholas Moore).