CIS 3207 - WEEK 3

Alex Russakoff

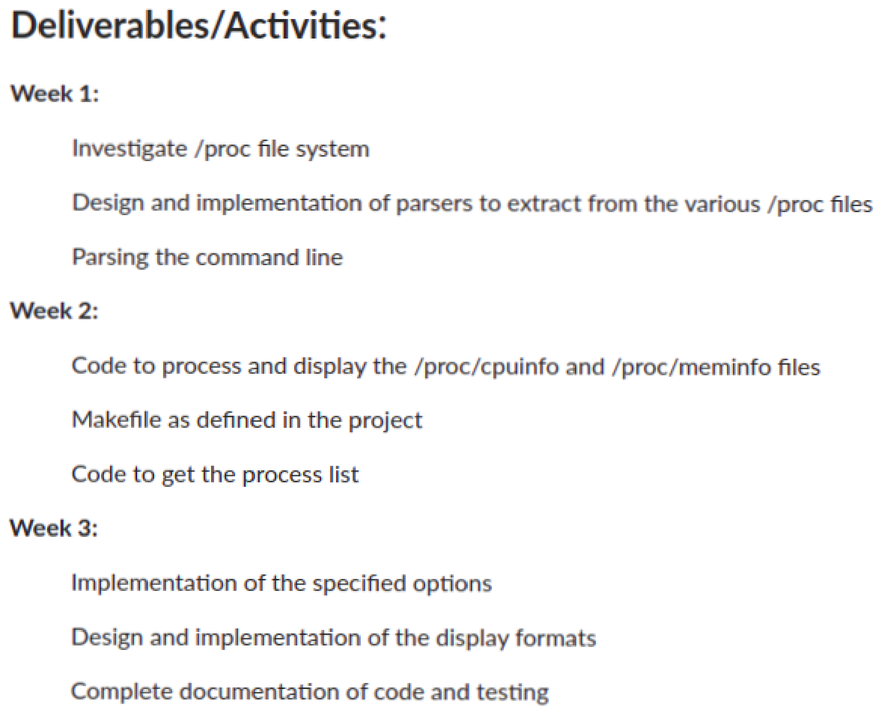
Important Information

- Project 0: Part B was due **today at 9 am**
    - If you have not submitted, please do so!
    - Remember that if you want to use a late day, please contact me
- DON’T FORGET TO DEMO!!
    - Last day to demo Part A: 9/14
    - Last day to demo Part B: 9/21
- Reminder that you will demo based off of your submission on Canvas
    - Make sure you are on a Linux or Posix compliant system for demos!

- MAKE SURE YOU ARE MAKING GITHUB COMMITS!

Project 1
The /PROC File System

Your First Deliverable: Due 9/14 at 9am



Deliverables/Activities:

Week 1:

Investigate /proc file system

Design and implementation of parsers to extract from the various /proc files

Parsing the command line

Week 2:

Code to process and display the /proc/cpuinfo and /proc/meminfo files

Makefile as defined in the project

Code to get the process list

Week 3:

Implementation of the specified options

Design and implementation of the display formats

Complete documentation of code and testing

Your First Deliverable: Due 9/14 at 9am

- You need to submit all your code for parsing command line arguments to your executing process (all .c and .h files)

- Please keep track of how you are designing your solution and testing it as you build it - you will have a personal README in the final submission, but you need to keep track of what you are doing now!
- Make sure that you are committing to your GitHub repository with quality commits
- Submit your deliverable on Canvas
    - You will not be demoing your deliverable to me, only the final submission

The proc file system!

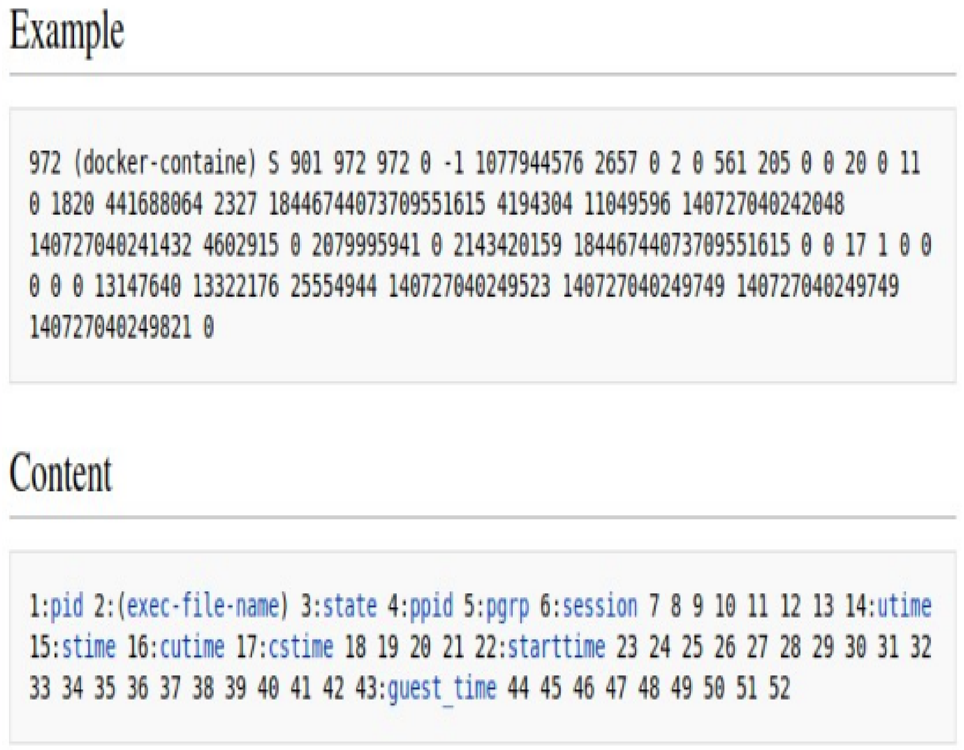
- What is the proc file system?
  - Virtual – ‘files’ have no size
  - Typically read only – Most ‘files’ cannot be changed
    - There are exceptions, but we don’t need to worry about this for the project.
  - Contains kernel information about:
    - System (Hardware and OS)
    - Processes
    - Many other things!

/proc/[pid] && /proc/self

- /proc/pid/ gives access to kernel information of a particular process
  - Generated on the fly by kernel
  - Important ‘files’ included in /proc/[pid]/:
    - /stat
    - /statm
    - /cmdline
  - Nearly everything you need will be in one of these folders
- /proc/self/ – symbolic link that redirects to current process’s /proc/[pid]/ folder

/proc/[pid]/stat

- Contains:
  - PID: Process ID
  - State
  - Utime: Total User Time
  - Stime: Total System Time
- Consider!!
  - Second entry surrounded by ()
  - All other entries are numbers
  - Parsing strategy must account for this!!!
    - Work backwards until ‘)’??
    - Work forward until #‘(‘ == #’)’



Example

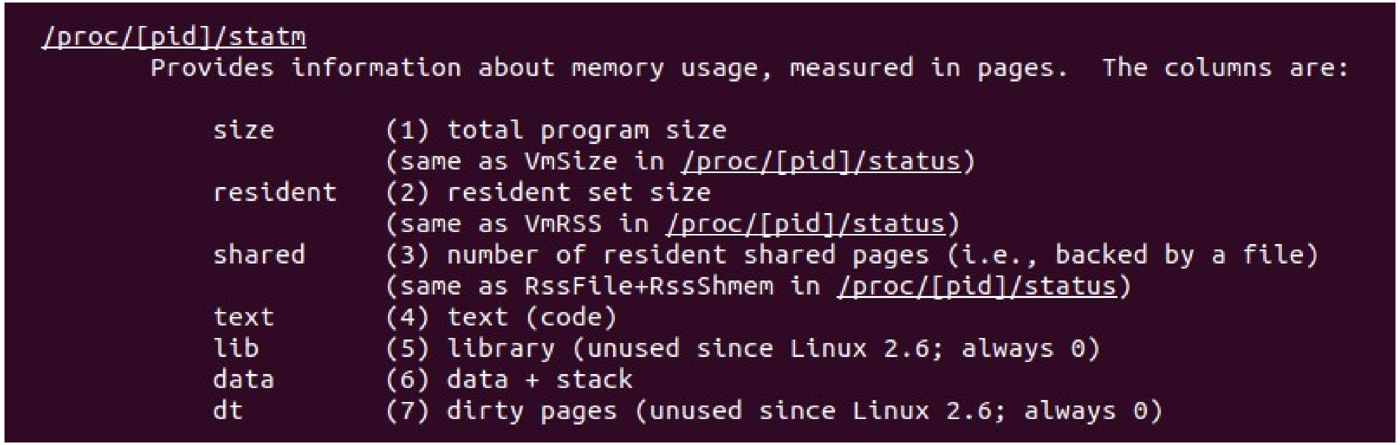
```
972 (docker-containe) S 901 972 972 0 -1 1077944576 2657 0 2 0 561 205 0 0 20 0 11
0 1820 441688064 2327 18446744073709551615 4194304 11049596 140727040242048
140727040241432 4602915 0 2079995941 0 2143420159 18446744073709551615 0 0 17 1 0 0
0 0 0 13147640 13322176 25554944 140727040249523 140727040249749 140727040249749
140727040249821 0
```

Content

```
1:pid 2:(exec-file-name) 3:state 4:ppid 5:pgrp 6:session 7 8 9 10 11 12 13 14:utime
15:stime 16:cutime 17:cstime 18 19 20 21 22:starttime 23 24 25 26 27 28 29 30 31 32
33 34 35 36 37 38 39 40 41 42 43:guest_time 44 45 46 47 48 49 50 51 52
```

/proc/[pid]/statm

- Contains memory information
- We only care about the first value
- Already in pages (lucky you!)
- Space seperated



```
/proc/[pid]/statm
       Provides information about memory usage, measured in pages.  The columns are:

           size       (1) total program size
                      (same as VmSize in /proc/[pid]/status)
           resident   (2) resident set size
                      (same as VmRSS in /proc/[pid]/status)
           shared     (3) number of resident shared pages (i.e., backed by a file)
                      (same as RssFile+RssShmem in /proc/[pid]/status)
           text       (4) text (code)
           lib        (5) library (unused since Linux 2.6; always 0)
           data       (6) data + stack
           dt         (7) dirty pages (unused since Linux 2.6; always 0)
```

/proc/[pid]/cmdline

- Contains NULL BYTE SEPERATED command line arguments
- Exactly the values of argv
- First delivery, START NOW!
- Consider:
  - Definition of a string in C (set of characters followed by a null byte)
  - Consider the following where /0 is a nullbyte
    - “something/0somethingelse/0”
    - I argue the above is indistinguishable from 2 strings in memory if stored contiguously
    - How can we get a pointer to string 2? (hint, start with strlen)

Useful Functions

- strtok_r(char*str,char*delim,char**saveptr)
  - Just like strtok, but you pick the delimiter
  - Remember subsequent calls are not the same as first (see man pages)

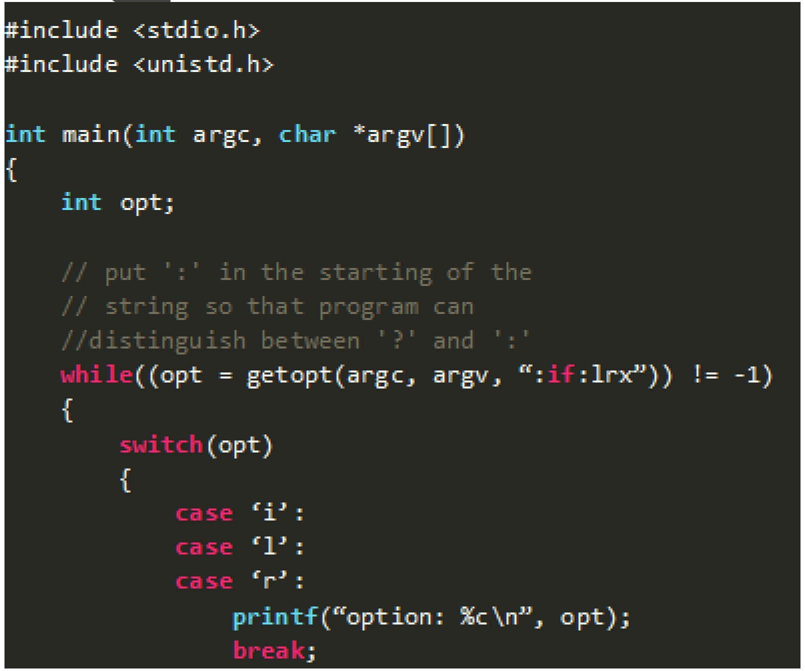
- Char * strdup(const char*s)
  - Returns pointer to new string, duplicate of argument string
  - Why is it useful? Because it allocates memory! Why is that useful? Leverage the heap to return values from a function easily.
  - DON’T FORGET TO FREE!!

Useful Functions

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

- Passed both the argc count and the argv array as parameters
- The optstring parameter is all the options that you want to include
- Arguments essentially delimited by the “-” sign
- Iterates through the arguments of argv starting with -
- Ancillary arguments to an option can be passed and included in the execution
- Returns the option character that was located
- Returns -1 of all arguments have been parsed
- Returns ‘?’ if character that was not defined in the optstring was encountered

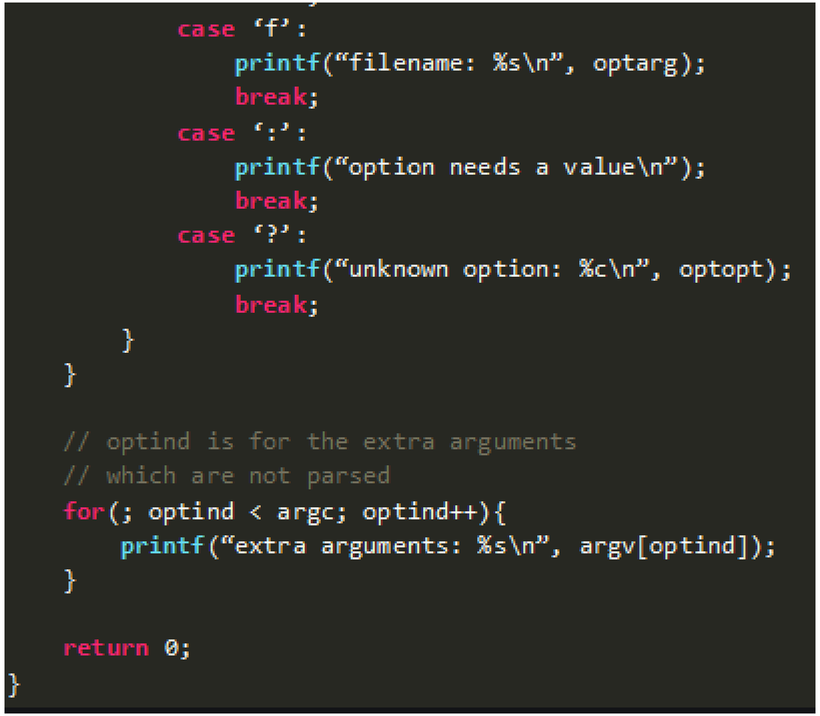
GeeksForGeeks Example (link)



```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int opt;

    // put ':' in the starting of the
    // string so that program can
    //distinguish between '?' and ':'
    while((opt = getopt(argc, argv, “:if:lrx”)) != -1)
    {
        switch(opt)
        {
            case ‘i’:
            case ‘l’:
            case ‘r’:
                printf(“option: %c\n”, opt);
                break;
```



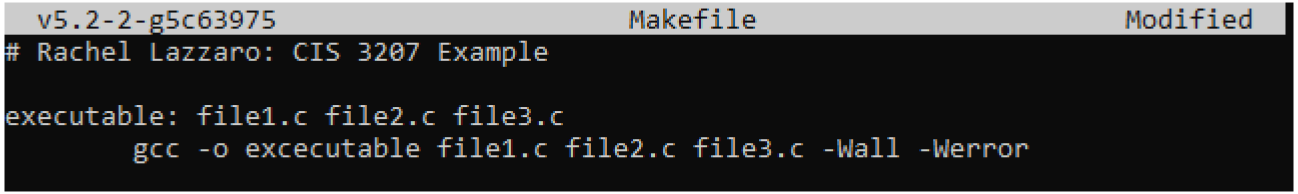
```c
            case ‘f’:
                printf(“filename: %s\n”, optarg);
                break;
            case ‘:’:
                printf(“option needs a value\n”);
                break;
            case ‘?’:
                printf(“unknown option: %c\n”, optopt);
                break;
        }
    }

    // optind is for the extra arguments
    // which are not parsed
    for(; optind < argc; optind++){
        printf(“extra arguments: %s\n”, argv[optind]);
    }

    return 0;
}
```

Creating a Makefile

- Need to prepare a makefile for this lab
- Tutorial on makefiles
- Create a file called makefile (no file ending format)
- Add all files that you want to compile together in the makefile



```
  v5.2-2-g5c63975                     Makefile                     Modified
# Rachel Lazzaro: CIS 3207 Example

executable: file1.c file2.c file3.c
        gcc -o excecutable file1.c file2.c file3.c -Wall -Werror
```

- Add a tab space before gcc, makefile will not run without a tab
- Type ‘make’ on the command line to compile your files together
- It will build executable for you to run
- Naming convention: Makefile should be capitalized, no file format (no .txt, .c etc)

Parting Tips

- Get started on understanding how to parse the command line now!
- This will be an important skill to learn that will help you in future labs
- Remember that your deliverable is due next week before lab
- Think about testing and your documentation now so that you don’t cram it at the end of the project!
- Take some time to explore the /proc file system before you start - getting in there will make it easier to understand the assignment
    - It will also make it easier for you to design methods of testing your program!
- Review the recommendations for this lab on the project page in Canvas and reach out with any concerns ASAP!
- GITHUB CLASSROOM LINK

thanks :-)