

Lab 1 - Observing Process Behaviour

Sasha Milne

Course: CSI3131Z

Professor: Mohamed Ibrahim

University of Ottawa

May 19, 2025

1 Introduction

This lab introduces fundamental operating system concepts by introducing the `/proc` directory – a virtual filesystem that provides access to process states.

2 Objectives

The goals of this lab are as follows:

1. **Process Monitoring via `/proc`:** To examine how a process executes and changes via the `/proc` virtual filesystem by directly inspecting its contents using CLI programs such as `cat`.
2. **Executing POSIX System Calls in C:** To develop and test simple C programs that demonstrate the use of `fork()` and `exec()` system calls for process creation and control.

3 Methodology

3.1 Initial Setup & Basic Exploration

Since a Linux environment is needed to explore the `/proc` filesystem and for POSIX compliance, the first step was to install a Virtual Machine (VM) and a Linux ISO to run on it. I already had an installation of Debian set up on my PC, so I used this as my testing environment. I then used basic CLI tools to explore the `/proc` filesystem.

```
sashamilne@sashadesktop:~  
File Edit View Search Terminal Help  
~ > ls /proc 02:07:23  
1      1299  1498  1906  264   552   8     97228  kpagecount  
10     13    15    1912  26407 553   815   97232  kpageflags  
1000   1300  1501  2     26408 557   816   982    loadavg  
1001   1303  1504  20    27848 558   86    983    locks  
1003   1313  15211 21    27849 561   88    999    meminfo  
1004   1318  154   23    28     564   89    acpi    misc  
101    1322  1559  23146 29     566   898    asound  modules  
1018   1333  16    24     3     570   90    buddyinfo mounts  
102    1353  1612  24330 30     571   91    bus     mtrr  
11     1354  1630  25     31     575   9410   cgroups net  
11199  1355  164   25071 326    6     9440   cmdline pagetypeinfo  
1137   1381  16403 25072 327    607   9441   consoles partitions  
1139   1386  165   25159 33     61    953    cpuinfo pressure  
1145   1397  16516 25160 34     62    9557   crypto  schedstat  
1147   14    1655   25161 345    63     96    devices self  
1153   1401  166   25162 35     64    96225  diskstats slabinfo  
1155   1405  167   25164 36     641   96280  dma     softirqs  
1169   1407  168   252   38     65    96324  driver  stat  
1170   1414  1710  25230 39     66    965    dynamic_debug swaps  
1181   1423  1723  25271 396    67    96564  execdomains sys  
12     14610 174   26     4     68    96638  fb       sysrq-trigger  
1254   14614 177   26158 40     69    96688  filesystems sysvipc  
1259   14617 17709 26159 41     70    96820  fs       thread-self
```

Figure 1: Exploring /proc using ls

```
sashamilne@sashadesktop:~  
File Edit View Search Terminal Help  
~ > cat /proc/version 02:07:26  
Linux version 6.1.0-12-amd64 (debian-kernel@lists.debian.org) (gcc-12 (Debian 12  
.2.0-14) 12.2.0, GNU ld (GNU Binutils for Debian) 2.40) #1 SMP PREEMPT_DYNAMIC D  
ebian 6.1.52-1 (2023-09-07)  
  
~ > cat /proc/cpuinfo 02:11:19  
processor       : 0  
vendor_id      : AuthenticAMD  
cpu family     : 23  
model          : 8  
model name     : AMD Ryzen 7 2700 Eight-Core Processor  
stepping       : 2  
microcode      : 0x800820d  
cpu MHz        : 3393.616  
cache size     : 512 KB  
physical id    : 0  
siblings       : 8  
core id        : 0  
cpu cores      : 8  
apicid         : 0  
initial apicid : 0  
fpu            : yes  
fpu_exception  : yes
```

Figure 2: Further exploration of /proc using cat

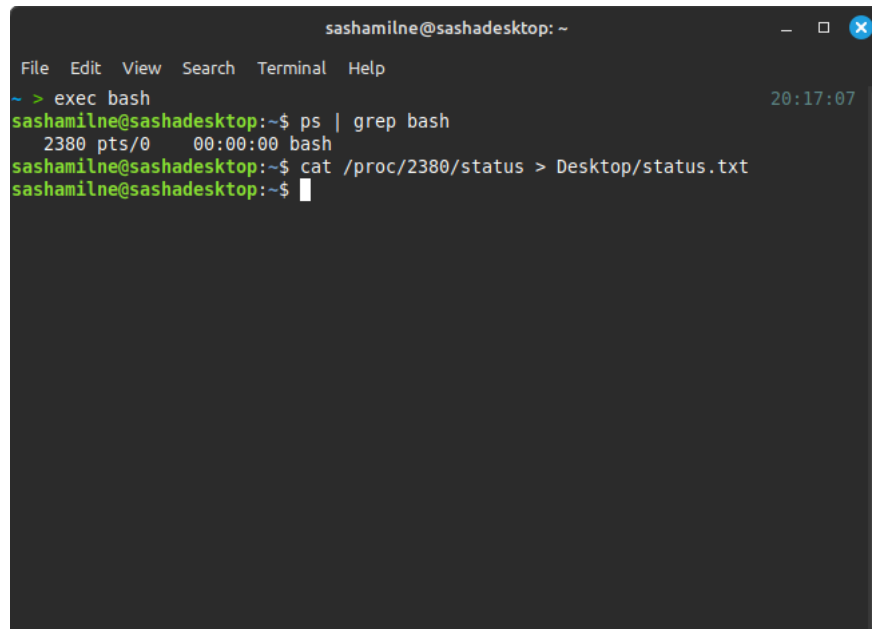
In *Figure 1*, the CLI output of `ls /proc` can be observed. This command lists all the contents of the `/proc` directory to `stdout`. Each folder with a number contains information about a process referenced by its Process ID (PID). There are other miscellaneous files which contain critical system info which can be observed.

In *Figure 2*, two such files are examined, `version` and `cpuinfo`, using `cat`. Here, information

is displayed onto the CLI about what version of **proc** is being used and information about the CPU. For example, I allocated 8 of my 16 threads in my Ryzen 7 2700, and we can observe the system seeing 8 CPU cores available.

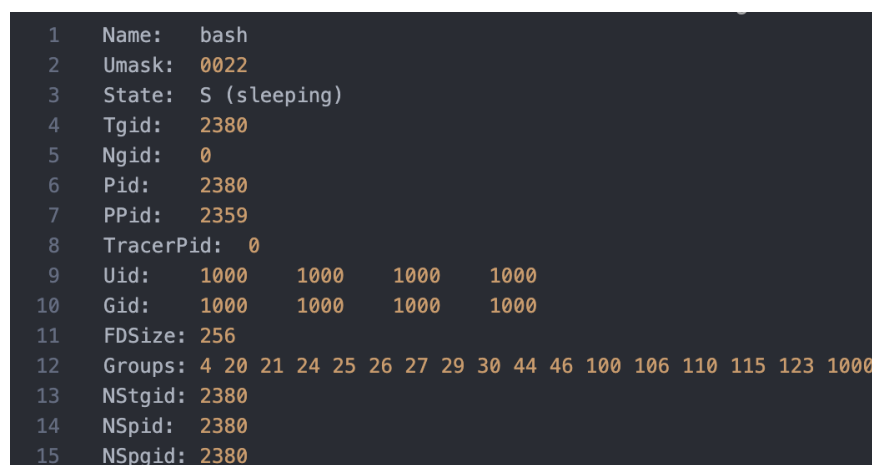
3.2 Finding Process Info

The goal of the next activity was to find the PID of the shell and explore its process status. We used **bash** as the shell and **ps** to get the PID.



```
sashamilne@sashadesktop: ~  
File Edit View Search Terminal Help  
~ > exec bash  
sashamilne@sashadesktop:~$ ps | grep bash  
2380 pts/0    00:00:00 bash  
sashamilne@sashadesktop:~$ cat /proc/2380/status > Desktop/status.txt  
sashamilne@sashadesktop:~$
```

Figure 3: Finding the PID of the shell and getting its status



```
1  Name:  bash  
2  Umask: 0022  
3  State: S (sleeping)  
4  Tgid: 2380  
5  Ngid: 0  
6  Pid: 2380  
7  PPid: 2359  
8  TracerPid: 0  
9  Uid: 1000 1000 1000 1000  
10 Gid: 1000 1000 1000 1000  
11 FDSize: 256  
12 Groups: 4 20 21 24 25 26 27 29 30 44 46 100 106 110 115 123 1000  
13 NSTgid: 2380  
14 NSpid: 2380  
15 NSpgid: 2380
```

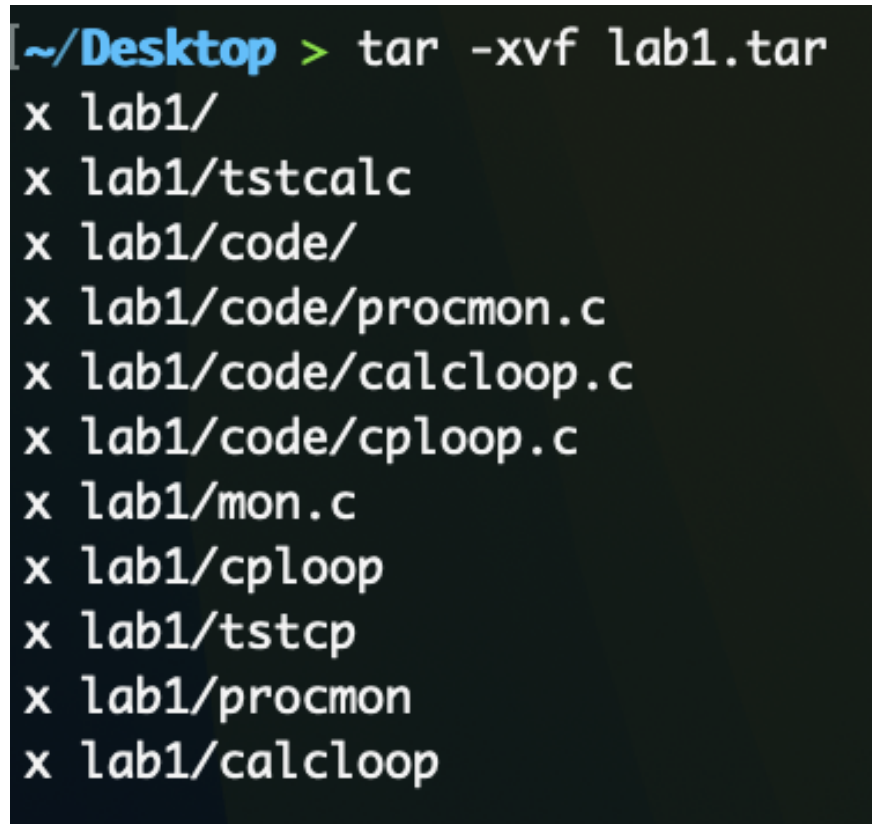
Figure 4: Viewing the output file

In *Figure 3*, the shell PID is obtained using **ps**. We then output the contents of **/proc/PID/status**, redirecting it into **status.txt**. The full contents of this file can be found

in `logs/status.log`, however, a screenshot is provided in *Figure 4*. From this figure, we can see that the process was sleeping at the time of execution of `cat` along with many other details.

3.3 Preparing Experiments & Understanding Tools

The goal of this section is to explore the state changes of processes using specially created programs. Using the provided `lab1.tar`, we will launch and observe other, more interesting programs.

A terminal window with a black background and white text. The prompt is `[~/Desktop >]` in blue. The command `tar -xvf lab1.tar` is entered. The output shows the extraction of various files and directories, each preceded by an 'x' character.

```
[~/Desktop > tar -xvf lab1.tar
x lab1/
x lab1/tstcalc
x lab1/code/
x lab1/code/procmon.c
x lab1/code/calclloop.c
x lab1/code/cplloop.c
x lab1/mon.c
x lab1/cplloop
x lab1/tsttcp
x lab1/procmon
x lab1/calclloop
```

Figure 5: Extracting lab1 tar file

After extracting the contents of `lab1.tar`, I moved everything to my workspace directory and reorganized the project structure for neatness. Inside the `lab1` directory, there are three compiled binaries: `procmon`, `calclloop`, and `cplloop`.

3.3.1 Analysis of Binaries

The purpose of the `procmon` program is to monitor the status of a process given its PID. The program accepts an argument PID which it then monitors using the `proc` filesystem.

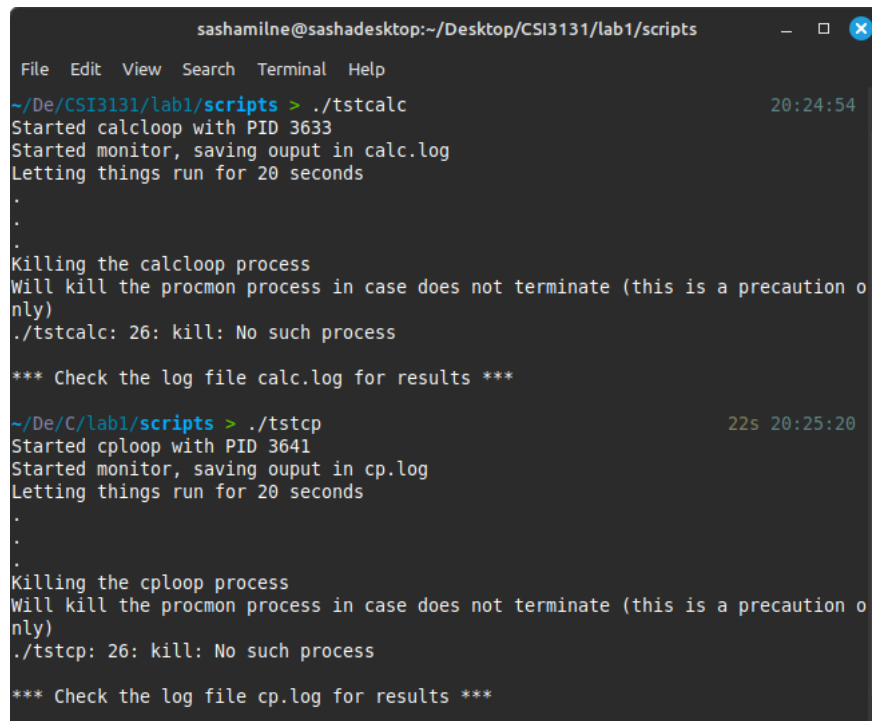
`cplloop` is a program that creates a file named *fromfile* containing 500 thousand "x" characters and runs ten iterations of a loop that sleeps for 10 seconds and copies the contents of *fromfile*

to a new file named *tofile*.

`calclloop` is a program that runs 10 iterations of a loop that sleeps for 3 seconds then increments a variable 400 million times

3.4 Running Provided Scripts

The next step is to run the scripts `tstcalc` and `tstcp` provided in the `lab1` directory. I moved the scripts to a `scripts` subdirectory and slightly modified the scripts to output the log files to a `logs` subdirectory for cleanliness.



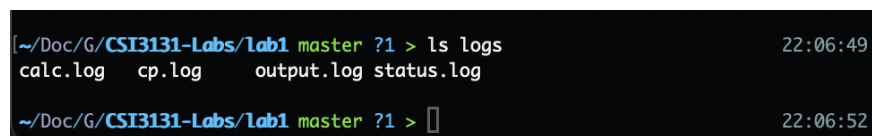
```
sashamilne@sashadesktop:~/Desktop/CSI3131/lab1/scripts
File Edit View Search Terminal Help
~/De/CSI3131/lab1/scripts > ./tstcalc 20:24:54
Started calclloop with PID 3633
Started monitor, saving output in calc.log
Letting things run for 20 seconds
.
.
.
Killing the calclloop process
Will kill the procmon process in case does not terminate (this is a precaution only)
./tstcalc: 26: kill: No such process

*** Check the log file calc.log for results ***

~/De/C/lab1/scripts > ./tstcp 22s 20:25:20
Started cploop with PID 3641
Started monitor, saving output in cp.log
Letting things run for 20 seconds
.
.
.
Killing the cploop process
Will kill the procmon process in case does not terminate (this is a precaution only)
./tstcp: 26: kill: No such process

*** Check the log file cp.log for results ***
```

Figure 6: Running `tstcalc` and `tstcp`



```
[~/Doc/G/CSI3131-Labs/lab1 master ?1 > ls logs 22:06:49
calc.log cp.log output.log status.log

~/Doc/G/CSI3131-Labs/lab1 master ?1 > 22:06:52
```

Figure 7: Confirming generation of log files using `ls`

I forgot to take screenshots of the verification that the log files were generated using `ls`, so I had to do it after the fact using a cloned version of `lab1` on my Macbook Air, which is why the filepaths are not consistent.

3.5 Programming mon.c

The program `mon.c` is intended to take the name of another program as an argument. Using this argument, it launches the other program and monitors it using `procmon`. The first challenge of this spec was to convert the relative path passed in through `argv` to an absolute path accessible by `execlp`.

```
17     const char* program = argv[1];
18
19     char path[PATH_MAX];
20     ssize_t len = readlink("/proc/self/exe", path, sizeof(path) - 1);
21     if(len == -1)
22     {
23         perror("readlink");
24         exit(EXIT_FAILURE);
25     }
26     path[len] = '\0';
27
28
29     char* last_slash = strrchr(path, '/');
30     if(last_slash != NULL)
31     {
32         *(last_slash+1) = '\0';
33     }
34
35
36     char program_path[PATH_MAX];
37     char procmon_path[PATH_MAX];
38
39     snprintf(program_path, sizeof(program_path), "%s%s", path, program);
40     snprintf(procmon_path, sizeof(procmon_path), "%s%s", path, "procmon");
```

Figure 8: Finding the absolute path

```
42     // 1. Launch the program specified by the variable 'fileName' and get its pid
43     pid_t child_pid = fork();
44
45     if (child_pid == 0) {
46         execlp(program_path, program, NULL);
47         perror("Failed to exec program");
48         exit(-1);
49     }
50
51     if (child_pid < 0) {
52         perror("Failed to fork");
53         exit(-1);
54     }
55
56     // 2. Launch 'procmon pid' where pid is the pid of the program launched in step 1
57     char pid_str[20];
58     sprintf(pid_str, "%d", child_pid);
59
60     pid_t procmon_pid = fork();
61     if (procmon_pid == 0) {
62         execlp(procmon_path, "procmon", pid_str, NULL);
63         perror("Failed to exec program");
64         exit(-1);
65     }
66
67     if (procmon_pid < 0) {
68         perror("Failed to fork");
69         exit(-1);
70     }
```

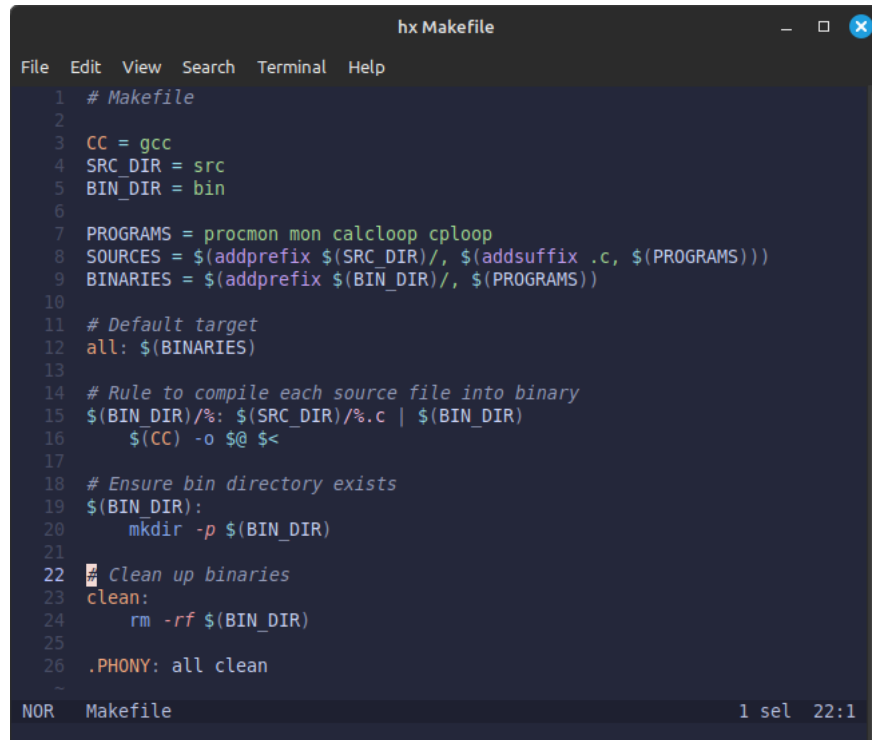
Figure 9: Starting program and procmon processes

Figure 9 demonstrates the usage of `fork` and `exec` to start new processes. `Fork` creates a copy of the existing process (child process) and returns the PID of the child in the parent

process while returning 0 for the child process. **Exec** replaces the existing process with a new one. Using these two functions, we can start two new processes, **procmon** and the process given in **argv[1]**

3.6 Compiling mon.c

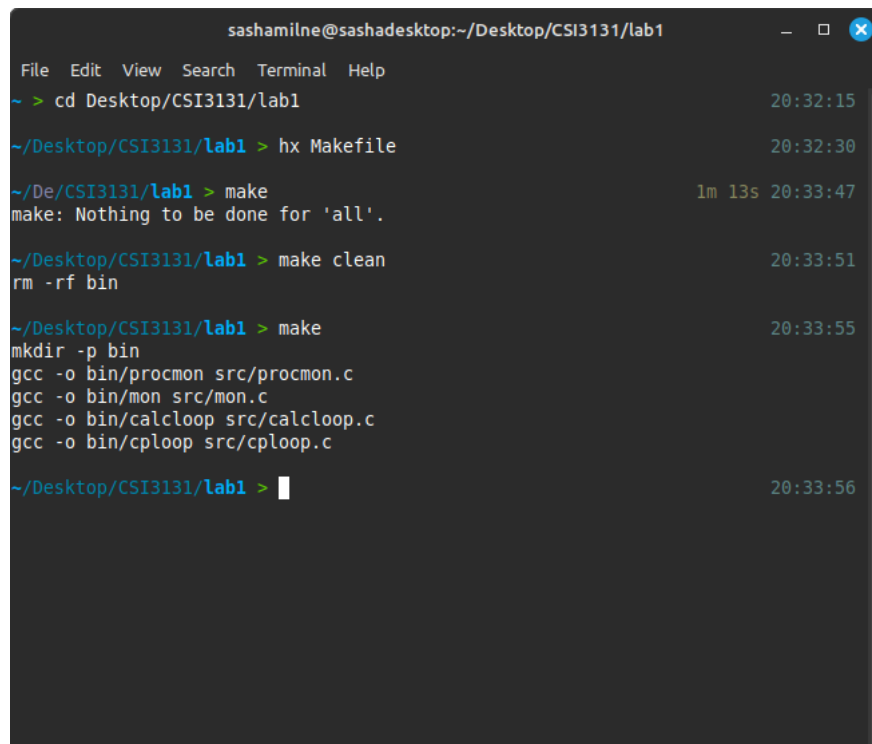
When it comes to compiling from source in C, I believe it is almost always worth the effort to create a **Makefile** instead of manually calling **gcc** everytime you want to compile a binary. I also took the time to reorganize the project structure for cleanliness.



```
1 # Makefile
2
3 CC = gcc
4 SRC_DIR = src
5 BIN_DIR = bin
6
7 PROGRAMS = procmon mon calcloop cploop
8 SOURCES = $(addprefix $(SRC_DIR)/, $(addsuffix .c, $(PROGRAMS)))
9 BINARIES = $(addprefix $(BIN_DIR)/, $(PROGRAMS))
10
11 # Default target
12 all: $(BINARIES)
13
14 # Rule to compile each source file into binary
15 $(BIN_DIR)/%: $(SRC_DIR)/%.c | $(BIN_DIR)
16     $(CC) -o $@ $<
17
18 # Ensure bin directory exists
19 $(BIN_DIR):
20     mkdir -p $(BIN_DIR)
21
22 # Clean up binaries
23 clean:
24     rm -rf $(BIN_DIR)
25
26 .PHONY: all clean
```

NOR Makefile 1 sel 22:1

Figure 10: Creating a Makefile to automate compilation process

A terminal window titled 'sashamilne@sashadesktop:~/Desktop/CSI3131/lab1' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the following sequence of commands and output:

```
~ > cd Desktop/CSI3131/lab1
~/Desktop/CSI3131/lab1 > hx Makefile
~/De/CSI3131/lab1 > make
make: Nothing to be done for 'all'.
~/Desktop/CSI3131/lab1 > make clean
rm -rf bin
~/Desktop/CSI3131/lab1 > make
mkdir -p bin
gcc -o bin/procmon src/procmon.c
gcc -o bin/mon src/mon.c
gcc -o bin/calclloop src/calclloop.c
gcc -o bin/cplloop src/cplloop.c
~/Desktop/CSI3131/lab1 >
```

Timestamps on the right side of the terminal indicate the time taken for each command: 20:32:15, 20:32:30, 1m 13s 20:33:47, 20:33:51, 20:33:55, and 20:33:56.

Figure 11: Running Makefile to recompile binaries

As demonstrated in *Figure 9*, the result is the same as manually calling `gcc -o bin/mon src/mon.c` except it is cleaner and more organized.

4 Results

4.1 Analysis of Log Files

4.1.1 calc.log

```
2
3      Monitoring /proc/3633/stat:
4
5      Time      State      SysTm      UsrTm
6      0      Sleeping(memory)      0      0
7      1      Sleeping(memory)      0      0
8      2      Sleeping(memory)      0      0
9      3      Running      0      0
10     4      Sleeping(memory)      0      89
11     5      Sleeping(memory)      0      89
12     6      Sleeping(memory)      0      89
13     7      Running      0      108
14     8      Sleeping(memory)      0      177
15     9      Sleeping(memory)      0      177
16    10      Sleeping(memory)      0      177
17    11      Running      0      225
18    12      Sleeping(memory)      0      266
19    13      Sleeping(memory)      0      266
20    14      Sleeping(memory)      0      266
21    15      Running      0      333
22    16      Sleeping(memory)      0      355
23    17      Sleeping(memory)      0      355
24    18      Sleeping(memory)      0      355
25    19      Running      0      440
26    procmon: Cannot open /proc/3633/stat, the monitored process is not running any more.
```

Figure 12: Output of calc.log

The output of this log file shows a pattern of the process sleeping for three seconds and then running for one. This follows the behaviour of the code exactly as `calcloop` is programmed to sleep for three seconds before each iteration. `SysTm` stays at 0 for the entirety of the execution which is also expected as the program does not operate in kernel mode at any point. Finally, we see fairly intensive increases in `UstrTm` when the program is running. This is due to CPU-intensive looping (400 million increments) each iteration which uses exclusively user-space computation.

4.1.2 cp.log

```
2
3      Monitoring /proc/7005/stat:
4
5 Time      State      SysTm    UsrTm
6 0 Running      0        0
7 1 Sleeping(memory) 0        0
8 2 Sleeping(memory) 0        0
9 3 Running      0        0
10 4 Sleeping(memory) 30       11
11 5 Sleeping(memory) 30       11
12 6 Sleeping(memory) 30       11
13 7 Sleeping(memory) 65       19
14 8 Sleeping(memory) 65       19
15 9 Sleeping(memory) 65       19
16 10 Running     87       23
17 11 Sleeping(memory) 99       27
18 12 Sleeping(memory) 99       27
19 13 Sleeping(memory) 99       27
20 14 Sleeping(memory) 130      38
21 15 Sleeping(memory) 130      38
22 16 Sleeping(memory) 130      38
23 17 Sleeping(memory) 161      49
24 18 Sleeping(memory) 161      49
25 19 Sleeping(memory) 161      49
26 procmon: Cannot open /proc/7005/stat, the monitored process is not running any more.
```

Figure 13: Output of cp.log

4.2 Testing mon.c

The output of this process shows

5 Discussion

Interpret your results. Compare with theoretical expectations. Explain discrepancies.

6 Conclusion

References

- Author, *Title*, Journal Name, Year.
- Author, *Title*, Book Name, Publisher, Year.