CS 481: Operating Systems
Lab 1

Jacob Hurst

February 14, 2019

1 Part 1

1.1 Write a description of how the fork and execv system calls, along with file and pipe operations, were used to make the command work. You'll need to also mention what each part of the command does, and how the command line arguments are handled at the system call level.

I obtained my process id and used 'strace -ff -p 1512 -o part1.p' to trace and log all system calls from a new screen. On the initial screen, I ran 'cat usr/share/dict/american-english | grep "cs" | uniq | sort | head -n 5 | rev' as the command to be traced. I viewed the resulting files from the trace and used 'cat * | grep (execve/clone)" on them to determine how the system executed the command.

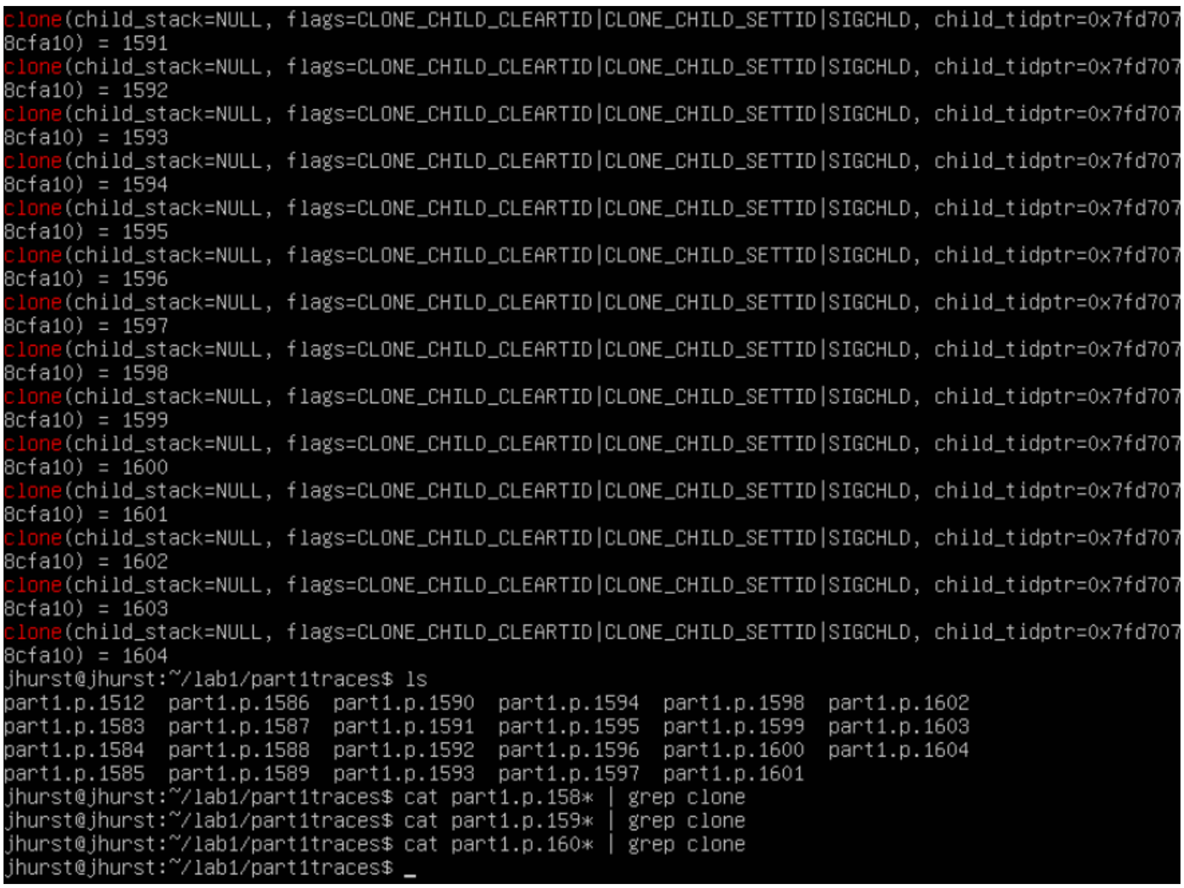


Figure 1: clone system calls from strace on command

In the above figure, we can observe that none of the 'cat (part1.p.158*, part1.p.159*, or part1.p.160*) | grep clone' commands output anything. The only process that isn't included in that check is the initial process, 1512. In running that same command on part1.p.1512, we can observe that all clone calls originate from the original process. The original process, at time of call, cloned children to handle all of the separate pieces of the command typed.
As the process shares parts of the execution context (property of clone command), we can know that the file and pipe operations must've been initialized in that initial trace. Using 'cat

part1.p.1512 | grep pipe' we can confirm that all pipes are set up in the original process prior to cloning (by verifying that no pipes were created in the other processes by 'cat (part1.p.158*, part1.p.159*, or part1.p.160*) | grep pipe', output being empty).

```
jhurst@jhurst:~/lab1/part1traces$ ls
part1.p.1512  part1.p.1586  part1.p.1590  part1.p.1594  part1.p.1598  part1.p.1602
part1.p.1583  part1.p.1587  part1.p.1591  part1.p.1595  part1.p.1599  part1.p.1603
part1.p.1584  part1.p.1588  part1.p.1592  part1.p.1596  part1.p.1600  part1.p.1604
part1.p.1585  part1.p.1589  part1.p.1593  part1.p.1597  part1.p.1601
jhurst@jhurst:~/lab1/part1traces$ cat * | grep execve
execve("/bin/cat", ["cat", "/usr/share/dict/american-english"], 0x5636c73c9f60 /* 25 vars */) = 0
execve("/bin/grep", ["grep", "--color=auto", "cs"], 0x5636c73c9f60 /* 25 vars */) = 0
execve("/usr/bin/uniq", ["uniq"], 0x5636c73c9f60 /* 25 vars */) = 0
execve("/usr/bin/sort", ["sort"], 0x5636c73c9f60 /* 25 vars */) = 0
execve("/usr/bin/head", ["head", "-n", "5"], 0x5636c73c9f60 /* 25 vars */) = 0
execve("/usr/bin/rev", ["rev"], 0x5636c73c9f60 /* 25 vars */) = 0
jhurst@jhurst:~/lab1/part1traces$ _
```

Figure 2: execve system calls from strace on command

As shown in the above figure, execve was called from 6 different processes. The first seen is the cat command, followed by grep, uniq, sort, head, and rev. The execve system call executes the program pointed to by filename argument.

The individual parts of the command being traced work by:

1. '|' (pipe) - modifies standard input, standard output, and standard error values in the file descriptor table for processes in order to redirect IO for interprocess communication.

2. 'cat' - concatenating and printing files from standard input to standard output.

3. 'grep "cs"' - searches the input for the specified "cs" pattern and prints the list of words containing the pattern to standard output.

4. 'uniq' - omits any repeated lines from the input and prints the input without any repeated lines to standard output.

5. 'sort' - alphabetically sorts and prints the sorted input to standard output.

6. 'head -n 5' - selects and prints the first 5 lines of the list to standard output.

7. 'rev' - reverses the characters in each line of input, prints the result to standard output (no pipe redirect here).

2 Part 2

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

int BYTES_READ = 0;

int main() {
	printf("Hello, World!\n");

	FILE *fp = fopen("infile.txt", "r");
	char *buffer;

	if(fp == NULL) exit(1);
	else {
		fseek(fp, 0, SEEK_END);
		BYTES_READ = ftell(fp);
		fseek(fp, 0, SEEK_SET);
		buffer = (char *) malloc(BYTES_READ);
		if(BYTES_READ != fread(buffer, sizeof(*buffer), BYTES_READ, fp)) {
			free(buffer);
			exit(1);
		}
	}
	//printf("%d, %s\n", BYTES_READ, buffer);

	sleep(1000000);
	free(buffer);
	fclose(fp);
	exit(0);
}
```

Figure 3: C Code to print "Hello, World!", read from a file into a buffer on the heap, set a global integer to bytes read, then go into wait state.

2.1 For the three processes that were created, look at their virtual address spaces and the different mappings. Which ones are likely unique to each process and which ones are likely pointing to the same physical pages of memory? How do you know? What similarities and differences do you notice from one process to another for the same binary?

In observing the mappings for each of the three running processes, it's likely that the bulk of '.so' shared object files are shared between processes as these files have only the read (and possibly execute) permissions set and have the same inode values. If the region was mapped from a file, the inode number is the file number. It should be noted, however, that in each of these '.so' files it is not the data that is shared but it is the code for the shared library which is mapped. Heap and Stack are unique to each process in most modern systems. The object, gconv-modules.cache, appears to be shared in each as its perms field has the s flag set.

Comparing the mappings of each of the three running processes I noticed that the address fields all differed, which makes sense as these are different processes with different sets memory. Aside from that, all other fields, perms, offset, dev, inode, and pathname seemed to contain identical values. Stack, heap, vdso, vsyscall, and the various anonymous objects all had inode values of 0. I suspect this value might indicate that no inode is associated with the memory region.

2.2 Attach to one of the processes with gdb. Using screencaps of disassembly and core dumps, give some indication of where the main program is and what's there, where at least one library is and what's there, where the heap is and what's there, and where the stack is and what's there, and where the global variables are and what's there. Be sure to refer back to the maps from Question 2.

```
   0x000055de4bd388cf <+5>:     sub    $0x18,%rsp
   0x000055de4bd388d3 <+9>:     lea    0x18a(%rip),%rdi        # 0x55de4bd38a64
   0x000055de4bd388da <+16>:    callq  0x55de4bd38720 <puts@plt>
   0x000055de4bd388df <+21>:    lea    0x18c(%rip),%rsi        # 0x55de4bd38a72
   0x000055de4bd388e6 <+28>:    lea    0x187(%rip),%rdi        # 0x55de4bd38a74
   0x000055de4bd388ed <+35>:    callq  0x55de4bd38780 <fopen@plt>
   0x000055de4bd388f2 <+40>:    mov    %rax,-0x20(%rbp)
   0x000055de4bd388f6 <+44>:    cmpq   $0x0,-0x20(%rbp)
   0x000055de4bd388fb <+49>:    jne    0x55de4bd38907 <main+61>
   0x000055de4bd388fd <+51>:    mov    $0x1,%edi
   0x000055de4bd38902 <+56>:    callq  0x55de4bd38790 <exit@plt>
   0x000055de4bd38907 <+61>:    mov    -0x20(%rbp),%rax
   0x000055de4bd3890b <+65>:    mov    $0x2,%edx
   0x000055de4bd38910 <+70>:    mov    $0x0,%esi
   0x000055de4bd38915 <+75>:    mov    %rax,%rdi
   0x000055de4bd38918 <+78>:    callq  0x55de4bd38770 <fseek@plt>
   0x000055de4bd3891d <+83>:    mov    -0x20(%rbp),%rax
   0x000055de4bd38921 <+87>:    mov    %rax,%rdi
   0x000055de4bd38924 <+90>:    callq  0x55de4bd38750 <ftell@plt>
   0x000055de4bd38929 <+95>:    mov    %eax,0x2006e5(%rip)        # 0x55de4bf39014 <BYTES_READ>
   0x000055de4bd3892f <+101>:   mov    -0x20(%rbp),%rax
   0x000055de4bd38933 <+105>:   mov    $0x0,%edx
   0x000055de4bd38938 <+110>:   mov    $0x0,%esi
   0x000055de4bd3893d <+115>:   mov    %rax,%rdi
   0x000055de4bd38940 <+118>:   callq  0x55de4bd38770 <fseek@plt>
   0x000055de4bd38945 <+123>:   mov    0x2006c9(%rip),%eax        # 0x55de4bf39014 <BYTES_READ>
   0x000055de4bd3894b <+129>:   cltq
   0x000055de4bd3894d <+131>:   mov    %rax,%rdi
   0x000055de4bd38950 <+134>:   callq  0x55de4bd38760 <malloc@plt>
   0x000055de4bd38955 <+139>:   mov    %rax,-0x18(%rbp)
   0x000055de4bd38959 <+143>:   mov    0x2006b5(%rip),%eax        # 0x55de4bf39014 <BYTES_READ>
   0x000055de4bd3895f <+149>:   movslq %eax,%rbx
   0x000055de4bd38962 <+152>:   mov    0x2006ac(%rip),%eax        # 0x55de4bf39014 <BYTES_READ>
   0x000055de4bd38968 <+158>:   movslq %eax,%rdx
   0x000055de4bd3896b <+161>:   mov    -0x20(%rbp),%rcx
   0x000055de4bd3896f <+165>:   mov    -0x18(%rbp),%rax
   0x000055de4bd38973 <+169>:   mov    $0x1,%esi
   0x000055de4bd38978 <+174>:   mov    %rax,%rdi
   0x000055de4bd3897b <+177>:   callq  0x55de4bd38730 <fread@plt>
   0x000055de4bd38980 <+182>:   cmp    %rax,%rbx
   0x000055de4bd38983 <+185>:   je     0x55de4bd389a7 <main+221>
   0x000055de4bd38985 <+187>:   mov    -0x18(%rbp),%rax
   0x000055de4bd38989 <+191>:   mov    %rax,%rdi
   0x000055de4bd3898c <+194>:   callq  0x55de4bd38710 <free@plt>
   0x000055de4bd38991 <+199>:   mov    -0x20(%rbp),%rax
   0x000055de4bd38995 <+203>:   mov    %rax,%rdi
   0x000055de4bd38998 <+206>:   callq  0x55de4bd38740 <fclose@plt>
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb) x/s 0x55de4bd38a64
0x55de4bd38a64: "Hello, World!"
(gdb)
```

Figure 4: Disassembled code for main, print statement with the address associated to "Hello, World!" is shown.

```
(gdb) where
#0  0x00007fe8245da687 in __GI___waitpid (pid=-1, stat_loc=0x7fffecd7ecf0,
    options=10) at ../sysdeps/unix/sysv/linux/waitpid.c:30
#1  0x00005622cfd81ae9 in ?? ()
#2  0x00005622cfd8321b in wait_for ()
#3  0x00005622cfd72af4 in execute_command_internal ()
#4  0x00005622cfd72b82 in execute_command ()
#5  0x00005622cfd5d254 in reader_loop ()
#6  0x00005622cfd5bc7f in main ()
```

Using the command 'where' to backtrace to main - my original program. The output is shown above, entry #6 on the output directs us to the location of main at address 0x00005622cfd5bc7f.

This corresponds to the first entry in the proc map as shown below (note the read and execute permissions).

```
5622cfd2b000-5622cfe2f000 r-xp 00000000 08:01 31588359                   /bin/bash
```

In the proc mapping, below, the shared code from libraries are marked with read and execute permissions.

Using the command 'where', I was able to backtrace the program state and locate several libraries. An interesting library found via disas is the wait library __GI___waitpid where the function is currently at in execution. In the figure below, you can see the program is currently at a cmp statement - it's likely comparing a value to the signal value it's waiting on.

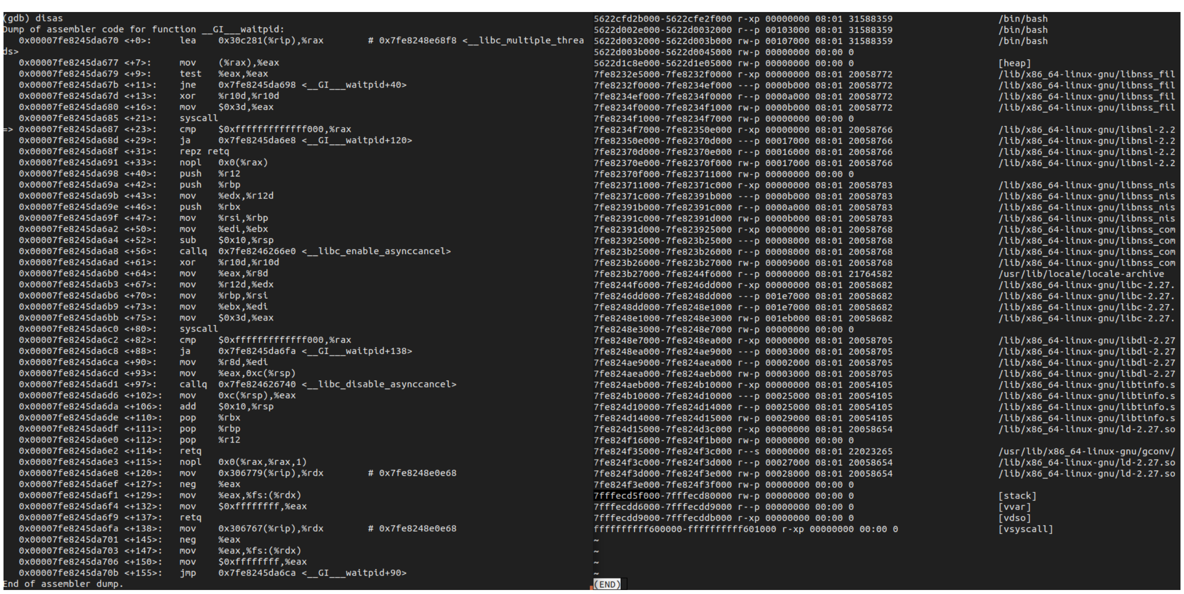


Figure 5: Disassembled code of current program state (left), proc mapping (right).

The heap resides in the below address range and contains contextual items, dynamically allocated runtime memory, and user-allocated memory. Searching through the heap, we can find local values to the function.

```
5622d1c8e000-5622d1e05000 rw-p 00000000 00:00 0                          [heap]
```

Below are the locations on heap of the printed message "Hello, World!" and the message from file "Goodbye, World!".

```
0x55de4c71025f: ""
0x55de4c710260: "Hello, World!\n"
0x55de4c71026f: ""
```

Figure 6: Address of "Hello, World!" on heap.

```
0x55de4c71089f: ""
0x55de4c7108a0: "Goodbye, World!\n"
0x55de4c7108b1: ""
```

Figure 7: Address of "Goodbye, World!" on heap.

As you can see from the below proc map, both messages reside in the range of the heap.

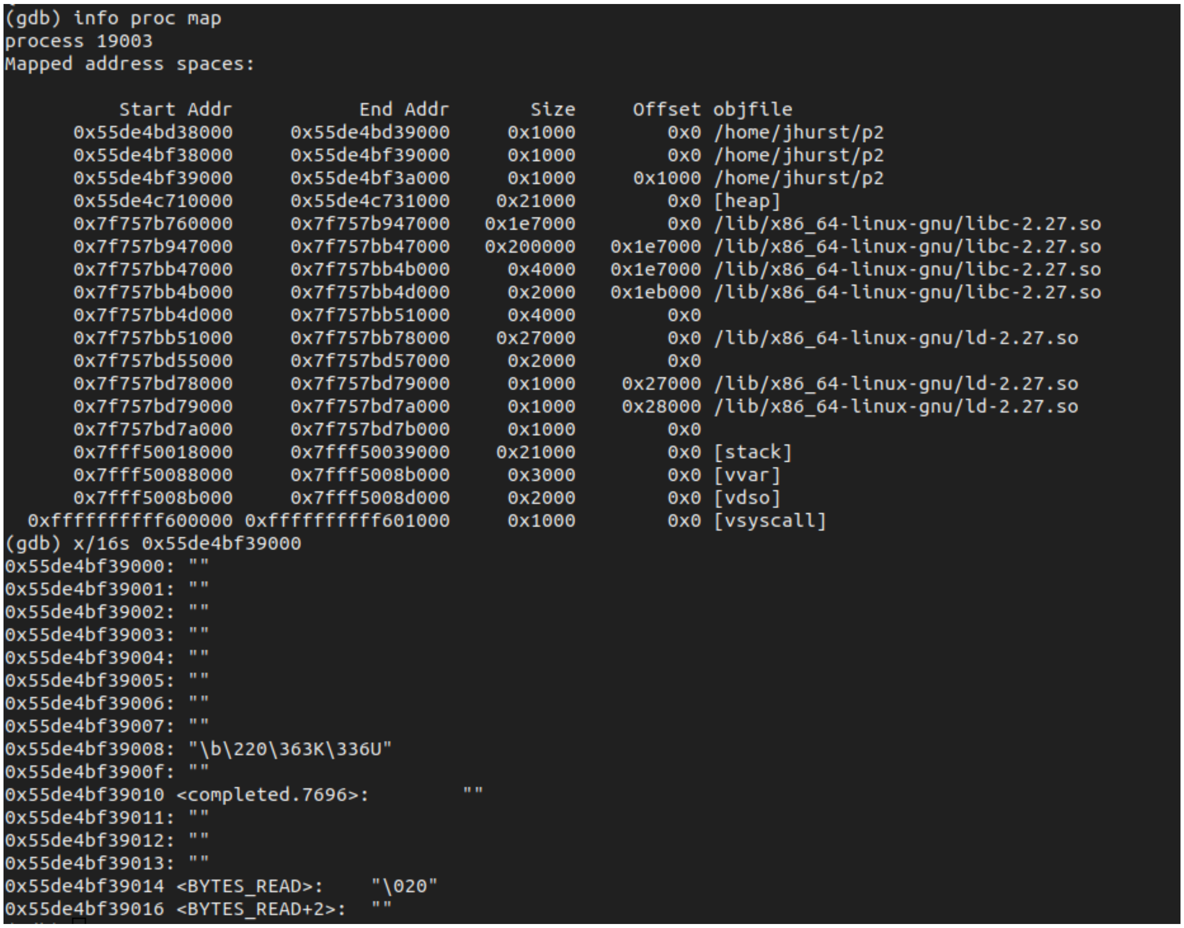
```
(gdb) info proc map
process 19003
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
      0x55de4bd38000     0x55de4bd39000     0x1000        0x0 /home/jhurst/p2
      0x55de4bf38000     0x55de4bf39000     0x1000        0x0 /home/jhurst/p2
      0x55de4bf39000     0x55de4bf3a000     0x1000     0x1000 /home/jhurst/p2
      0x55de4c710000     0x55de4c731000    0x21000        0x0 [heap]
      0x7f757b760000     0x7f757b947000   0x1e7000        0x0 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7f757b947000     0x7f757bb47000   0x200000   0x1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7f757bb47000     0x7f757bb4b000     0x4000   0x1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7f757bb4b000     0x7f757bb4d000     0x2000   0x1eb000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7f757bb4d000     0x7f757bb51000     0x4000        0x0
      0x7f757bb51000     0x7f757bb78000    0x27000        0x0 /lib/x86_64-linux-gnu/ld-2.27.so
      0x7f757bd55000     0x7f757bd57000     0x2000        0x0
      0x7f757bd78000     0x7f757bd79000     0x1000    0x27000 /lib/x86_64-linux-gnu/ld-2.27.so
      0x7f757bd79000     0x7f757bd7a000     0x1000    0x28000 /lib/x86_64-linux-gnu/ld-2.27.so
      0x7f757bd7a000     0x7f757bd7b000     0x1000        0x0
      0x7fff50018000     0x7fff50039000    0x21000        0x0 [stack]
      0x7fff50088000     0x7fff5008b000     0x3000        0x0 [vvar]
      0x7fff5008b000     0x7fff5008d000     0x2000        0x0 [vdso]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
(gdb)
```

Figure 8: The proc map showing address range of heap.

The stack resides in the below address range and contains the current execution state including local variables.

```
7ffc5961a000-7ffc5963b000 rw-p 00000000 00:00 0                          [stack]
```

The global variables are in their own memory area separate from the heap and stack. The name of the global int was BYTES_READ, by searching various addresses with the 'find' command. I located it in one of the objfiles, /home/jhurst/p2 (third line from the top), as shown below.



```
(gdb) info proc map
process 19003
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
      0x55de4bd38000     0x55de4bd39000     0x1000        0x0 /home/jhurst/p2
      0x55de4bf38000     0x55de4bf39000     0x1000        0x0 /home/jhurst/p2
      0x55de4bf39000     0x55de4bf3a000     0x1000     0x1000 /home/jhurst/p2
      0x55de4c710000     0x55de4c731000    0x21000        0x0 [heap]
      0x7f757b760000     0x7f757b947000   0x1e7000        0x0 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7f757b947000     0x7f757bb47000   0x200000   0x1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7f757bb47000     0x7f757bb4b000     0x4000   0x1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7f757bb4b000     0x7f757bb4d000     0x2000   0x1eb000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7f757bb4d000     0x7f757bb51000     0x4000        0x0
      0x7f757bb51000     0x7f757bb78000    0x27000        0x0 /lib/x86_64-linux-gnu/ld-2.27.so
      0x7f757bd55000     0x7f757bd57000     0x2000        0x0
      0x7f757bd78000     0x7f757bd79000     0x1000    0x27000 /lib/x86_64-linux-gnu/ld-2.27.so
      0x7f757bd79000     0x7f757bd7a000     0x1000    0x28000 /lib/x86_64-linux-gnu/ld-2.27.so
      0x7f757bd7a000     0x7f757bd7b000     0x1000        0x0
      0x7fff50018000     0x7fff50039000    0x21000        0x0 [stack]
      0x7fff50088000     0x7fff5008b000     0x3000        0x0 [vvar]
      0x7fff5008b000     0x7fff5008d000     0x2000        0x0 [vdso]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
(gdb) x/16s 0x55de4bf39000
0x55de4bf39000: ""
0x55de4bf39001: ""
0x55de4bf39002: ""
0x55de4bf39003: ""
0x55de4bf39004: ""
0x55de4bf39005: ""
0x55de4bf39006: ""
0x55de4bf39007: ""
0x55de4bf39008: "\b\220\363K\336U"
0x55de4bf3900f: ""
0x55de4bf39010 <completed.7696>:        ""
0x55de4bf39011: ""
0x55de4bf39012: ""
0x55de4bf39013: ""
0x55de4bf39014 <BYTES_READ>:    "\020"
0x55de4bf39016 <BYTES_READ+2>:  ""
```

Figure 9: Address of the global variables.

2.3 What is your process waiting for? How can you confirm this? Try to use language that is specific to kernel data structures or Linux system implementation, rather than high-level, "it's waiting for a timer".

```
ps -eo pid,args,wchan | less
  908 ./p2                          hrtimer\_nanosleep
  909 ./p2                          hrtimer\_nanosleep
  910 ./p2                          hrtimer\_nanosleep
```

The processes are waiting for a signal from hrtimer_nanosleep. This method can be used for a program to sleep for specific intervals of time, with nanosecond accuracy.

The call to sleep() makes the calling thread sleep until seconds seconds have elapsed or a signal arrives which is not ignored by calling hrtimer_nanosleep().

2.4 Exactly how many voluntary and involuntary context switches has the process made? Is that what you expected? Why or why not?

```
cat /proc/631/status | tail -n 2
  voluntary\_ctxt\_switches: 19
  nonvoluntary\_ctxt\_switches: 0
```

I expected the context switches to be mostly voluntary as the program sets itself in a sleep state and tells the operating system to wake it via signal.

3 Part 3

3.1 Can you find evidence of dynamic priorities being used to make processes that do a lot of I/O more responsive by giving them a higher priority?

In order to test this, I wrote two programs - cpuhog.c and iohog.c. The below figures show the code for each.

```c
#include <stdio.h>

int main() {
        printf("start\n");
        volatile unsigned long long i = 1;
        while(1) {
                i *= 2;
        }
        printf("stop\n");
        exit(0);
}
```

Figure 10: C program to simulate a CPU bound process.

```c
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv) {
        int fd = open(argv[1], O_DIRECT);
        off_t size = lseek(fd, 0, SEEK_END);

        for(int i = 0; i < 10000000; i++) {
                lseek(fd, rand() % size, SEEK_SET);
        }

        close(fd);
        exit(0);
}
```

Figure 11: C program to simulate an IO bound process.

The code for iohog was inspired by this posting on stack overflow. To run the iohog, I ran the following command which will run it through find for every file in the filesystem.

```
find / -exec ./iohog {} \; 2>/dev/null
```

Below is the output from top which details the niceness, priority, and usages for both programs. Unfortunately, I did not witness this niceness value or the priority value change but I suspect this would be a good way of determining dynamic priority changes.

```
top - 03:51:53 up 1 day, 20:20,  4 users,  load average: 2.00, 1.91, 1.25
Tasks: 119 total,   3 running,  63 sleeping,   4 stopped,   0 zombie
%Cpu(s): 29.9 us, 13.3 sy,  0.0 ni, 56.5 id,  0.0 wa,  0.0 hi,  0.3 si,  0.0 st
KiB Mem :  8167948 total,  7275804 free,   138968 used,   753176 buff/cache
KiB Swap:  2023420 total,  2023420 free,        0 used.  7724220 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 4180 jhurst    20   0    4508    812    748 R 100.0  0.0  14:28.70 cpuhog
 4253 jhurst    20   0    4376    712    648 R  26.5  0.0   0:00.80 iohog
 3331 jhurst    20   0   30740   3448   2420 S   0.7  0.0   0:10.65 screen
 4179 jhurst    20   0   42804   3968   3372 R   0.7  0.0   0:06.54 top
```

Figure 12: Priority results from top.

3.2 What kinds of things can you find in the file descriptor tables of the processes in the system? Can you find network sockets? Named pipes? Unnamed pipes? Plain old files? UNIX sockets that are not network sockets? Other types of interprocess communication? Libraries? What else that's interesting?

Possible value types for the file descriptors are cwd (current working directory), txt (text file), mem (memory mapped file), mmap (memory mapped device), 0r (standard input), 1u (standard output), 2u (standard error). and a general descriptor of the form (number)(letter), where number is any whole number greater than or equal to 0 and letter is one of r (read), w (write), u (read and write).
In the below figure, the file descriptors for the login process are shown. The login process has all of the standard input, output, and error descriptor values. It also has two additional file descriptors, one of type unix and the other of type FIFO. The FIFO descriptor is a named pipe. The unix file descriptor is some interprocess communication socket (type=DGRAM).

```
login   3125 root  mem    REG                8,2   170960 137531 /lib/x86_64-linux-gnu/ld-2.27.so
login   3125 root    0u   CHR                4,1      0t0     20 /dev/tty1
login   3125 root    1u   CHR                4,1      0t0     20 /dev/tty1
login   3125 root    2u   CHR                4,1      0t0     20 /dev/tty1
login   3125 root    3u  unix 0xffff8e768ad6f000      0t0  35281 type=DGRAM
login   3125 root    6w  FIFO               0,22      0t0    614 /run/systemd/sessions/45.ref
jhurst@jhurst:/proc$ sudo lsof -p 3125
```

Figure 13: Using lsof to view login process file descriptors.

In the below figure, the file descriptors for the current bash process are shown. The bash process also has all of the standard descriptors as well as one additional file descriptor, 255u.

```
bash    3332 jhurst  mem    REG    8,2    26376    8077 /usr/lib/x86_64-linux-gnu/gconv/gconv-modules
.cache
bash    3332 jhurst    0u   CHR  136,0      0t0       3 /dev/pts/0
bash    3332 jhurst    1u   CHR  136,0      0t0       3 /dev/pts/0
bash    3332 jhurst    2u   CHR  136,0      0t0       3 /dev/pts/0
bash    3332 jhurst  255u   CHR  136,0      0t0       3 /dev/pts/0
jhurst@jhurst:/proc$
```

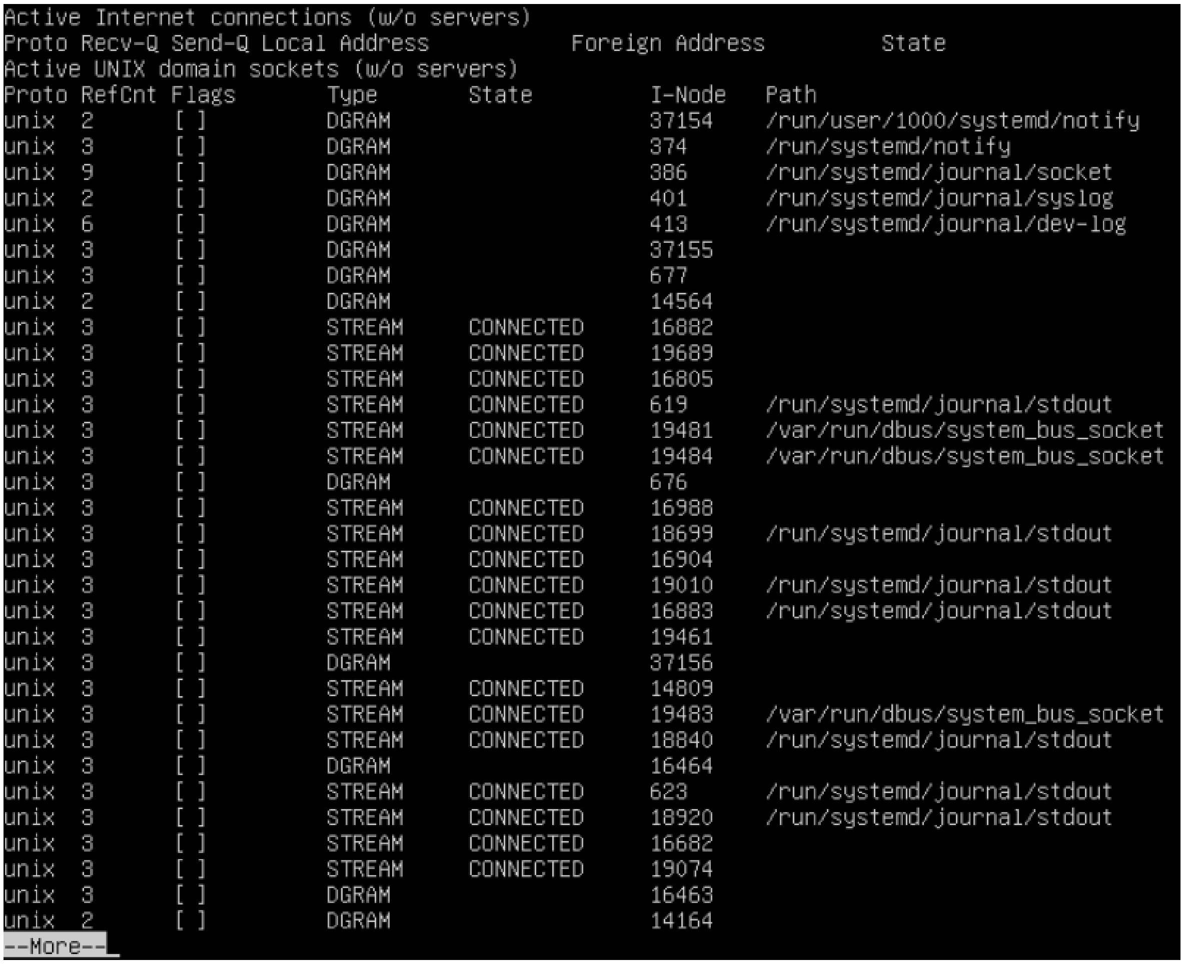
Figure 14: Using lsof to view bash process file descriptors.

In the below figure, the file descriptors for sshd are shown. The standard file descriptors input, output, and error appear to be redirected though as the types are not standard CHR type and the last entry of name is different (/dev/null, and type=STREAM). The two additional file descriptors, 3u and 4u, are TCP sockets.

```
sshd    1099 root  mem    REG                8,2    170960 137531 /lib/x86_64-linux-gnu/ld-2.27.so
sshd    1099 root    0r   CHR                1,3       0t0      6 /dev/null
sshd    1099 root    1u  unix 0xffff8e768ae4b000       0t0  21672 type=STREAM
sshd    1099 root    2u  unix 0xffff8e768ae4b000       0t0  21672 type=STREAM
sshd    1099 root    3u  IPv4              21679       0t0    TCP *:ssh (LISTEN)
sshd    1099 root    4u  IPv6              21683       0t0    TCP *:ssh (LISTEN)
jhurst@jhurst:/proc$ sudo lsof -p 1099
```

Figure 15: Using lsof to view sshd process file descriptors.

In the below figure, various active connections on the system are shown. The internal message bus daemon, dbus, makes up a majority of these communications. Dbus is a library that provides one-to-one communication between any two applications.



```
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags       Type       State         I-Node   Path
unix  2      [ ]         DGRAM                    37154    /run/user/1000/systemd/notify
unix  3      [ ]         DGRAM                    374      /run/systemd/notify
unix  9      [ ]         DGRAM                    386      /run/systemd/journal/socket
unix  2      [ ]         DGRAM                    401      /run/systemd/journal/syslog
unix  6      [ ]         DGRAM                    413      /run/systemd/journal/dev-log
unix  3      [ ]         DGRAM                    37155
unix  3      [ ]         DGRAM                    677
unix  2      [ ]         DGRAM                    14564
unix  3      [ ]         STREAM     CONNECTED     16882
unix  3      [ ]         STREAM     CONNECTED     19689
unix  3      [ ]         STREAM     CONNECTED     16805
unix  3      [ ]         STREAM     CONNECTED     619      /run/systemd/journal/stdout
unix  3      [ ]         STREAM     CONNECTED     19481    /var/run/dbus/system_bus_socket
unix  3      [ ]         STREAM     CONNECTED     19484    /var/run/dbus/system_bus_socket
unix  3      [ ]         DGRAM                    676
unix  3      [ ]         STREAM     CONNECTED     16988
unix  3      [ ]         STREAM     CONNECTED     18699    /run/systemd/journal/stdout
unix  3      [ ]         STREAM     CONNECTED     16904
unix  3      [ ]         STREAM     CONNECTED     19010    /run/systemd/journal/stdout
unix  3      [ ]         STREAM     CONNECTED     16883    /run/systemd/journal/stdout
unix  3      [ ]         STREAM     CONNECTED     19461
unix  3      [ ]         DGRAM                    37156
unix  3      [ ]         STREAM     CONNECTED     14809
unix  3      [ ]         STREAM     CONNECTED     19483    /var/run/dbus/system_bus_socket
unix  3      [ ]         STREAM     CONNECTED     18840    /run/systemd/journal/stdout
unix  3      [ ]         DGRAM                    16464
unix  3      [ ]         STREAM     CONNECTED     623      /run/systemd/journal/stdout
unix  3      [ ]         STREAM     CONNECTED     18920    /run/systemd/journal/stdout
unix  3      [ ]         STREAM     CONNECTED     16682
unix  3      [ ]         STREAM     CONNECTED     19074
unix  3      [ ]         DGRAM                    16463
unix  2      [ ]         DGRAM                    14164
--More--
```

Figure 16: Using netstat to view active connections.

3.3 Take a look at the process tree. Where is your command shell in the tree and what is its "heritage" all the way back to the init process? What are the other branches? Are they system services? Other ways of logging in? Something else?

```
systemd(1)-+-accounts-daemon(780)-+-{accounts-daemon}(799)
           |                      `-{accounts-daemon}(822)
           |-atd(894)
           |-cron(815)
           |-dbus-daemon(785)
           |-irqbalance(860)---{irqbalance}(882)
           |-login(3125)---bash(3247)---screen(3330)---screen(3331)---bash(3332)-+-more(3348)
           |                                                                      `-pstree(3347)
           |-lvmetad(424)
           |-lxcfs(898)-+-{lxcfs}(939)
           |            |-{lxcfs}(940)
           |            |-{lxcfs}(1895)
           |            `-{lxcfs}(1896)
           |-networkd-dispat(885)---{networkd-dispat}(1120)
           |-polkitd(999)-+-{polkitd}(1033)
           |              `-{polkitd}(1039)
           |-rsyslogd(899)-+-{rsyslogd}(963)
           |               |-{rsyslogd}(964)
           |               `-{rsyslogd}(965)
           |-snapd(930)-+-{snapd}(1067)
           |            |-{snapd}(1069)
           |            |-{snapd}(1070)
           |            |-{snapd}(1071)
           |            |-{snapd}(1072)
           |            |-{snapd}(1101)
           |            |-{snapd}(1115)
           |            |-{snapd}(1124)
           |            |-{snapd}(1125)
           |            |-{snapd}(1153)
           |            |-{snapd}(1204)
           |            |-{snapd}(1205)
           |            |-{snapd}(1244)
           |            |-{snapd}(1790)
           |            `-{snapd}(2044)
           |-sshd(1099)
           |-systemd(3236)---(sd-pam)(3237)
--More--
```

Figure 17: The process tree

If you follow from systemd(1) to login(3125), the command shell is on this branch from bash(3247). There are currently 2 screen processes opened for this bash. The bash located within screen(3331), labeled bash(3332), is the process which ran 'pstree -s -p — more' in order to get the figure above. You can observe the branch which creates more(3348) and pstree(3347) immediately following.
The other branches are mostly system services, systemd is the system and service manager, cron is a daemon for executing scheduled commands, atd runs jobs queued for later execution, dbus-daemon is a message bus used for one-to-one communication between applications, irqbalance distributes hardware interrupts, networkd-dispatcher is a service for systemd-networkd connection status changes, rsyslogd handles system logging, sshd is an OpenSSH daemon.

3.4 Take a look at the filesystem tree. What are each of the major subdirectories in the root "/" directory for? Which ones take up the most space on the filesystem? What kinds of special filesystems and files can you find, like proc filesystems, named pipes, setuid bits, hard and soft links, etc.?

```
jhurst@jhurst:/$ ls -alhs
total 2.0G
4.0K drwxr-xr-x  23 root root 4.0K Feb  6 21:03 .
4.0K drwxr-xr-x  23 root root 4.0K Feb  6 21:03 ..
4.0K drwxr-xr-x   2 root root 4.0K Feb  6 21:32 bin
4.0K drwxr-xr-x   3 root root 4.0K Feb  6 21:36 boot
   0 drwxr-xr-x  18 root root 3.8K Feb 13 16:33 dev
4.0K drwxr-xr-x  94 root root 4.0K Feb  6 21:59 etc
4.0K drwxr-xr-x   3 root root 4.0K Feb  6 21:09 home
   0 lrwxrwxrwx   1 root root   33 Feb  6 21:03 initrd.img -> boot/initrd.img-4.15.0-45-generic
   0 lrwxrwxrwx   1 root root   33 Jul 25  2018 initrd.img.old -> boot/initrd.img-4.15.0-29-generic
4.0K drwxr-xr-x  22 root root 4.0K Feb  6 21:29 lib
4.0K drwxr-xr-x   2 root root 4.0K Jul 25  2018 lib64
 16K drwx------   2 root root  16K Feb  6 21:00 lost+found
4.0K drwxr-xr-x   2 root root 4.0K Jul 25  2018 media
4.0K drwxr-xr-x   2 root root 4.0K Jul 25  2018 mnt
4.0K drwxr-xr-x   2 root root 4.0K Jul 25  2018 opt
   0 dr-xr-xr-x 124 root root    0 Feb 12 02:45 proc
4.0K drwx------   3 root root 4.0K Feb  6 21:09 root
   0 drwxr-xr-x  25 root root  880 Feb 13 16:34 run
 12K drwxr-xr-x   2 root root  12K Feb  6 21:33 sbin
4.0K drwxr-xr-x   4 root root 4.0K Feb  6 21:09 snap
4.0K drwxr-xr-x   2 root root 4.0K Jul 25  2018 srv
2.0G -rw-------   1 root root 2.0G Feb  6 21:03 swap.img
   0 dr-xr-xr-x  13 root root    0 Feb 12 02:44 sys
4.0K drwxrwxrwt   9 root root 4.0K Feb 13 17:00 tmp
4.0K drwxr-xr-x  10 root root 4.0K Jul 25  2018 usr
4.0K drwxr-xr-x  13 root root 4.0K Jul 25  2018 var
   0 lrwxrwxrwx   1 root root   30 Feb  6 21:03 vmlinuz -> boot/vmlinuz-4.15.0-45-generic
   0 lrwxrwxrwx   1 root root   30 Jul 25  2018 vmlinuz.old -> boot/vmlinuz-4.15.0-29-generic
```

Figure 18: The filesystem tree

Major subdirectories of "/":

1. bin - user binaries: contains binary executables, common user commands are located here (ps, ls, etc.).
2. boot - boot loader files: contains boot loader related files.
3. dev - device files: contains device files (terminal devices - tty, usb devices, etc.).
4. etc - configuration files: contains configuration files required by programs.
5. home - home directories: contains user stored personal files.
6. lib - system libraries: contains library files that support the binaries under bin and sbin.
7. media - removable media devices: mount directory for removable devices.
8. mnt - mount directory: temporary, used for mounting filesystems.
9. opt - optional add-on applications: contains add-on applications from individual vendors.

10. proc - process information: contains information about system processes.

11. sbin - system binaries: contains binary executables, common system commands are located here (reboot, iptables, etc.).

12. tmp - temporary files: contains temporary files created by system and users, files deleted on reboot.

13. usr - user programs: contains binaries, libraries, documentation, code, etc.

14. var - variable files: contains files that are expected to grow (ex: system log files).

Using the command 'du -sh *', the top consumers of space for the major subdirectories are usr (1009Mb), lib (792Mb), var (643Mb), and boot (140Mb).

There are lots of symbolic links throughout the subdirectories. For example in /bin/ domainname → hostname, lessfile → lesspipe. In /dev/, I found a links from stderr → /proc/self/fd/2, stdin → /proc/self/fd/0, and stdout → /proc/self/fd/1. In /sbin/, commands like shutdown, halt, and reboot linked to /bin/systemctl.

In /proc/, I found an interesting file called timer_statuses which contains information regarding all timers being used by the system including how long they've been running, when they will expire, etc..

3.5 Teach me something interesting about a modern Linux system. By exploring the processes in the system, the filesystem, address spaces, file descriptor tables, states, etc. what sticks out as being particularly interesting?

Exploring the /proc subdirectory further, I found a couple interesting features.

In the /proc/sys/ subdirectory I located, among others, a directory labeled kernel. In doing some reading online, I read that this source is not only a source of information, but it also allows you to change parameters within the kernel without the need for recompilation or even a system reboot. To change a value, simply echo the new value into the file. I didn't actually change anything but I found this interesting.

In /proc/sys/kernel/random, I found a couple interesting files like poolsize and entropy_avail. The available entropy on my machine was 1263.

In the command /proc/self/(property), self refers to current calling process and can be used to view your process properties more simply than echo $$ followed by /proc/PID/(property).

In /proc there is a file called stat which contains overall/various statistics about the system, such as the number of page faults since the system was booted.