Name: Michael Rojas SID#: 862034677
Name: Carson Welty SID#: 862012918

CS 202 Lab #1 Report

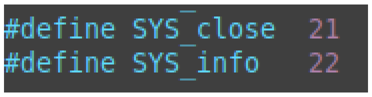
List of all edited files:

- syscall.h
- syscall.c
- sysproc.c
- usys.pl
- user.h
- proc.c
- proc.h
- defs.h
- test.c

The rest of the report will detail how we defined the system call and how the program can interact with it via numerical input and how we implemented the 3 specific functionalities of the system call, all in their own sections.

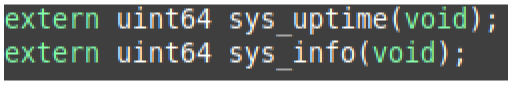
System call creation:

First, added the necessary definition line for "sys_info" to **syscall.h**:



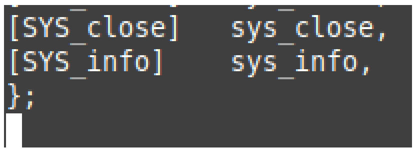
```
#define SYS_close  21
#define SYS_info   22
```

Then added the method signature declaration in **syscall.c**:



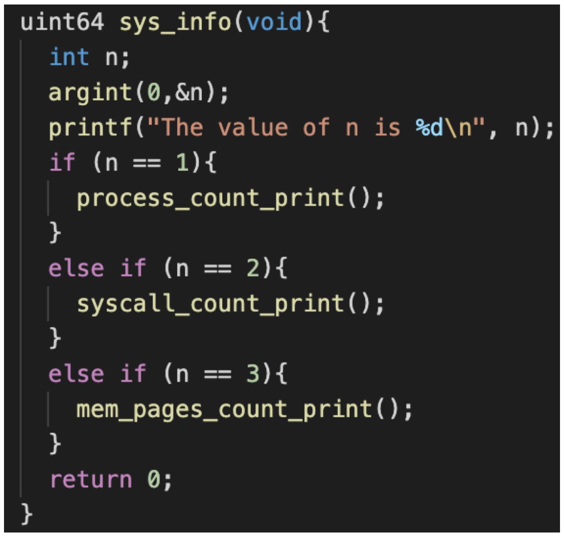
```
extern uint64 sys_uptime(void);
extern uint64 sys_info(void);
```

Then, also inside of **syscall.c,** added the "sys_info" system call to the array of system calls:



```
[SYS_close]   sys_close,
[SYS_info]    sys_info,
};
```

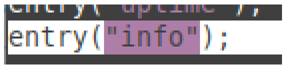
Inside of **sysproc.c**, we define the system call, uint64 sys_info(void):



```
uint64 sys_info(void){
  int n;
  argint(0,&n);
  printf("The value of n is %d\n", n);
  if (n == 1){
    process_count_print();
  }
  else if (n == 2){
    syscall_count_print();
  }
  else if (n == 3){
    mem_pages_count_print();
  }
  return 0;
}
```

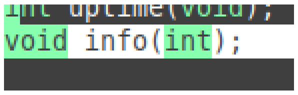
This system calls the corresponding function, specified with n, to output the desired information. These functions will be covered in greater detail later in this report.

Add the system call to **user/usys.pl**:



```
entry("uptime");
entry("info");
```

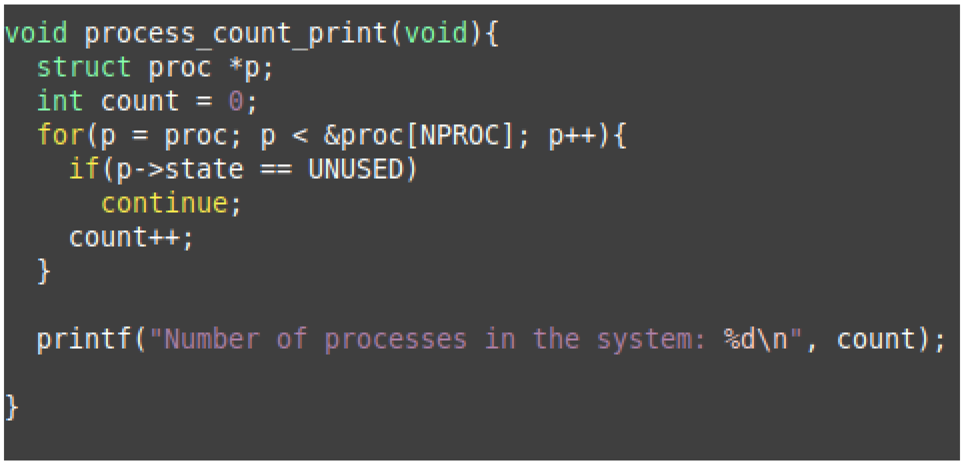
Then add it to the list of system calls inside of **user/user.h**:



```
int uptime(void);
void info(int);
```

First functionality:

If the input to the system call is 1, we run "void process_count_print(void)" in **proc.c**:



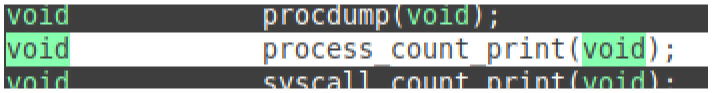
```
void process_count_print(void){
  struct proc *p;
  int count = 0;
  for(p = proc; p < &proc[NPROC]; p++){
    if(p->state == UNUSED)
      continue;
    count++;
  }

  printf("Number of processes in the system: %d\n", count);

}
```

This function will iterate all possible processes and check if they are in the "UNUSED" state (which would mean they are not activated / allocproc() was not called on them). If they are UNUSED, they are not a real / active process and we continue. Otherwise, we increment an integer storing the current process count. Once we have iterated all of the possible processes, we print out the current process count using printf().

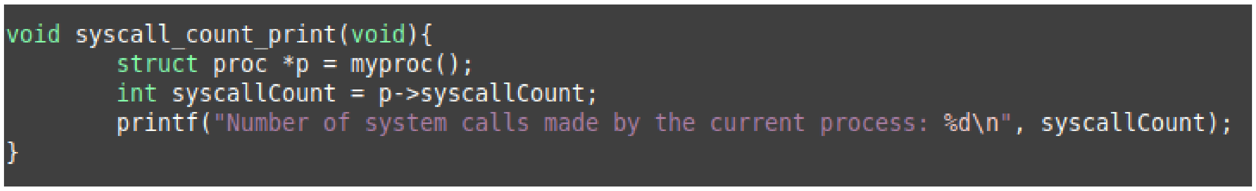
Inside of **defs.h**, we have the process_count_print() signature definition:



```
void            procdump(void);
void            process_count_print(void);
void            syscall_count_print(void);
```

Second functionality:

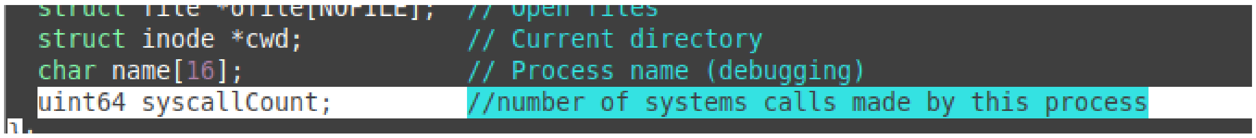
If the input to the system call is 2, we call syscall_count_print(), which is defined in **proc.c**:



```
void syscall_count_print(void){
        struct proc *p = myproc();
        int syscallCount = p->syscallCount;
        printf("Number of system calls made by the current process: %d\n", syscallCount);
}
```

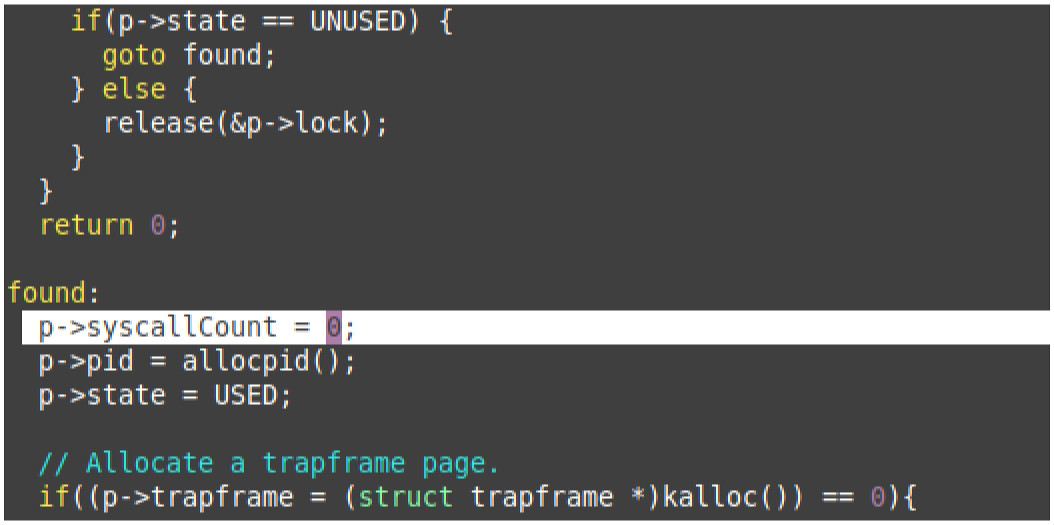
First, we're capturing myproc() in proc *p. Then, we set an int syscallCount equal to p->syscallCount. Finally, we simply print syscallCount.

This member, syscallCount, is an addition I made to the proc struct inside of it's definition in **proc.h:**



```
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  uint64 syscallCount;         //number of systems calls made by this process
};
```

It's just a uint64 field. It gets initialized inside of **proc.c** inside the allocproc() function:



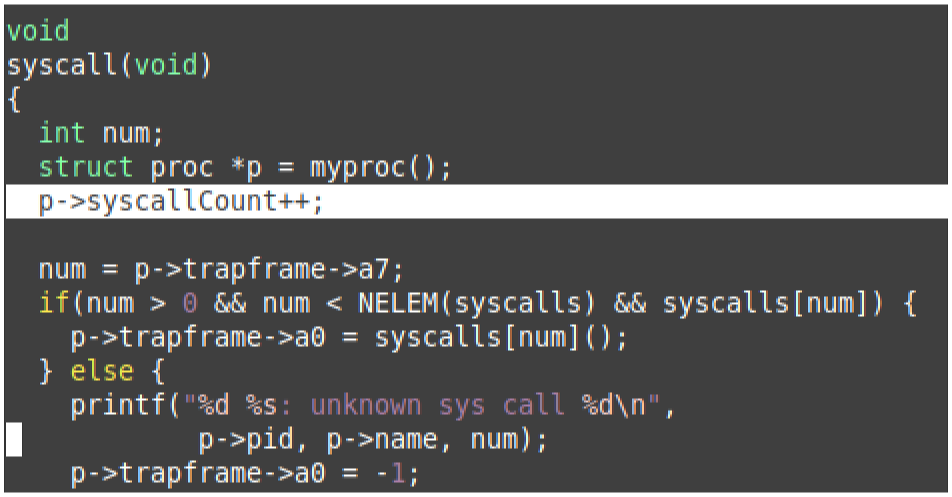
```
    if(p->state == UNUSED) {
      goto found;
    } else {
      release(&p->lock);
    }
  }
  return 0;

found:
  p->syscallCount = 0;
  p->pid = allocpid();
  p->state = USED;

  // Allocate a trapframe page.
  if((p->trapframe = (struct trapframe *)kalloc()) == 0){
```

This is because this is where we have found an UNUSED proc and are going to initialize it to a state that can run inside of the kernel. Therefore, initialize the syscallCount of the process to 0.

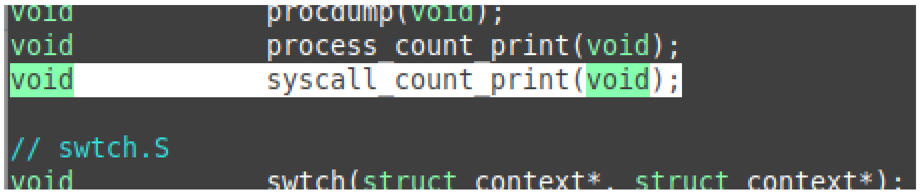
The "syscallCount" parameter is incremented inside of **syscall.c** inside of the void syscall(void) function, since this is the function responsible for running the correct syscall in syscalls[num]:



```
void
syscall(void)
{
  int num;
  struct proc *p = myproc();
  p->syscallCount++;

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
```

Proper reference in **defs.h**:

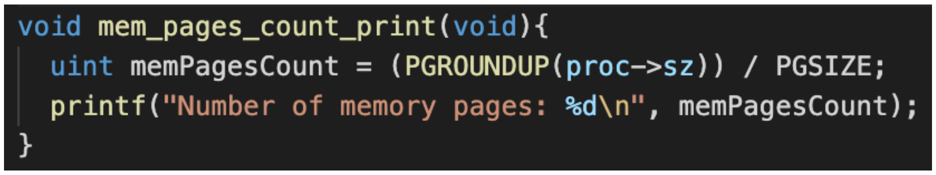


```
void            procdump(void);
void            process_count_print(void);
void            syscall_count_print(void);

// swtch.S
void            swtch(struct context*, struct context*);
```

Third functionality:

If the input to the system call is 3, we call mem_pages_count_print(), which is defined in **proc.c**:

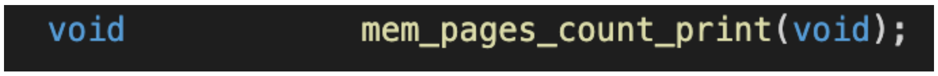


```
void mem_pages_count_print(void){
  uint memPagesCount = (PGROUNDUP(proc->sz)) / PGSIZE;
  printf("Number of memory pages: %d\n", memPagesCount);
}
```

We can calculate the number of memory pages being used by the current process by using PGSIZE and PGROUNDUP. These are defined for allocuvm, meaning allocate user virtual memory, which is found in the file mmu.h. The xv6 operating system uses allocuvm to allocate both page tables and virtual memory.

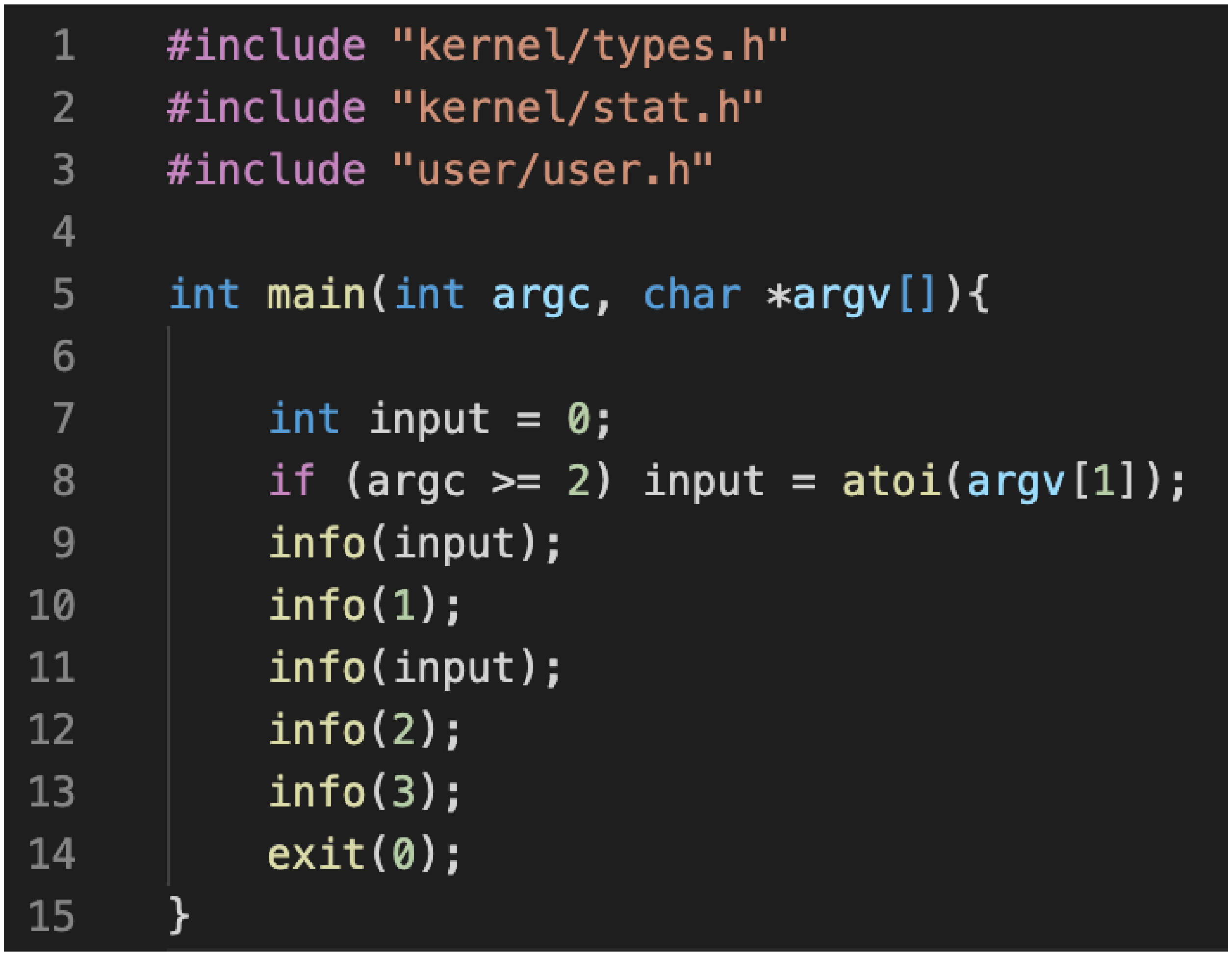
Using the proc struct’s sz attribute, we must use PGROUNDUP to find the rounded-up address of the current process, and then divide by the constant PGSIZE to get the total number of memory pages being used.

Inside of **defs.h**, we include the mem_pages_print() signature definition:



```c
void            mem_pages_count_print(void);
```

Test.c file:



```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

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

    int input = 0;
    if (argc >= 2) input = atoi(argv[1]);
    info(input);
    info(1);
    info(input);
    info(2);
    info(3);
    exit(0);
}
```

Output is shown in the demo video.