

## CS 202 Lab #1: System Call Implementation

Name	Student ID	Net ID
Mohammed Syed Akbar Hashmi	862393370	mhash034
Mohammad Mahaboob Ali Ashraf	862393694	mmoha055
Nazanin Nasiri Abrishamchi	862393735	nnasi004

Demo video link:

[https://drive.google.com/drive/folders/1QX7JOfcavmLpGrVC04xPe4I6\\_mKL4u4n?usp=sharing](https://drive.google.com/drive/folders/1QX7JOfcavmLpGrVC04xPe4I6_mKL4u4n?usp=sharing)

### List of Files Changed

#### 1. Root Files

##### a. Makefile

Here we add our newly added main function class to build upon building of the operating system.

```
M Makefile
114 # details:
115 # http://www.gnu.org/software
116 .PRECIOUS: %.o
117
118 UPROGS=\
119     $_cat\
120     $_echo\
121     $_forktest\
122     $_grep\
123     $_init\
124     $_kill\
125     $_ln\
126     $_ls\
127     $_mkdir\
128     $_rm\
129     $_sh\
130     $_stressfs\
131     $_usertests\
132     $_grind\
133     $_wc\
134     $_zombie\
135     $_lab1-sysinfo\
136
```

## 2. Kernel Files

### a. Syscall.h

Here we declare the two system calls that we have added to our OS, sysinfo and sysprocinfo

```
kernel > C syscall.h
1 // System call numbers
2 #define SYS_fork 1
3 #define SYS_exit 2
4 #define SYS_wait 3
5 #define SYS_pipe 4
6 #define SYS_read 5
7 #define SYS_kill 6
8 #define SYS_exec 7
9 #define SYS_fstat 8
10 #define SYS_chdir 9
11 #define SYS_dup 10
12 #define SYS_getpid 11
13 #define SYS_sbrk 12
14 #define SYS_sleep 13
15 #define SYS_uptime 14
16 #define SYS_open 15
17 #define SYS_write 16
18 #define SYS_mknod 17
19 #define SYS_unlink 18
20 #define SYS_link 19
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 #define SYS_info 22
24 #define SYS_procinfo 23
```

### b. Syscall.c

Here we do two things

A. We add the new system call mappings

B. We modify the syscall function to store the count of total system calls made and process specific system calls made.

```

kernel > C syscall.c
131 [SYS_close] sys_close,
132 [SYS_procin] sys_procin,
133 [SYS_info] sys_info,
134 };
135
136 void
137 syscall(void)
138 {
139     int num;
140     struct proc *p = myproc();
141
142     num = p->trapframe->a7;
143     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
144         // Use num to lookup the system call function for num, call it,
145         // and store its return value in p->trapframe->a0
146         p->trapframe->a0 = syscalls[num]();
147         // if(1==1){
148         //     printf("inc counter cnt_syscalls");
149         //     cnt_syscalls++;
150         //     p->TotalSysCallsMade++;
151         // }
152         // if(sizeof(syscalls)/sizeof(syscalls[0])
153         // else{
154         //     printf("skipping inc counter cnt_syscalls");
155         // }
156     } else {
157         printf("%d %s: unknown sys call %d\n",
158             p->pid, p->name, num);
159         p->trapframe->a0 = -1;
160     }
161 }
162
163

```

#### c. Sysproc.c

Here we call the system calls which we implemented in proc.c. This class is directly invoked by our user class and processes the parameters passed as arguments.

```

92
93 // sys info
94 uint64
95 sys_info(void){
96     printf("Hello.. this is sys info in sysproc.c\n");
97     int option;
98
99     argint(0, &option);
100     return info(option);
101 }
102
103 // sys proc info
104 uint64
105 sys_procin(void){
106     printf("Hello.. this is sys PROC info in sysproc.c\n");
107     // int option;
108     // argaddr(0, &option);
109     uint64 p;
110     argaddr(0, &p);
111     return procinfo((struct Pinfo *) p);
112 }

```

#### d. Proc.c

Here we define the two main system call functions needed in the lab. Proc info and info

```

705 // nextpid = nextpid + 1;
706
707 // printf("count of sys call from bootup of OS is %d", cnt_syscalls);
708 int cnt_processes = 0;
709 switch (userInput)
710 {
711     case 0:
712         //the total number of active processes (ready, running, waiting, or zombie) in the system
713         /* code */
714         printf("In switch case 0");
715         for (p = proc; p < &proc[NPROC]; p++)
716         {
717             acquire(&p->lock);
718             // printf("p-> id is ; %d", p->pid);
719             if (p->pid == 0)
720             {
721                 release(&p->lock);
722                 break; // no more processes in the array
723             }
724             // UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE
725             // printf(" p-> id is %d", (p->pid));
726             // printf("\n");
727
728             if (p->state == RUNNABLE || p->state == RUNNING || p->state == ZOMBIE || p->state == SLEEPING) //
729             {
730                 cnt_processes++;
731             }
732             release(&p->lock);
733         }
734
735         // printf("cnt active process %d", cnt_processes);
736         printf(" \n");
737         return cnt_processes;
738         // break;
739

```

```

kernel > C proc.c
762 }
763
764 /*
765 * int ppid: the PID of its parent process.
766 * int syscall_count: the total number of system calls that the current process has made so
767 far.(system call number not to be included)
768 * int page_usage: the current process's memory size in pages (e.g., 10000 bytes [] 3 pages)
769 */
770 int
771 procinfo(struct Pinfo *p)
772 // printf("in proc.c procinfo %d", p);
773 // printf("\n...1 in proc.c p->ppid %p", ((int*)(&p->ppid)));
774 // printf("\n...in proc.c p->ppid %d", (int*)(*p).ppid);
775 // printf("\n...2 in proc.c p->page_usage %d", ((int)(*(int*)(p+ sizeof(int)))));
776 // printf("\n...3 in proc.c p->page_usage %d", (int*)(&p->syscall_count));
777
778 struct Pinfo param;
779 struct proc * pr = myproc();
780 acquire(&wait_lock);
781 param.ppid = pr->parent->pid;
782 release(&wait_lock);
783 param.page_usage = (PGROUNDUP(pr->sz))/PGSIZE;
784 param.syscall_count = pr->TotalSysCallsMade;
785
786
787 uint64 addr;
788 argaddr(0,&addr);
789
790 if(copyout(pr->pagetable, addr, (char *) &param, sizeof(param)) < 0){
791     return -1;
792 }
793 // todo
794 // printf("p->ppid is %d -- p->syscall_count is %d -- p->page_usage is %d ",(&p->ppid),(p->sys
795

```

#### e. Proc.h

Here, we add a new variable to keep track of system calls made by each

process.

```
kernel > C proc.h
77      /* 264 */ uint64 t4;
78      /* 272 */ uint64 t5;
79      /* 280 */ uint64 t6;
80  };
81
82  enum procstate { UNUSED, USED, SLEEPING, RUNNABLE, R
83
84  // Per-process state
85  struct proc {
86      struct spinlock lock;
87
88      // p->lock must be held when using these:
89      enum procstate state;      // Process state
90      void *chan;                // If non-zero, sleep
91      int killed;                // If non-zero, have
92      int xstate;                // Exit status to be
93      int pid;                   // Process ID
94
95      int TotalSysCallsMade;
96      // wait_lock must be held when using this:
97      struct proc *parent;       // Parent process
```

f. Kalloc.c

Here we define a function which returns the total number of free memory pages in the system. We then added these codes to the file kernel/kalloc.c. Because kmem.freelist is a linked list of pointers of free memory pages, we must count the number of nodes in this list and return the results.

```
C syscall.h  C lab1-sysinfo.c  C proc.c  C defs.h  C
kernel > C kalloc.c
64
65  // returns the total free memory pages in the system
66  int getTotalNumberOfFreePages(void){
67      // printf("in kalloc.c .. getting number of free pages");
68      struct run *r;
69      int cnt = 0;
70      acquire(&kmem.lock);
71      r = kmem.freelist;
72
73      while (r)
74      {
75          cnt++;
76          // traverse the list untill next is null
77          if (r->next)
78          {
79              // printf(" cnt is %d ",cnt);
80              r = r->next;
81          }
82          else{
83              break;
84          }
85      }
86      release(&kmem.lock);
87
88  }
```

g. Defs.h

We define the two new calls for proc.c here, cnt\_syscalls and procdump

```
kernel > C defs.h
101 void      procinit(void);
102 void      scheduler(void) __attribute__((noreturn));
103 void      sched(void);
104 void      sleep(void*, struct spinlock*);
105 void      userinit(void);
106 int       wait(uint64);
107 void      wakeup(void*);
108 void      yield(void);
109 int       either_copyout(int user_dst, uint64 dst, void *src, uint64 len);
110 int       either_copyin(void *dst, int user_src, uint64 src, uint64 len);
111 int       info(int userInput); // sysinfo internally calls info() in proc.c
112 int       procinfo(struct Pinfo *p); // sysinfo internally calls procinfo() in proc.c
113 extern int cnt_syscalls;
114 void      procdump(void);
115
116 // swtch.S
117 void      swtch(struct context*, struct context*);
118
119 // spinlock.c
120 void      acquire(struct spinlock*);
121 int       holding(struct spinlock*);
122 void      initlock(struct spinlock*, char*);
```

3. User Files

a. User.h

Here we add the new functions in user code to make system calls

```
user > C user.h
23 int getpid(void);
24 char* sbrk(int);
25 int sleep(int);
26 int uptime(void);
27 int info(int);
28 int procinfo(struct Pinfo*);
```

4. New File

a. Lab1-sysinfo.c

This is the file which invokes our newly created processes and eventually makes the required system calls. It has the main function, struct to be filled and other

necessary imports.

```
user > C lab1-sysinfo.c
4 // #include "kernel/Defs.h"
5 // #include <stdlib.h>
6 // #include <stdio.h>
7 struct Pinfo {
8     int ppid;
9     int syscall_count;
10    int page_usage;
11 };
12 int main(int argc, char *argv[]){
13     printf("\n ***** BEGIN *****\n");
14     struct Pinfo param ;
15     // = malloc(sizeof(Pinfo));
16     param.ppid =123123;
17     param.syscall_count=192;
18     param.page_usage=12343;
19
20     int n_proc;
21
22     // mem = atoi(argv[2]);
23     n_proc = atoi(argv[1]);
24     printf(" i am main()....");
25     printf("\n");
26
27     printf("\n SYSINFO returns value is  %d \n", info(n_proc));
28
29     printf("\nMaking PROCINFO call\n");
30     int res = procinfo(&param);
31     printf("\nparam page_usage is %d", param.page_usage);
32     printf("\nparam ppid is %d", param.ppid);
33     printf("\nparam syscall_count is %d", param.syscall_count);
34     printf("\nres is %d", res);
35     printf("\nend of main()...\n");
36     printf("\n ***** END *****\n");
37     exit(0);
```

## Detailed Explanation on Changes made:

### Part 1: Sysinfo

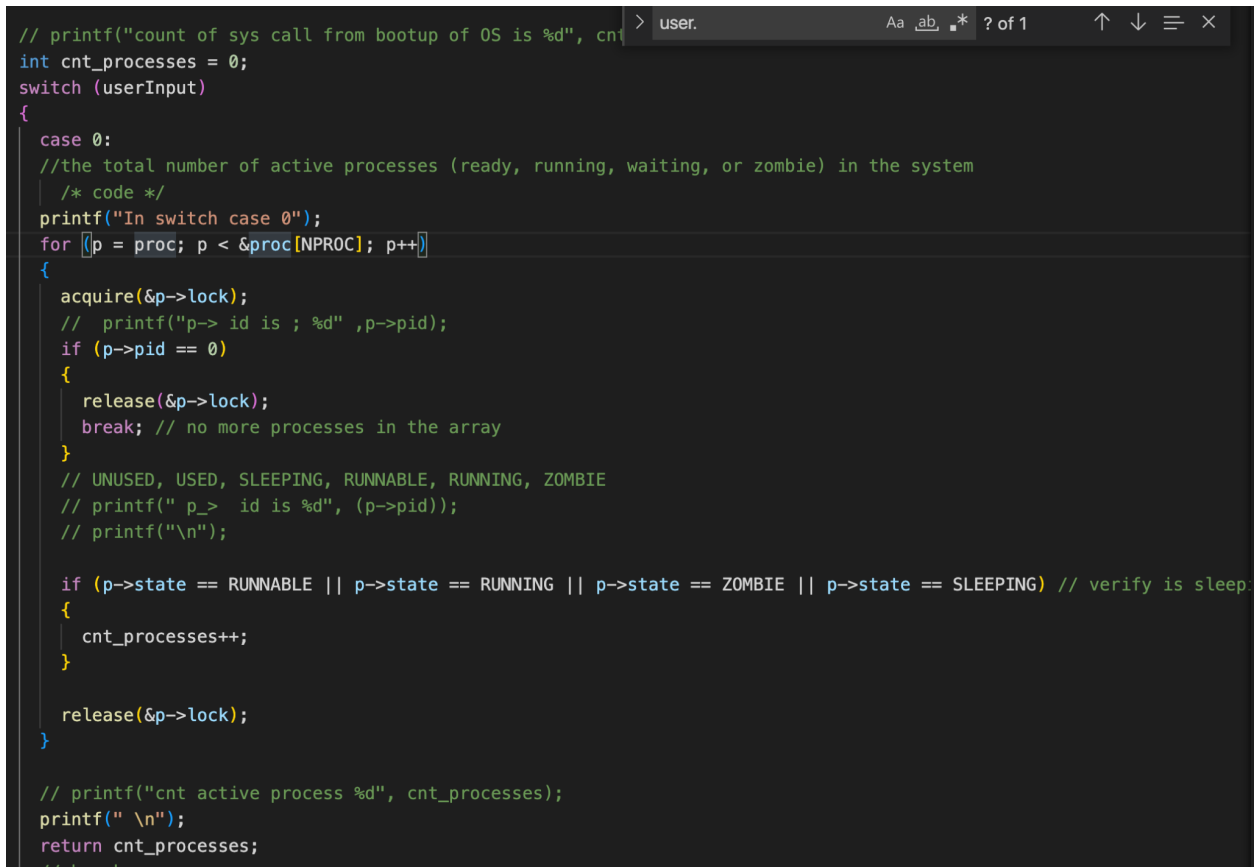
We are required to add a system call that takes an integer request param and returns the following

- If param == 0, the total number of active processes in the OS
- If param == 1, the total number of system calls made so far since launch
- If param == 2, the total number of free memory pages in the OS
- Otherwise, return -1 if the request does not match

#### 1. To get the total number of active processes in the OS:

We iterate through the proc array and check the state of each process. We increment the counter of every process that is either in RUNNABLE, RUNNING, SLEEPING,

ZOMBIE mode. We need to take a lock for each process before accessing it to ensure there is no disturbance.

A screenshot of a code editor with a dark theme. The code is in C and is part of a switch statement for 'userInput'. It defines a variable 'cnt\_processes' and iterates through an array of process structures 'proc'. For each process, it acquires a lock, checks if the PID is 0, and then checks if the process state is RUNNABLE, RUNNING, ZOMBIE, or SLEEPING. If so, it increments 'cnt\_processes'. Finally, it releases the lock and prints the total count of active processes.

```
// printf("count of sys call from bootup of OS is %d", cnt_processes);
int cnt_processes = 0;
switch (userInput)
{
    case 0:
        //the total number of active processes (ready, running, waiting, or zombie) in the system
        /* code */
        printf("In switch case 0");
        for (p = proc; p < &proc[NPROC]; p++)
        {
            acquire(&p->lock);
            // printf("p-> id is ; %d", p->pid);
            if (p->pid == 0)
            {
                release(&p->lock);
                break; // no more processes in the array
            }
            // UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE
            // printf(" p-> id is %d", (p->pid));
            // printf("\n");

            if (p->state == RUNNABLE || p->state == RUNNING || p->state == ZOMBIE || p->state == SLEEPING) // verify is sleep.
            {
                cnt_processes++;
            }

            release(&p->lock);
        }

        // printf("cnt active process %d", cnt_processes);
        printf(" \n");
        return cnt_processes;
    // break;
}
```

## 2. To get the total number of system calls made so far by the OS:

Here, we leverage the existing system call function which is called whenever a system call happens in the operating system. We can simply add a counter to keep track of this activity. This variable can be exported with extern command and used to get the system call. One thing we need to make sure is to subtract any additional system calls made due to us.



```

void
syscall(void)
{
    int num;
    struct proc *p = myproc();

    num = p->trapframe->a7;
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        // Use num to lookup the system call function for num,
        // and store its return value in p->trapframe->a0
        p->trapframe->a0 = syscalls[num]();
        // if(l==1){
        //     printf("inc counter cnt_syscalls");
        //     cnt_syscalls++;
        //     p->TotalSysCallsMade++;
        // }
        // if(sizeof(syscalls)/sizeof(syscalls[0])
        // else{
        //     printf("skipping inc counter cnt_syscalls");
        // }

    } else {
        printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
        p->trapframe->a0 = -1;
    }
}

```

### 3. To get the total number of free memory pages in the OS:

For this, we create a function called kllloc.c which helps us get the total number of free pages by iterating a linkedlist which contains pointers to these pages.

```

65 // returns the total free memory pages in the system
66 int getTotalNumberOfFreePages(void){
67     // printf("in kalloc.c .. getting number of free pages");
68     struct run *r;
69     int cnt = 0;
70     acquire(&kmem.lock);
71     r = kmem.freelist;
72
73     while (r)
74     {
75         cnt++;
76         // traverse the list untill next is null
77         if (r->next)
78         {
79             // printf(" cnt is %d ",cnt);
80             r = r->next;
81         }
82         else{
83             break;
84         }
85     }
86     release(&kmem.lock);
87
88     return cnt;
89 }

```

### 4. For all other cases, we just return -1

## Part 2: ProclInfo

The changes made have been mentioned in the '**List of Files Changed**' section.

We create a struct named 'Pinfo' in the user program. The struct has 3 integer member variables -ppid that represents parent process id, syscall\_count which tell the number of syscall that the current process has made, and page-usage indicates the current process's memory size in pages.

```
struct Pinfo {
int ppid;
int syscall_count;
int page_usage;
};
```

We pass the address of the struct to the procinfo system call as an argument. The system call uses argaddr method to read the address, and passes it to the procinfo function call in proc.c.

```
// sys proc info
uint64
sys_procinfo(void){
    printf("Hello.. this is sys PROC info in sysproc.c\n");
    // int option;
    // argaddr(0, &option);
    uint64 p;
    argaddr(0, &p);
    return procinfo((struct Pinfo *) p);
}
```

Now, the procinfo creates a local struct Pinfo, utilizing the myproc function which returns the current process proc. We acquire a lock on the process , get its parent's process id and release the lock. Using this we set Pinfo.ppid.

Since we modified the proc to have a TotalSysCallsMade property, that is incremented every time the process makes a system call.(logic resides in syscall() function in syscall.c file). We access this property from the proc and using this we set the Pinfo.syscall\_count.

Finally, we use the sz property of proc, and the PGSIZE that is defined in riscv.h. And compute the memory pages that are allocated to the current process. And we used the PGROUNDUP to round it off to the nearest integer.

```
/*
* int ppid: the PID of its parent process.
* int syscall_count: the total number of system calls that the current process has
made so
far.(system call number not to be included)
```

```

• int page_usage: the current process's memory size in pages (e.g., 10000 bytes 3
pages)
*/
int
procinfo(struct Pinfo *p){

    struct Pinfo param;
    struct proc * pr = myproc();
    acquire(&wait_lock);
    param.ppid = pr->parent->pid;
    release(&wait_lock);
    param.page_usage = (PGROUNDUP(pr->sz))/PGSIZE;
    param.syscall_count = pr->TotalSysCallsMade;

    uint64 addr;
    argaddr(0, &addr);

    if(copyout(pr->pagetable, addr, (char *) &param, sizeof(param)) < 0){
        return -1;
    }

    return 0;
}

```

```

// Return the current struct proc *, or zero if none.
struct proc*
myproc(void)
{
    push_off();
    struct cpu *c = mycpu();
    struct proc *p = c->proc;
    pop_off();
    return p;
}

```

We use the `argaddr` to retrieve a pointer type input in a syscall. And as the kernel cannot directly write data to the user space memory, we use `copyout()` to do it. We return -1 if `copyout` fails, and if everything is fine we return 0.

# Results

The screenshot displays a debugger interface with a source code editor on the left and a console window on the right. The source code is written in C and includes several system calls and sub-calls. The console window shows the output of the program, including the results of the system calls and sub-calls.

```
15 // = malloc(sizeof(int) * 10);
16 param.ppid = 123123;
17 param.syscall_count = 192;
18 param.page_usage = 12345;
19
20 int n_proc;
21
22 // mem = atoi(argv[2]);
23 n_proc = atoi(argv[3]);
24 printf("I am main(),...\n");
25 printf("\n");
26
27 printf("\n SYSINFO returns value is %d \n", info(n_proc));
28
29 printf("Making PROCINFO call\n");
30 int res = procinfo(&param);
```

The console window shows the following output:

```
$ lab1-sysinfo 0
***** BEGIN *****
I am main(),...
Hello.. this is sys info in sysproc.c
Hi there! I am info call in sysproc.c
user input is 0
In switch case 0
SYSINFO returns value is 3
Making PROCINFO call
Hello.. this is sys PROC info in sysproc.c
param page_usage is 4
param ppid is 2
param syscall_count is 416
res is 0
end of main(),...
***** END *****
$ lab1-sysinfo 1
***** BEGIN *****
I am main(),...
Hello.. this is sys info in sysproc.c
Hi there! I am info call in sysproc.c
user input is 1
In switch case 1
SYSINFO returns value is 945
Making PROCINFO call
Hello.. this is sys PROC info in sysproc.c
param page_usage is 4
param ppid is 2
param syscall_count is 648
res is 0
end of main(),...
***** END *****
$ lab1-sysinfo 2
***** BEGIN *****
I am main(),...
Hello.. this is sys info in sysproc.c
Hi there! I am info call in sysproc.c
user input is 2
In switch case 2
SYSINFO returns value is 32564
Making PROCINFO call
Hello.. this is sys PROC info in sysproc.c
param page_usage is 4
param ppid is 2
param syscall_count is 866
res is 0
end of main(),...
***** END *****
```

The above screenshot shows the execution of both the system calls and all sub calls made as specified in the lab. Apart from that, we also print out additional details such as number of calls made, entrance and exit in a call, failure calls, etc.

## **Member's contribution**

**All the members of the project were more or less equally responsible while working to solve this lab. Here are some specifics:**

**Member 1: Responsible for figuring out how to leverage existing files to get count of pages and fix some bugs.**

**Member 2: Responsible for designing the second system call and understanding the low level specifics of the Operating system and C language terminology**

**Member3: Responsible for writing the parent class and main class functionality and integrating these in the operating system.**