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/1QX7JOfcavmLpGrVC04xPe4l6_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.

```
Makefile
# details:
# http://www.gnu.org/softwar
.PRECIOUS: %.o

UPROGS=\
	$U/_cat\
	$U/_echo\
	$U/_forktest\
	$U/_grep\
	$U/_init\
	$U/_kill\
	$U/_ln\
	$U/_ls\
	$U/_mkdir\
	$U/_rm\
	$U/_sh\
	$U/_stressfs\
	$U/_usertests\
	$U/_grind\
	$U/_wc\
	$U/_zombie\
	$U/_lab1-sysinfo\
```

2. Kernel Files
    a. Syscall.h
       Here we declare the two system calls that we have added to our OS, sysinfo and sysprocinfo

ernel > syscall.h

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

```c
[SYS_close]   sys_close,
[SYS_procinfo]    sys_procinfo,
[SYS_info]    sys_info,
};

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, call it,
    // and store its return value in p->trapframe->a0
    p->trapframe->a0 = syscalls[num]();
    // if(1==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;
  }
}
```

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.

```c
// sys info
uint64
sys_info(void){
  printf("Hello.. this is sys info in sysproc.c\n");
  int option;

  argint(0, &option);
  return info(option);
}

// 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);
}
```

d. Proc.c

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

```c
// nextpid = nextpid + 1;

// printf("count of sys call from bootup of OS is %d", cnt_syscalls);
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) //
    {
      cnt_processes++;
    }

    release(&p->lock);
  }

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

```c
kernel > C proc.c
}

/*
* 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){
// printf("in proc.c procinfo %d", p);
// printf("\n...1 in proc.c p->ppid  %p", ((int*)(&p->ppid)));
// printf("\n...in proc.c p->ppid  %d", (int*)(*p).ppid);
// printf("\n...2 in proc.c p->page_usage  %d", ((int)(*(int*)(p+ sizeof(int)))));
// printf("\n...3 in proc.c p->page_usage  %d", (int*)(&p->syscall_count));

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;
}
// todo
// printf("p->ppid is %d -- p->syscall_count is %d -- p->page_usage is %d ",(&p->ppid),(p->sys
```

e. Proc.h

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

kernel > proc.h

```c
  /* 264 */ uint64 t4;
  /* 272 */ uint64 t5;
  /* 280 */ uint64 t6;
};

enum procstate { UNUSED, USED, SLEEPING, RUNNABLE, R

// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleep
  int killed;                  // If non-zero, have
  int xstate;                  // Exit status to be
  int pid;                     // Process ID

  int TotalSysCallsMade;
  // wait_lock must be held when using this:
  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.

syscall.h | lab1-sysinfo.c | proc.c | defs.h

kernel > kalloc.c

```c

// returns the total freee memory pages in the system
int getTotalNumberOfFreePages(void){
  // printf("in kalloc.c .. getting number of free pages");
  struct run *r;
  int cnt = 0;
  acquire(&kmem.lock);
  r = kmem.freelist;

  while (r)
  {
    cnt++;
    // traverse the list untill next is null
    if (r->next)
    {
      // printf(" cnt is %d ",cnt);
      r = r->next;
    }
    else{
      break;
    }
  }
  release(&kmem.lock);

```

g. Defs.h

We define the two new calls for proc.c here, cnt_syscalls and procdump

syscall.h | lab1-sysinfo.c | proc.c | defs.h | usys.pl | user.h

kernel > defs.h

```c
void            procinit(void);
void            scheduler(void) __attribute__((noreturn));
void            sched(void);
void            sleep(void*, struct spinlock*);
void            userinit(void);
int             wait(uint64);
void            wakeup(void*);
void            yield(void);
int             either_copyout(int user_dst, uint64 dst, void *src, uint64 len);
int             either_copyin(void *dst, int user_src, uint64 src, uint64 len);
int             info(int userInput); // sysinfo internally calls info() in proc.c
int             procinfo(struct Pinfo *p); // sysinfo internally calls procinfo() in proc.
extern int      cnt_syscalls;
void            procdump(void);

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

// spinlock.c
void            acquire(struct spinlock*);
int             holding(struct spinlock*);
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 > user.h

```c
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int info(int);
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 > lab1-sysinfo.c

```c
// #include "kernel/Defs.h"
// #include <stdlib.h>
// #include <stdio.h>
struct Pinfo {
int ppid;
int syscall_count;
int page_usage;
};
int main(int argc, char *argv[]){
    printf("\n ********** BEGIN *********\n");
    struct Pinfo param ;
    // = malloc(sizeof(Pinfo));
    param.ppid =123123;
    param.syscall_count=192;
    param.page_usage=12343;

    int n_proc;

    // mem = atoi(argv[2]);
    n_proc = atoi(argv[1]);
    printf(" i am main()....");
    printf("\n");

    printf("\n SYSINFO returns value is  %d \n", info(n_proc));

    printf("\nMaking PROCINFO call\n");
    int res = procinfo(&param);
    printf("\nparam page_usage is %d", param.page_usage);
    printf("\nparam ppid is %d", param.ppid);
    printf("\nparam syscall_count is %d", param.syscall_count);
    printf("\nres is %d", res);
    printf("\nend of main()...\n");
    printf("\n ********* END **********\n");
    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.

```
// printf("count of sys call from bootup of OS is %d", cnt
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;
```

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.

```c
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(1==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 klloc.c which helps us get the total number of free pages by iterating a linkedlist which contains pointers to these pages.

```c
// returns the total freee memory pages in the system
int getTotalNumberOfFreePages(void){
  // printf("in kalloc.c .. getting number of free pages");
  struct run *r;
  int cnt = 0;
  acquire(&kmem.lock);
  r = kmem.freelist;

  while (r)
  {
    cnt++;
    // traverse the list untill next is null
    if (r->next)
    {
      // printf(" cnt is %d ",cnt);
      r = r->next;
    }
    else{
      break;
    }
  }
  release(&kmem.lock);

  return cnt;
}
```

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

Part 2: ProcInfo

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 PGGROUNDUP to round it off to the nearest integer.

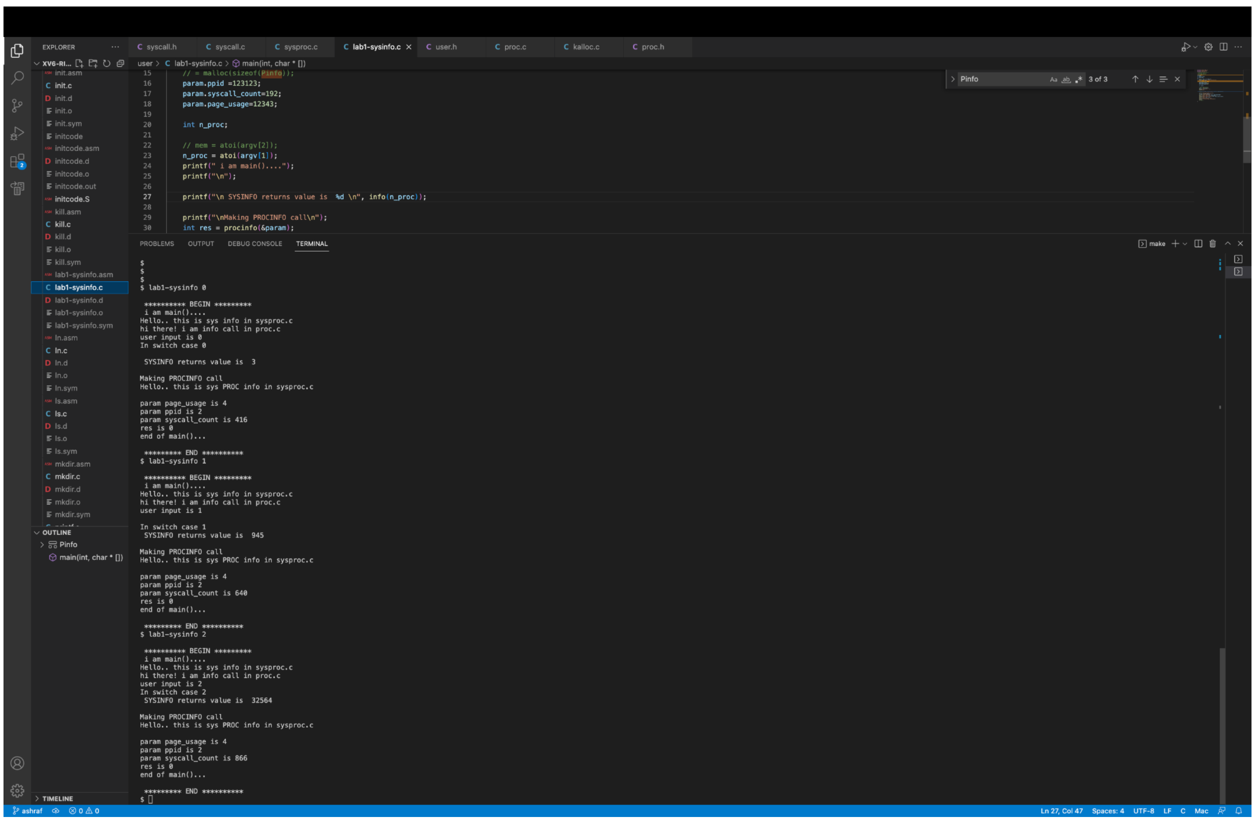
```
/*
* int ppid: the PID of its parent process.
• int syscall_count: the total number of system calls that the current process has
made so
far.(system call number not to be included)
```

```
• 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 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.