Lab 1 - Report

Karthik Harpanahalli - 862254197
Varun Sapre - 862255166

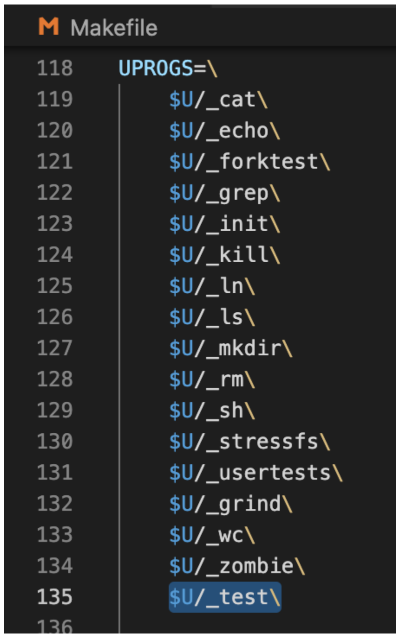
a) List of all files modified:

10 files were modified in total

1. Makefile
2. kernel/defs.h
3. kernel/proc.c
4. kernel/proc.h
5. kernel/syscall.c
6. kernel/syscall.h
7. kernel/sysproc.c
8. user/test.c
9. user/user.h
10. user/usys.pl

b) A detailed explanation on what changes you have made and screenshots showing your work and results

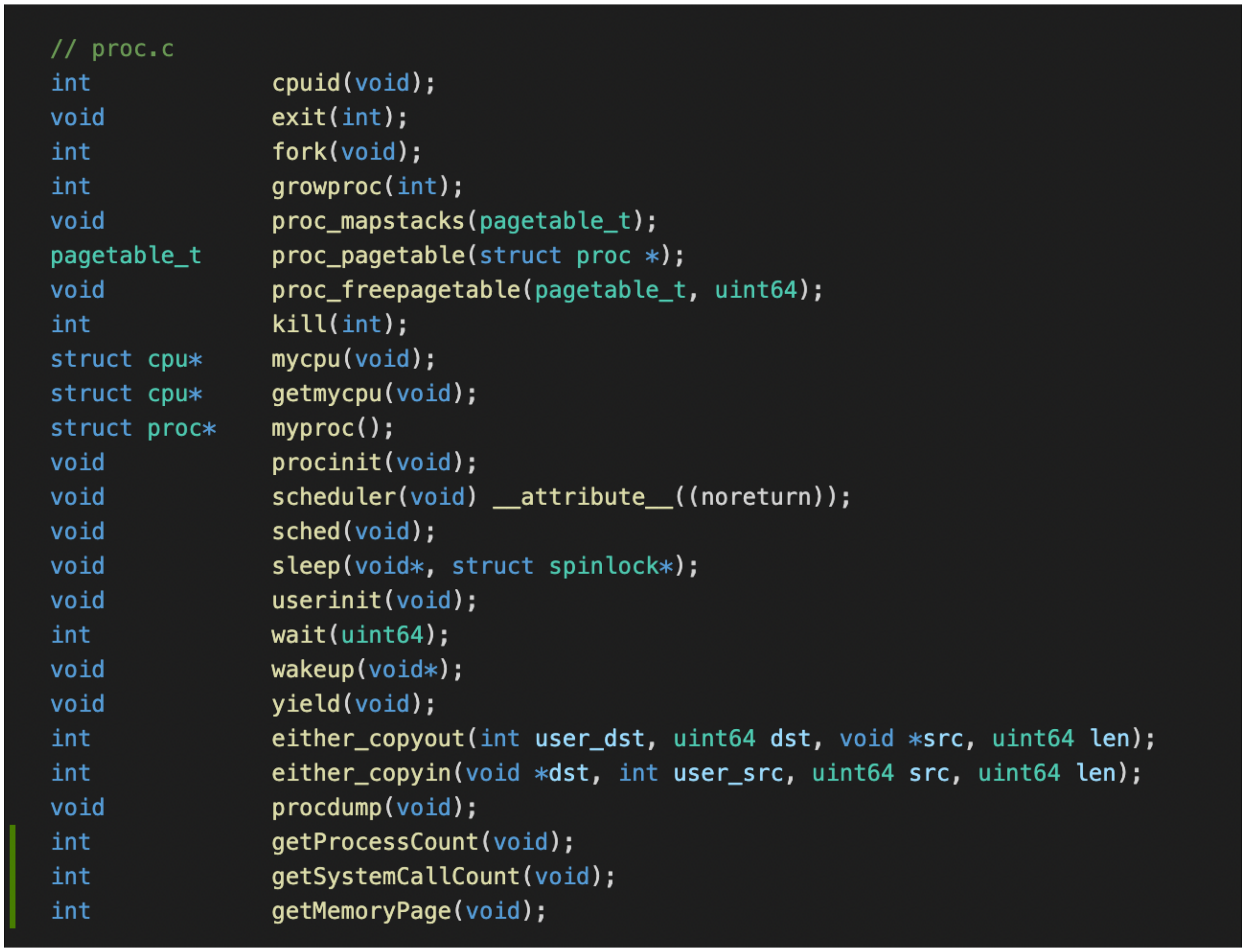
1) Makefile



```
Makefile
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/_test\
```

We add our user program in Makefile to make our user program available for the xv6 source code compilation.

2) kernel/defs.h

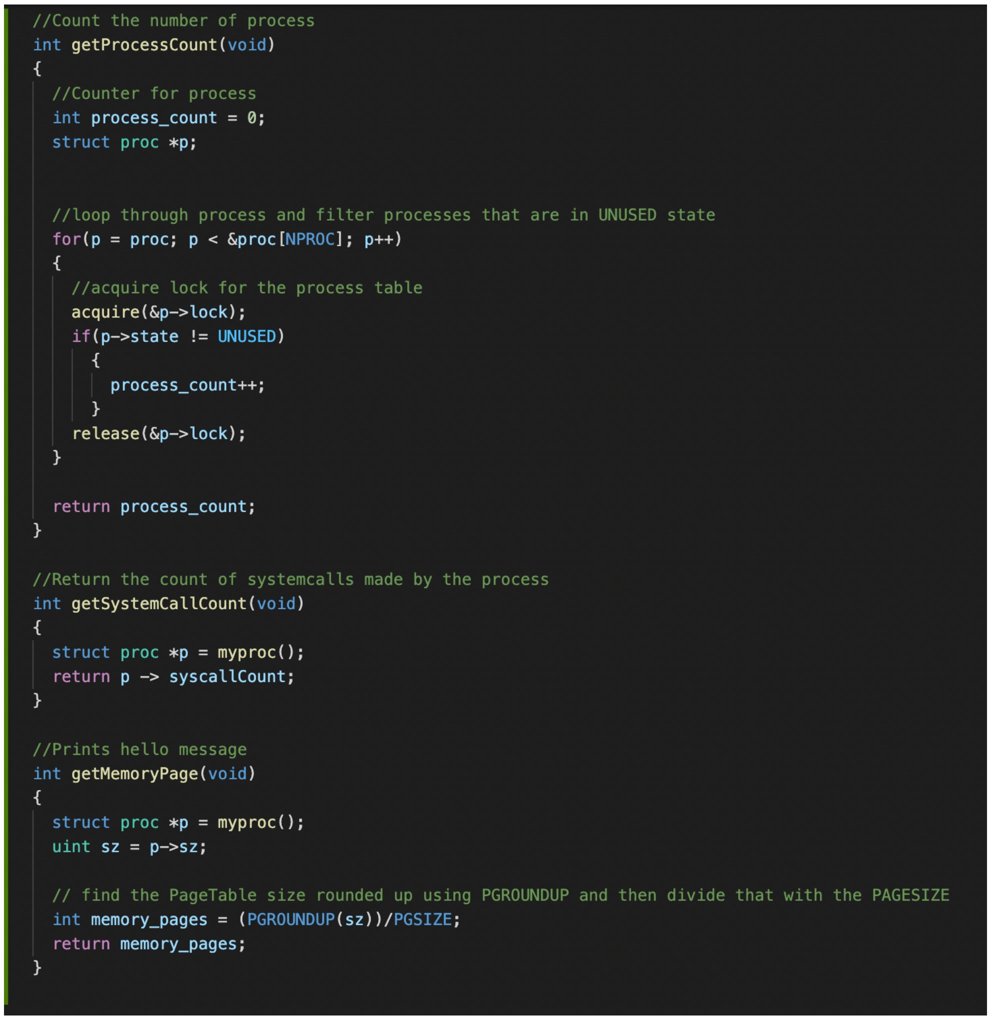


```
// proc.c
int             cpuid(void);
void            exit(int);
int             fork(void);
int             growproc(int);
void            proc_mapstacks(pagetable_t);
pagetable_t     proc_pagetable(struct proc *);
void            proc_freepagetable(pagetable_t, uint64);
int             kill(int);
struct cpu*     mycpu(void);
struct cpu*     getmycpu(void);
struct proc*    myproc();
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);
void            procdump(void);
int             getProcessCount(void);
int             getSystemCallCount(void);
int             getMemoryPage(void);
```

We add our procedures in the defs.h file to make them accessible. We define three methods to achieve each required functionality.

1. getProcessCount(void) returns the count of the processes in the system.
2. getSystemCallCount(void) returns the count of the total number of system calls that the current process has made so far; it doesn't include the current info syscall in the count.
3. getMemoryPage(void) returns the number of memory pages the current process is using.

3) kernel/proc.c



```
//Count the number of process
int getProcessCount(void)
{
  //Counter for process
  int process_count = 0;
  struct proc *p;


  //loop through process and filter processes that are in UNUSED state
  for(p = proc; p < &proc[NPROC]; p++)
  {
    //acquire lock for the process table
    acquire(&p->lock);
    if(p->state != UNUSED)
      {
        process_count++;
      }
    release(&p->lock);
  }

  return process_count;
}

//Return the count of systemcalls made by the process
int getSystemCallCount(void)
{
  struct proc *p = myproc();
  return p -> syscallCount;
}

//Prints hello message
int getMemoryPage(void)
{
  struct proc *p = myproc();
  uint sz = p->sz;

  // find the PageTable size rounded up using PGROUNDUP and then divide that with the PAGESIZE
  int memory_pages = (PGROUNDUP(sz))/PGSIZE;
  return memory_pages;
}
```

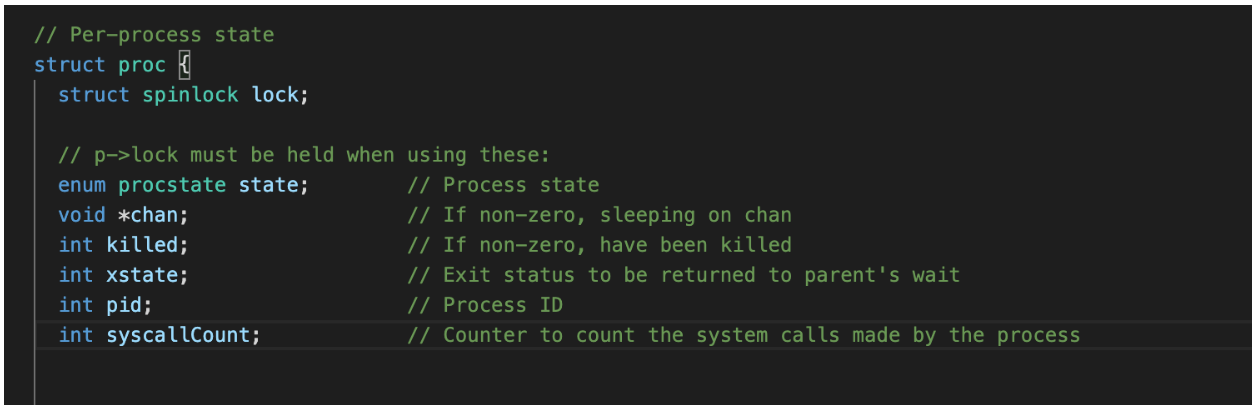
This is the implementation of the logic that achieves the above mentioned functionality.

1. getProcessCount(void) : We loop through all the available processes in the process table and count the processes that are not marked UNUSED. This gives the count of the processes in the system. In each loop we acquire and release a lock for each process.

2. getSystemCallCount(void) : We have modified the struc of proc to include a counter to count the number of syscalls made by the process. The counter logic is handled in the syscall method. We return the counter in this function.

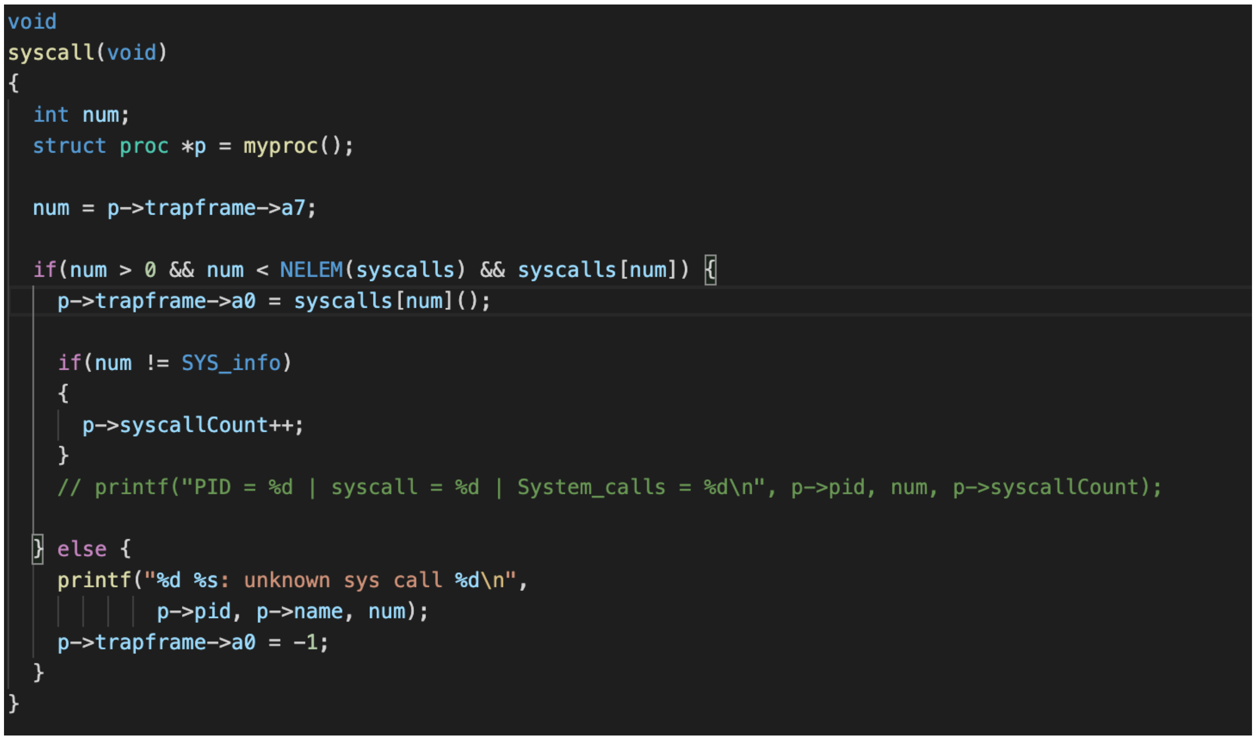
3. getMemoryPage(void) : We use the inbuilt function PGROUNDUP which takes the current size of the process and rounds it up to the next highest PAGESIZE multiple. Upon dividing this value with the PAGESIZE constant, we can determine the number of pages being used by the process.

4) kernel/proc.h


```
// 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, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID
  int syscallCount;            // Counter to count the system calls made by the process
```

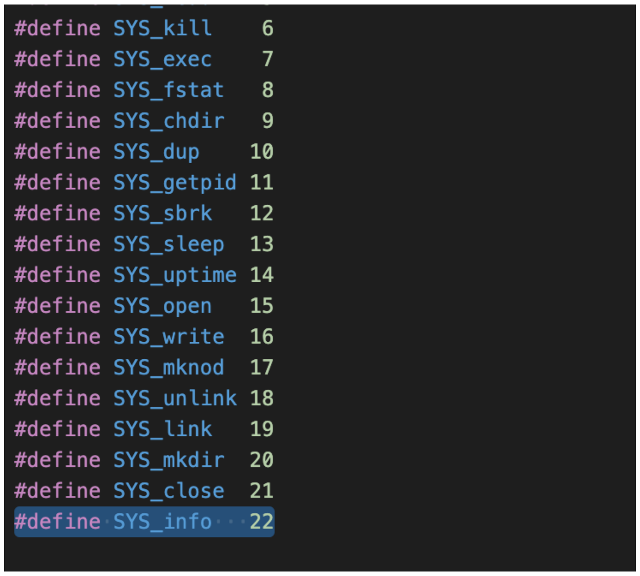
We are modifying the proc structure to include a variable that holds the count of the syscalls made by the associated process. This is required to implement the second functionality.

5) kernel/syscall.c


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

  num = p->trapframe->a7;

  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();

    if(num != SYS_info)
    {
      p->syscallCount++;
    }
    // printf("PID = %d | syscall = %d | System_calls = %d\n", p->pid, num, p->syscallCount);

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

For the second functionality, we are adding a small modification to the syscall function where we keep a count of all syscalls made by the current process. The count is stored in the proc struc. We do not count the info syscall itself by checking the current syscall being called. We disregard the syscall and keep count of every other syscall.

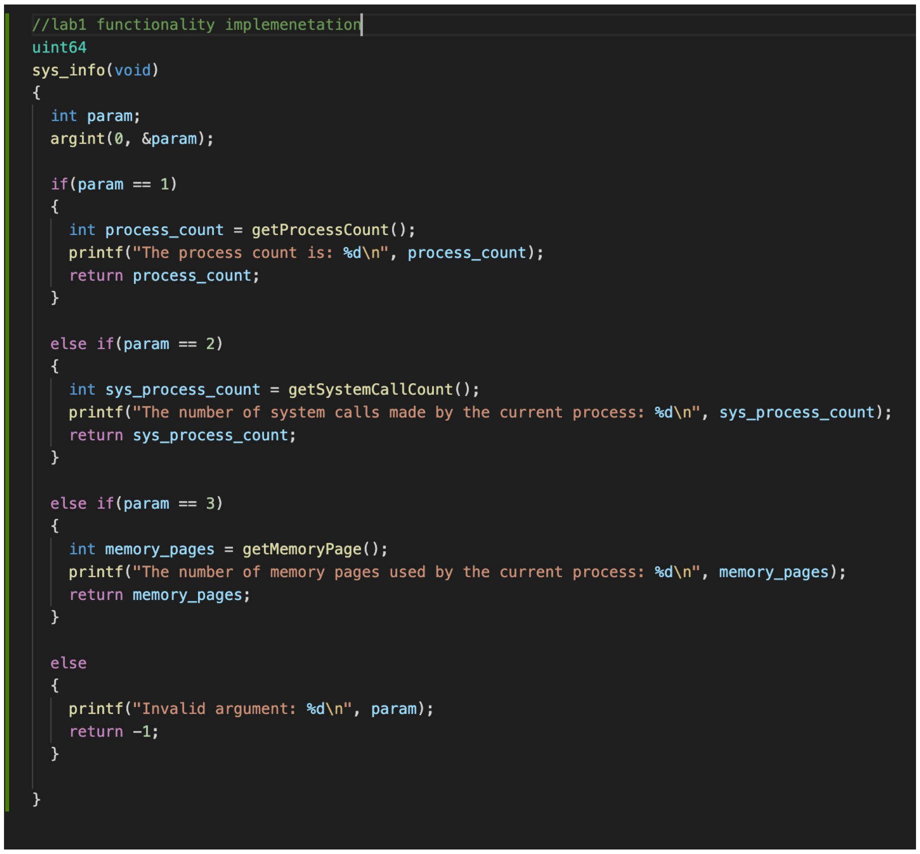
6) kernel/syscall.h



```
#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
```

We are defining our info syscall as 22 which is used to write in a7 register to identify the required syscall.

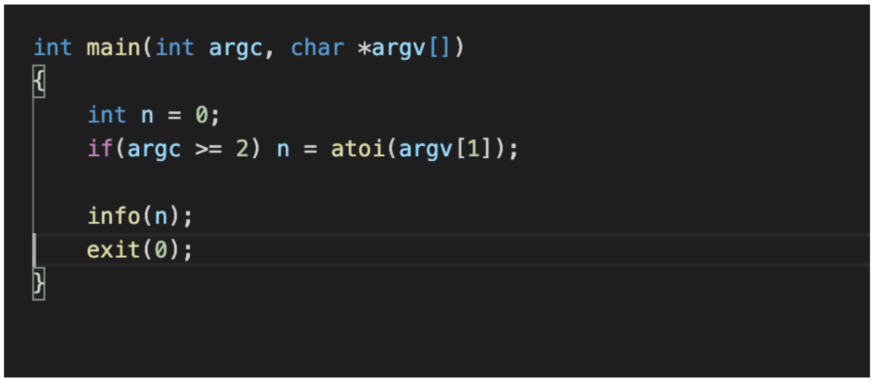
7) kernel/sysproc.c



```
//lab1 functionality implemenetation
uint64
sys_info(void)
{
  int param;
  argint(0, &param);

  if(param == 1)
  {
    int process_count = getProcessCount();
    printf("The process count is: %d\n", process_count);
    return process_count;
  }

  else if(param == 2)
  {
    int sys_process_count = getSystemCallCount();
    printf("The number of system calls made by the current process: %d\n", sys_process_count);
    return sys_process_count;
  }

  else if(param == 3)
  {
    int memory_pages = getMemoryPage();
    printf("The number of memory pages used by the current process: %d\n", memory_pages);
    return memory_pages;
  }

  else
  {
    printf("Invalid argument: %d\n", param);
    return -1;
  }

}
```

This method handles the actual logic and dispatches the required procedure call written in proc.c by reading the param value. Based on the param value, it calls the procedure that returns the value which it then prints to the console.

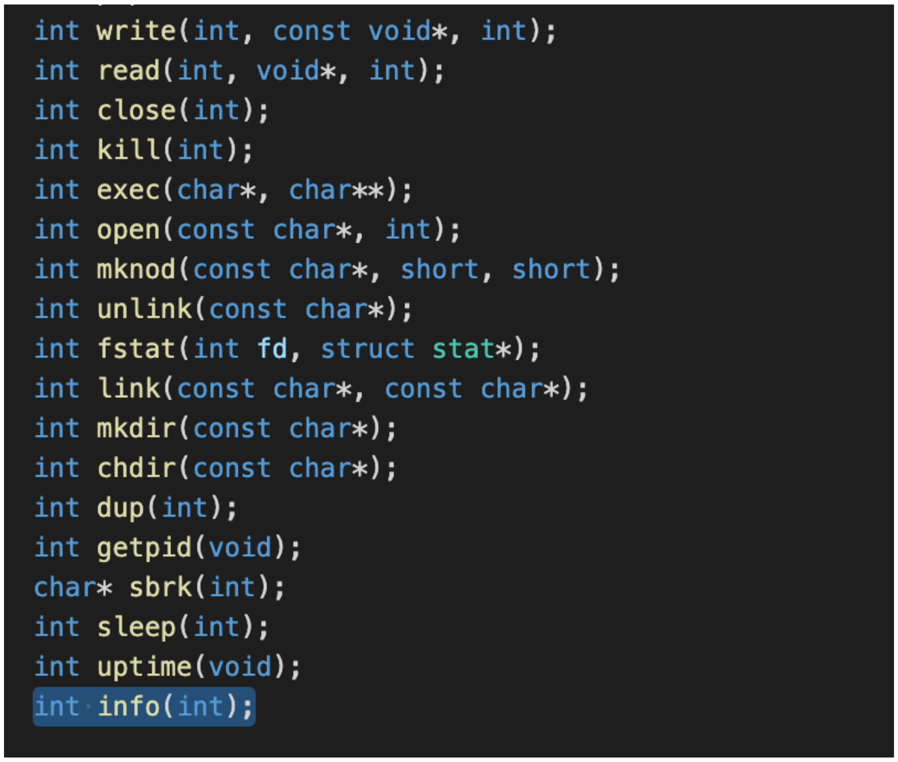
8) user/test_info.c



```
int main(int argc, char *argv[])
{
    int n = 0;
    if(argc >= 2) n = atoi(argv[1]);

    info(n);
    exit(0);
}
```

This is the user application that is the starting point for our implementation. It calls info(n) which dispatches the appropriate syscall.

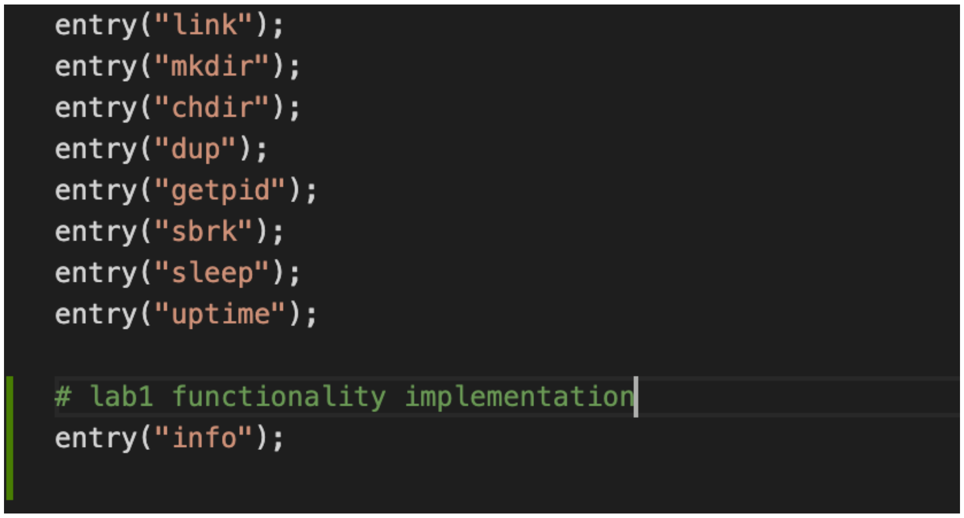
9) user/user.h



```
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int info(int);
```

We define our syscall here to provide all the available syscalls to the user that can be invoked.

10) user/usys.pl

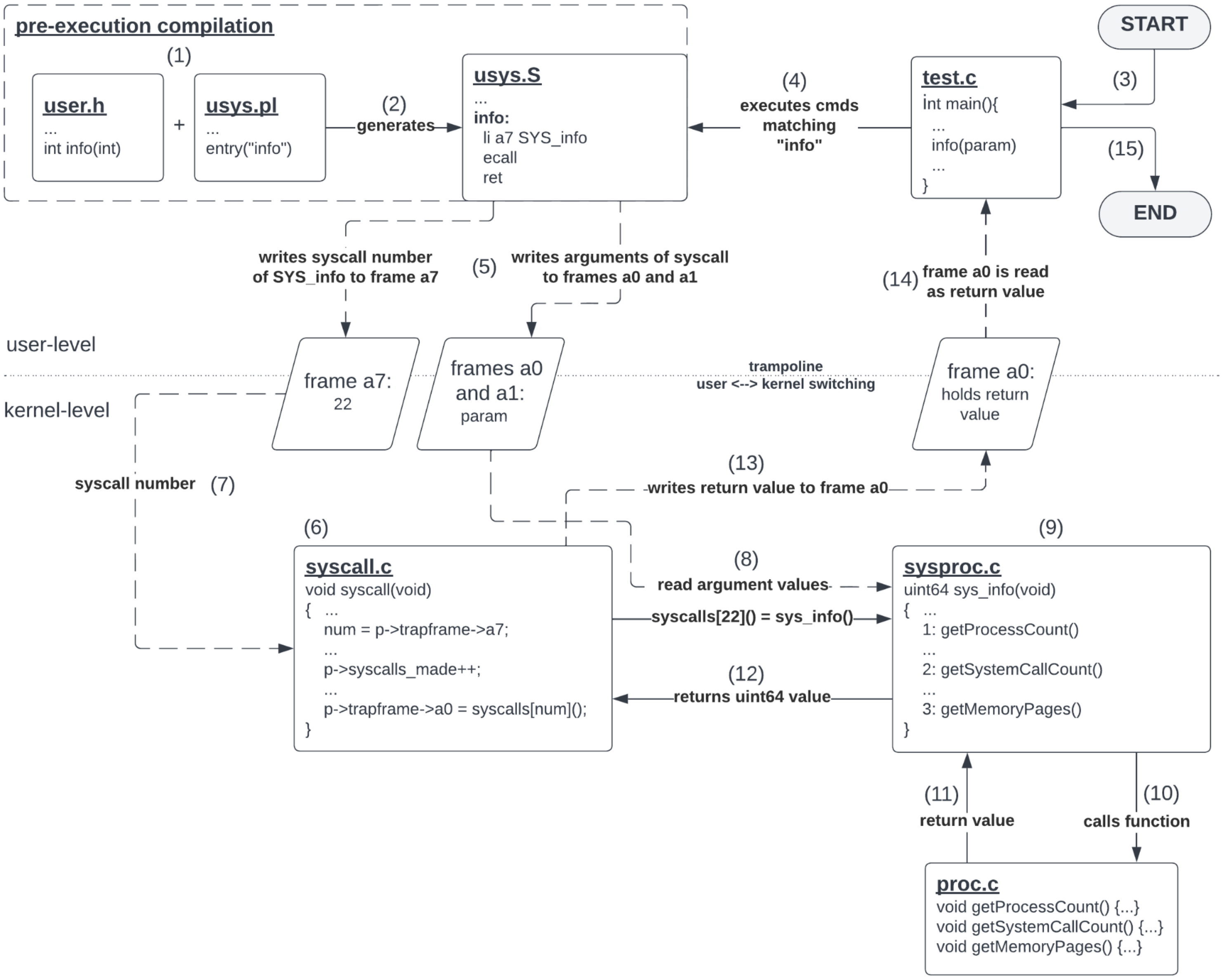


```
entry("link");
entry("mkdir");
entry("chdir");
entry("dup");
entry("getpid");
entry("sbrk");
entry("sleep");
entry("uptime");

# lab1 functionality implementation
entry("info");
```

We register our stub to identify the appropriate syscall to be dispatched from the user space.

c) A detailed description of XV6 source code (including your modifications) about how the info system call is processed, from the user-level program into the kernel code, and then back into the user-level program.

Here's a brief flow chart showing how the "info" system-call works. Please follow the numbers (1) to (15). Each step is explained in detail after the flow chart.



(1) and (2) Pre-execution Compilation takes place. The combination of “info” function registration in user.h and usys.pl generates a file “usys.S” during compilation. “usys.S” holds a binding between the user-level “info” function and the kernel-level “sys_info” function by holding the syscall number SYS_info.

(3) Execution of user-level program “test.c” starts. The main function calls “info(param)” with the option argument.

(4) Execution of commands in “usys.S”. Since we have registered the “info” function to execute a system call, the OS executes the commands present in “usys.S” stub for “info”.

(5) The commands in usys.S first write the syscall number to register a7, then “ecall” runs the system call (point 6). The arguments of the info function call are written to registers starting from a0.

(6) and (7) Here, we switch execution to the kernel-space. Inside syscall.c, the function “syscall()” reads register a7 to know which system call to execute. It then calls the function using “syscalls[num]()”, which in this case is sys_info().

(8) and (9) The function definition of sys_info() is present in sysproc.c. In the function, we read the argument passed to the system call by using “argint(0, ..)”. This reads the value inside register a0 as an integer. Depending on the value of this, we run 1, 2 or 3 functions.

(10) and (11) The kernel function gets called and executes the logic of counting processes or counting system calls or counting memory pages. The return value (if any) is passed back to sys_info.

(12) and (13) The return value is passed back to syscall() which is then written to register a0.

(14) Here, we switch execution back to the user-space and the value of register a0 is passed to the user-level program.

(15) Execution of the user-level program ends.

d) A brief summary of the contributions of each member

Both team members have implemented all customizations needed for the “info” syscall equally.