# CS 140 Lab Report 3

Dale Sealtiel T. Flores
2023-11373
THX/WXY

1. For each subitem below, show all relevant changes made (with corresponding filename) via code screenshots or snippets, and briefly describe what each change does. Ensure that your changes are properly commited and pushed to your Github Classroom repository. Note that none of the system calls below should print anything on the screen.

   (a) Create a new system call `opencount` with syscall number `24` that returns the number of times the `open` syscall has been invoked successfully (i.e reached the end of its syscall handler without returning `-1` prematurely) since xv6 startup as a `uint64`. Include a corresponding user mode wrapper function.


**Figure 1:** `syscall.h` change

This change adds the is required to assign the syscall code 24 to the new syscall `sys_opencount`.

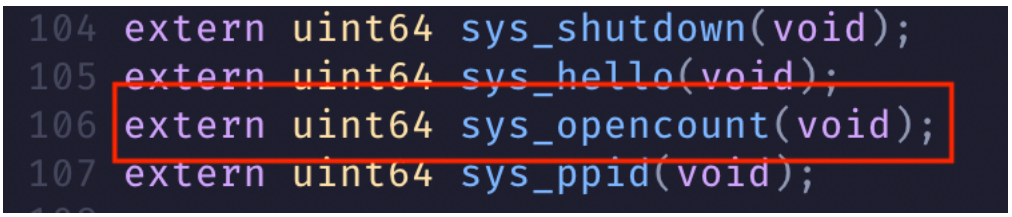



**Figure 2:** `syscall.c` change 1

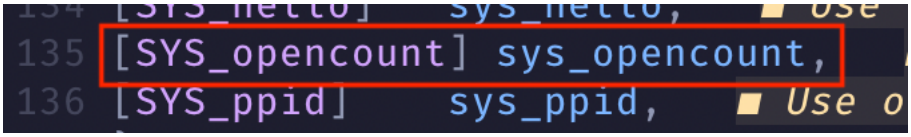



**Figure 3:** `syscall.c` change 2

These are changes to add the `sys_opencount` function to the syscall table.

```
18
19 uint64 open_count = 0;
20
```

**Figure 4:** `sysfile.h` change 1

```
511 uint64
512 sys_opencount(void)
513 {
514   return open_count;
515 }
```

**Figure 5:** `sysfile.h` change 2

```
372   open_count++;
373
374   return fd;
375 }
```

**Figure 6:** `sysfile.c` change 3

These are the changes to implement the `sys_opencount` function. `open_count` is a variable that is incremented each time the `sys_open` function is called and successfull (change 3 is added at the end of the functino). The `sys_opencount` function simply returns the value of `open_count`.

```
41 entry("opencount");
```

**Figure 7:** `usys.pl` change

This change is necessary to add the user mode wrapper function for `opencount`.

```
27 int opencount(void);
```

**Figure 8:** `user.h` change

This change allows `C` programs to call `opencount` as a regular `C` function despite it being defined in assembly.

(b) Examine how `sys.uptime` in `kernel/sysproc.c` is able to return the number of ticks since xv6 startupt to the user process, and examine how `sys_kill` in `kernel/sysproc.c` is able to retrieve the value passed in `a0` when `ecall` is executed. Create a new system call `ppid` with syscall number 25 that takes a PID as its argument, then returns a `uint64` corresponding to the parent PID of the process with the given PID, or `-1` if the PID is invalid. Include a corresponding user mdoe wrapper function. No need to perform locking.


**Figure 9:** `syscall.h` change

This change adds the is required to assign the syscall code 25 to the new syscall `sys_ppid`.

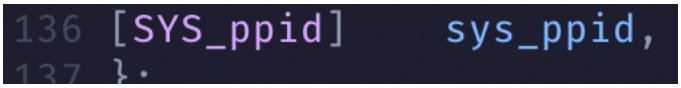
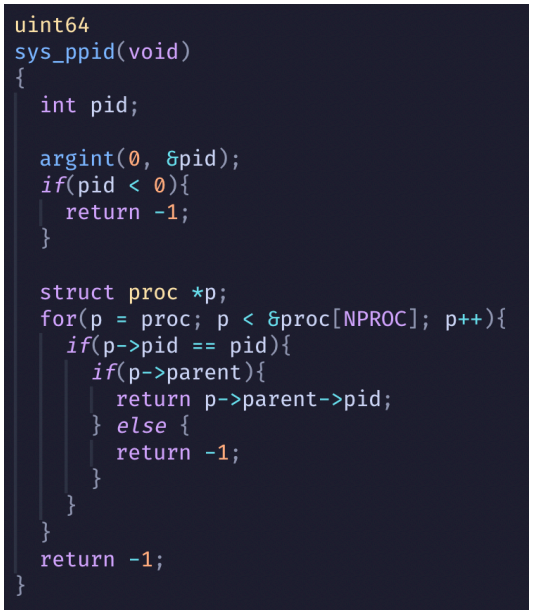



**Figure 10:** `syscall.c` change 1


**Figure 11:** `syscall.c` change 2

These are changes to add the `sys_ppid` function to the syscall table.



```
uint64
sys_ppid(void)
{
  int pid;

  argint(0, &pid);
  if(pid < 0){
    return -1;
  }

  struct proc *p;
  for(p = proc; p < &proc[NPROC]; p++){
    if(p->pid == pid){
      if(p->parent){
        return p->parent->pid;
      } else {
        return -1;
      }
    }
  }
  return -1;
}
```

**Figure 12:** `sysproc.h` change 1

This is the implementation of the `sys_ppid` function. It gets the argument passed to it using `argint` and then searches for the `ppid` of the process from the `proc` array.


**Figure 13:** `usys.pl` change

This change is necessary to add the user mode wrapper function for `ppid`.

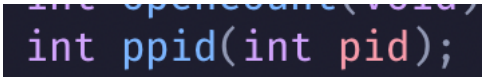



**Figure 14:** `user.h` change

This change allows `C` programs to call `ppid` as a regular `C` function despite it being defined in assembly.

2. Regarding the change described in Section 2.2 regarding the syscalls invoked on startup, show all relevant changes made (with corresponding filename) via code screenshots or snippets, and briefly describe what each change does.
   Additionally, take a screenshot of the output introduced by your changes when starting xv6 up, and annotate it so it shows the syscall name of each syscall number shown (you may also print the name programmatically). Ensure the name of the invoking process is shown per syscall.

**Answer:**

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

**Figure 15:** `syscall.c` change

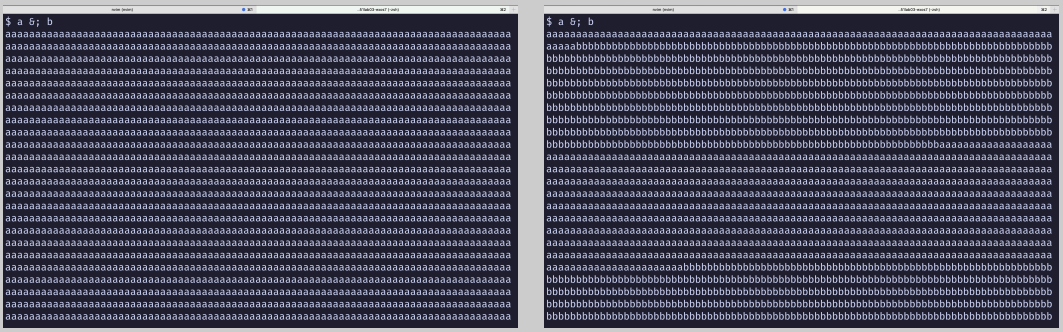
```
xv6 kernel is booting

syscall 15: init
syscall 10: init
syscall 10: init
isyscall 16: init
nsyscall 16: init
isyscall 16: init
tsyscall 16: init
:syscall 16: init
 syscall 16: init
ssyscall 16: init
tsyscall 16: init
asyscall 16: init
rsyscall 16: init
tsyscall 16: init
isyscall 16: init
nsyscall 16: init
gsyscall 16: init
 syscall 16: init
ssyscall 16: init
hsyscall 16: init

syscall 16: init
syscall 1: init
syscall 7: sh
syscall 15: sh
syscall 21: sh
$ syscall 16: sh
```

**Figure 16:** change output

3. Show the screenshot taken in Section 2.3.2, compare it with the output of the same command in Section 2.3.1, then explain why running `hello` causes the output discrepancy.

**Answer:** In 2.3.2, we added lines of code that runs whenever we call the process with syscall code 23 (which is `sys_hello`). In the changes, it disables timer interrupts by running the line `asm volatile("csrw sie, %0" : : "r" (0x200));` Since we disabled timer interrupts, it also disables the scheduler. This means that whoever runs first, in my case the user program `a` runs first, it will run until it terminates but since we programmed our `a` to run indefinitely, it will never terminate hence it will forever be printing the letter a.

**Table 1:** Output of 2.3.1 vs Output of 2.3.2



4. Show the screenshot taken in Section 2.3.3, describe the behavior of `hello` from startup to termination, then explain what causes `hello` to behave in atypical manner.

**Answer:** The `hello` program is unable to run the `printf("After disabling timer interrupts\n");` line because of the previous line `asm volatile("csrw sie, %0" : : "r" (0x200));` this line should be called in kernel mode but it is called in user mode causing a trap and terminating the program.

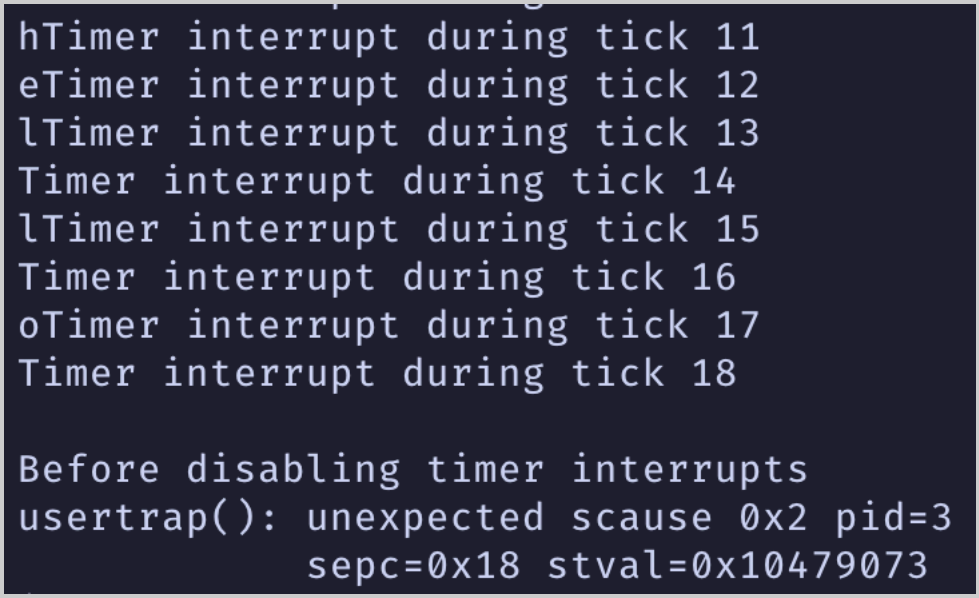



**Figure 17:** Output of 2.3.4