

CS 503 Fall 2022: Lab 2

Suraj Aralihalli

September 2022

3. Trapped system call implementation

3.5 Testing

Wrapper Function

1. Pushes arguments into the user stack.
2. Moves the system call number into %eax
3. Raises interrupt \$46
4. Moves the %eax or %ax value into local variable
5. Pops the arguments inserted into the user stack
6. Returns the local variable

Xint46

1. Saves %ebp
2. Saves general purpose registers except %eax
3. Point %cx and %dx registers to 2nd Kernel data segment entry in GDT
4. Fetch address of kernel stack from kstack[] for the current process
5. Push user stack %esp onto kernel stack
6. Copy syscall arguments into kernel stack
7. Call syscall based on %eax
8. Pop syscall arguments from kernel stack
9. Point %cx and %dx registers back to 1st Kernel data segment entry in GDT
10. Restore user stack %esp and other general purpose registers
11. Iret with %eax populated with syscall return value

Note: %ebp is not pushed into the kernel stack in my implementation because it is not being updated or used in kernel stack (updated and restored implicitly when system call is performed in the kernel stack). Even the instructions do not specifically ask us to push %ebp. However, it can be pushed into kernel stack and restored while exiting to be consistent across stacks (See implementation).

Testing

Note: All the testcases can be found in system/mytests.c

```
#####
Greetings from Suraj Aralihalli!
Username: saraliha
#####

###test1###

PASS oldPri!=30
PASS getprio(pid)==35

###test2###

PASS address2:0x15a1e8 - address1:0x15a1e0 == 8
PASS address3:0x15a1f0 - address2:0x15a1e8 == 8
PASS address4:0x15a210 - address4:0x15a1f0 == 8

###test3###

PASS getpid()==getpidx()

###q4test1###

id:0, iter:999999, userCpuTime:30, endTime:3161
id:1, iter:995809, userCpuTime:30, endTime:3162
id:2, iter:995809, userCpuTime:30, endTime:3163
id:3, iter:995797, userCpuTime:30, endTime:3164

###q4test2###

id:0, iter:999999, userCpuTime:30, endTime:9299
id:1, iter:995851, userCpuTime:30, endTime:9300
id:2, iter:107666, userCpuTime:30, endTime:9301
id:3, iter:107650, userCpuTime:30, endTime:9302

###test5###

PASS usercpu(currpid):226==usercpux(currpid):226

###test6###

PASS cpuUtil:4 != 100
```

Figure 1: Tests results (doesnot include all tests executed)

I have created a file named mytests.c (Please refer file system/mytests.c) that contains testcases to gauge the correctness of lab2 including the trapped system calls. Some of the tests are, check if

getpidx functions correctly. To test this I created a process (test3() in mytests.c) that checks whether (getpid()==getpidx()). If this condition fails, it means the assembly code for **_Xint46** has bugs. To check if chprio is functioning properly, I check if the value returned by chprio() matches with the initial priority that was set while creating the process. Additionally, I also fetched the priority of the process after performing chprio() using getprio() and compared it with the new priority of the process (See test1() in mytests.c). To test getmemx(), I performed getmemx() and getmem() consecutively. If the functions are working properly I would expect the arithmetic difference in the addresses returned by these functions to be equal to the argument passed when the argument is a multiple of 8 (See test2() in mytests.c). Other tests include:

1. Check if chprio returns -1 when pid is invalid (To make sure behaviour of chprio is identical to chprio even when invalid arguments are used).
2. Check whether wrapper functions perform a graceful exit of kernel stack.
 - (a) The address returned by kstack[pid] should remain the same for the process before the system call, after the system call and through out the lifetime of the process.
 - (b) Wrapper calls like getpidx, chprio should return valid results even when its called multiple times within the process.

4. User code CPU usage monitoring and process behavior

I have used the below formula to compute CPU Utilization. It is the percentage of time that is not idle.

$$CPUUtilization = 100 - (usercpu(nullprocess) * 100 / vfineclkcounter)$$

To test and assess correctness of your implementation I have considered the following scenarios:

1. Testcase 1: Create 4 processes of equal priority each performing the same while loop.
2. Testcase 2: Make two of the above four processes call getprio() within the while loop

Testcase 1

```
void loopALot(int id)
{
    int i=0;
    while(i!=999999 && breakLoop==0)
    {
        i++;
    }
    if(breakLoop==0)
    {
        breakLoop = 1;
    }

    iter[id] = i;
    userCpuTime[id] = usercpu(currpid);
    return;
}
```

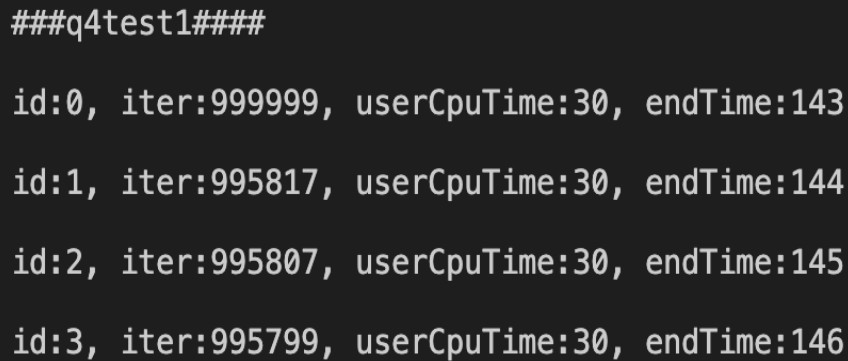
```

}

void loops1()
{
    resume(create(loopALot,1024,40,"loop1", 1, 0));
    resume(create(loopALot,1024,40,"loop2", 1, 1));
    resume(create(loopALot,1024,40,"loop3", 1, 2));
    resume(create(loopALot,1024,40,"loop4", 1, 3));
}

void q4test1()
{
    resume(create(loops1,1024,45,"q4test1", 0));
    sleep(5);
    int i=0;
    while(i!=4)
    {
        kprintf("\nid:%d, iter:%d, userCpuTime:%d\n",i,iter[i],userCpuTime[i]);
        i++;
    }
}

```



```

###q4test1###

id:0, iter:999999, userCpuTime:30, endTime:143
id:1, iter:995817, userCpuTime:30, endTime:144
id:2, iter:995807, userCpuTime:30, endTime:145
id:3, iter:995799, userCpuTime:30, endTime:146

```

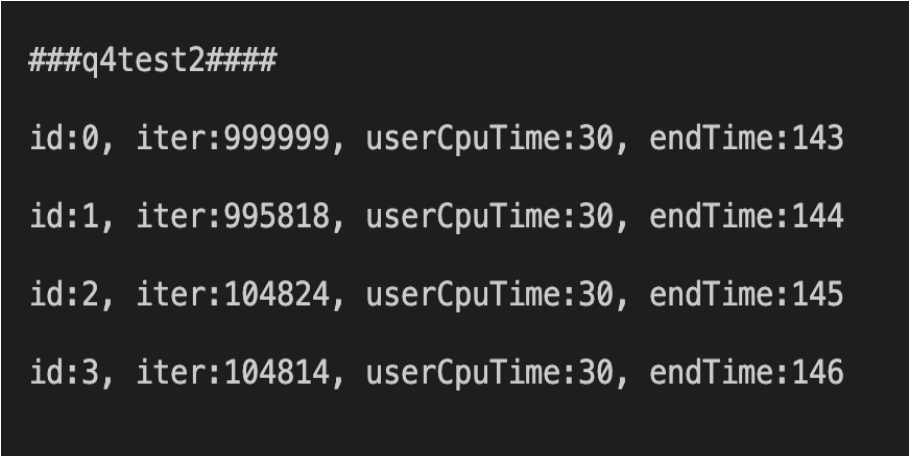
Figure 2: Testcase1 with 4 processes running simple while loops

I have created a parent process named q4test1 (file: system/mytests.c) that spawns 4 child processes and has a priority higher than all other processes. Since this process has a higher priority than the child processes, child processes cannot start. However, I have added a `sleep(5)` in my parent process that would force the parent to context switch to child processes and they all start at about the same time (will differ by quantum). I have used a global variable called `breakLoop` which is initialized to 0 as a mechanism to ensure all my child processes terminate at about the same time. The first process to complete 999999 iterations will set the `breakLoop` to 1 and store the iteration count and `usercpu()` in a global array. Consequently other processes will terminate and also store

their iteration count and `usercpu()`. After the context is switched back to parent after `sleep(5)`, the parent process prints the output (See Figure 2). From the figure, it is clear that all the 4 processes run for similar number of iterations. The `userCpu` time for all the processes is 30ms, indicating they ran for the same amount of time. Endtime (i.e `vfineclkcounter`) shows that the processes terminated at about same time.

Testcase2

```
void loopWithGetPrio(int id)
{
    int i=0;
    while(breakLoop==0)
    {
        getprio(currpid);
        i++;
    }
    iter[id] = i;
    userCpuTime[id] = usercpu(currpid);
    return;
}
```



```
###q4test2####

id:0, iter:999999, userCpuTime:30, endTime:143
id:1, iter:995818, userCpuTime:30, endTime:144
id:2, iter:104824, userCpuTime:30, endTime:145
id:3, iter:104814, userCpuTime:30, endTime:146
```

Figure 3: Testcase1 with 4 processes running simple while loops

I used a similar mechanism as in the testcase1 to ensure the child processes start and end at about same time. See Figure 3 for results. From the figure, we can see that the process that didn't run `getprio()` (id: 0, 1) ran for more number of iterations as compared to processes that ran `getprio()` in their while loop (id: 0, 1). However, all the processes ran for same amount of time (based on `userCpu`) and ended at about the same time. When the `getprio()` system call runs, interrupts are disabled. This means that `clkhandler()` is not called, `vfineclkcounter` is not incremented and `preempt` is not decremented when `getprio()` system call is running. However, the reason for decrease in the number of iterations for the processes that run `getprio()` could be due to the time consumed running the operations in `getprio()` that run before interrupts are disabled and after interrupts are enabled (small window).

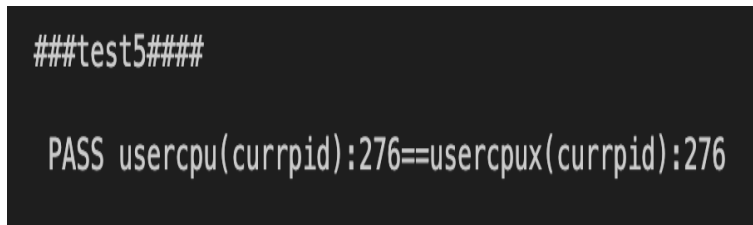
Bonus problem

Test code for `usercpux()`:

```
void process5()
{
    int i=10000000;
    while(i!=0)
    {
        i--;
    }
    int32 ms1 = usercpu(currpid);
    int32 ms2 = usercpux(currpid);
    if(ms1!=ms2)
    {
        kprintf("\n FAIL %d!=%d \n",ms1,ms2);
    }
    else
    {
        kprintf("\n PASS %d==%d \n",ms1,ms2);
    }
    return;
}

void test5()
{
    resume(create(process5,1024,45,"process5", 0));
}
```

Test results: See Figure 4

A terminal window with a black background and white text. The first line is '###test5####'. The second line is 'PASS usercpu(currpid):276==usercpux(currpid):276'.

```
###test5####

PASS usercpu(currpid):276==usercpux(currpid):276
```

Figure 4: Test output for `usercpux`