CS 503 Fall 2022: Lab 1

Suraj Aralihalli

September 2022

3. Inspecting and modifying XINU source, compiling, and executing on backend machines

3.1 Basic system initialization and process creation

1. **Question**: Why the output is consistent with correct operation of create2()?
Solution: The output i.e sequence of A's and B's is consistent and matches the output of create() because in my implementation of create2() I use the function ready() to change the state, add the process to the ready queue and reschedule. Additionally, I am using the same priority for all the processes. Each process runs for Quantum unit of time before its switched. Process "send A" prints a sequence of As for 1 quantum before it is context switched to process "send B" which prints a sequence of Bs. Since I have used the same priority value in create() and create2() while creating the processes and in the same order ($A \rightarrow B$), the output printed on the screen is consistent w.r.t both create() and create2().

2. **Question**: Trace the sequence of events that transpire when the function executed by create() returns.
Solution: When the function executed by create() returns, the control flows to the instructions at INITRET. INITRET is an alias to userret. Function userret() is responsible for killing the function from its PID. userret() function uses kill() system call which changes the state of the process to PR_FREE and calls resched() to maintain fixed priority scheduling. kill() also sends a message to the parent process. To reiterate the sequence of steps undertaken,

    (a) Control flows to userret()

    (b) userret() captures cur pid and calls kill(pid)

    (c) kill saves the interrupt mask and performs sanity checks

    (d) Decrements the prcount (process count)

    (e) Sends a message to the parent process

    (f) Frees the stack space of the process created by create() by calling freestk()

    (g) Changes the state of the child process id to PR_FREE

    (h) Calls resched() to maintain fixed priority scheduling

    (i) Restores the interrupt mask

3.2 Clock interrupt handling

Question: Test that fineclkcounter and vfineclkcounter are being updated correctly by XINU's modified clock interrupt handling code by making main() sleep for 5 seconds by calling sleep(), then printing the value of the two global variables.
Solution: See Figure 1

```
kprintf("\n fineclkcounter: %d", fineclkcounter);
kprintf("\n vfineclkcounter: %d", vfineclkcounter);
sleep(5);
kprintf("\n fineclkcounter: %d", fineclkcounter);
kprintf("\n vfineclkcounter: %d", vfineclkcounter);
```

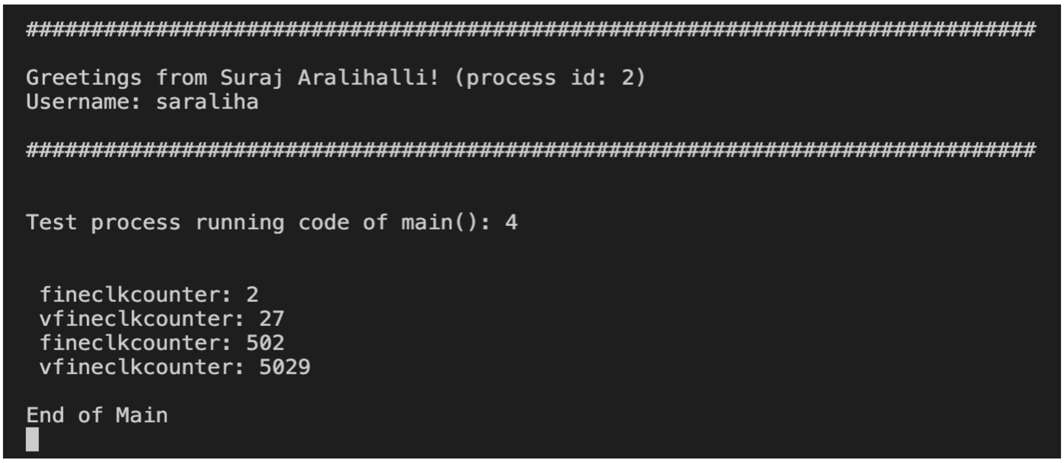

Figure 1: fineclkcounter and vfineclkcounter

3.3 Time slice management

Question: Test whether prcpuhungry accurately tallies how CPU intensive a process is by creating a test case.
Solution: My test consists of running two different processes and comparing the corresponding prcpuhungry value. As seen below I have a function named cpuIntensiveLow() that runs 10 million times and computes $sum = (sum + i - 1)\%i$; in each iteration. Similarly, I have another function named cpuIntensiveHigh() that runs the same operations but runs 100 million times. I then created the 2 processes in the main using create()/create2() and printed the prcpuhungry value of the process right before the process returns. Since this value represents the number of quantums a process consumed from the time it was created to the time the process was killed, I would expect cpuIntensiveHigh() to run for more quantums than cpuIntensiveLow(). From the figure 3, it is evident that prcpuhungry accurately tallies how CPU intensive a process as per my tests.

```
void cpuIntensiveLow()
{
    int i = 0;
```

```
    int sum = 0;
    while(i!=10000000)
    {
       sum=(sum+i-1)%i;
       i++;
    }
}

void cpuIntensiveHigh()
{
    int i = 0;
    int sum = 0;
    while(i!=100000000)
    {
       sum=(sum+i-1)%i;
       i++;
    }
}


process main(void)
{
    kprintf("\nTest process running code of main(): %d\n\n", getpid());

    create2(cpuIntensiveLow, 1024, 20, "cpuLow", 0);
    create2(cpuIntensiveHigh, 1024, 20, "cpuHigh", 0);

    kprintf("\n\nEnd of Main\n");
    return OK;
}
```

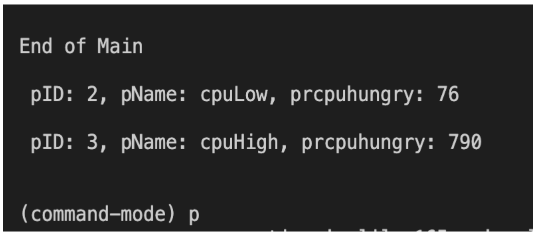

Figure 2: cpuIntensiveLow() and cpuIntensiveHigh()

3.4 Process lifetime

Question: Test and verify that the system call works correctly.
Solution: I modified the cpuIntensiveLow() function to display the lifetime() once before the for loop and once after the for loop. From the Figure **??** it is clear that the lifetime() accurately computes the total time the process was alive in the unit of 10ms.

```
void cpuIntensiveLow()
    {
        kprintf("\npid: %d, lifetime before loop: %d\n", getpid(), lifetime(getpid()));
        int i = 1;
        int sum = 0;
        while(i!=10000000)
        {
           sum=(sum+i-1)%i;
           i++;
        }
        kprintf("\npid: %d, lifetime after loop: %d\n", getpid(), lifetime(getpid()));
    }
```

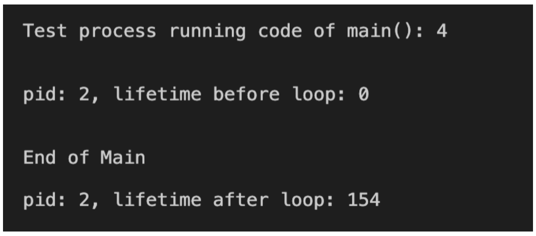

Figure 3: cpuIntensiveLow() with lifetime()

3.5 Energy conservation

Question: Use the tests of 3.3 and 3.4 to gauge that the energy efficient idle process appears to be working correctly.
Solution: Running the tests 3.3 and 3.4 after adding asm(”hlt”) gave similar results.

4. Interfacing C and assembly code

4.1 Calling assembly function from C function

Question: Test and verify that addtwo() and addfour() work correctly.
Solution: Functions testAddtwo() and testAddfour() can be found in mytests.c in systems directory. These functions are called in main() to test addtwo() and addfour() functions written in assembly. See Figure 4 for reference.

4.2 Calling C function from assembly function

Question: Test and verify that your code for testgreaterfirst and greaterfirst works correctly. **Solution:** testgreaterfirst() successfully accesses the arguments passed by its caller and communicates it to greaterfirst. It also ensures value contained in EAX is not disturbed and is returned to its caller. Function testTestgreaterfirst() used for testing can be found in mytests.c in systems directory.

```
        .text
        .globl testgreaterfirst
```

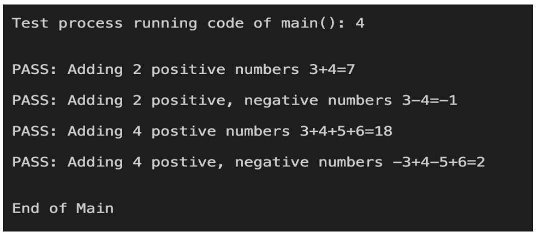

Figure 4: testAddtwo() with testAddfour()

```
testgreaterfirst:
        pushl   %ebp
        movl    %esp,%ebp
        pushfl
        pushl   %ebx
        movl    8(%ebp),%eax
        movl    12(%ebp),%ebx
        pushl   %ebx
        pushl   %eax
        call    greaterfirst
        popl    %ebx
        popl    %ebx
        popl    %ebx
        popfl
        movl    (%esp),%ebp
        add     $4,%esp
        ret
```

5. Run-time manipulation of return addresses

5.1 Wrong turn

Figure 5 shows the system entering panic state when greaterfirst1 is called from testgreaterfirst. Code snippet from greaterfirst1.c responsible for the trap follows.

```
asm("movl %ebp,ebp");
// ebp + 4 has return address
returnAddressToCaller = ebp + 0x1;
*returnAddressToCaller = 0x123456;
```

5.2 Expressway to main()

In greaterfirst2.c, ebp points to address that stores frame pointer to testgreaterfirst(). We then make ebpInCaller point to this address. We know that the return address to testgreaterfirst() is stored in the address pointed by ebp + 4bytes. Similarly return address to main() is stored in ebpInCaller + 4bytes. We now replace the return address to testgreaterfirst() with return address to main(). This

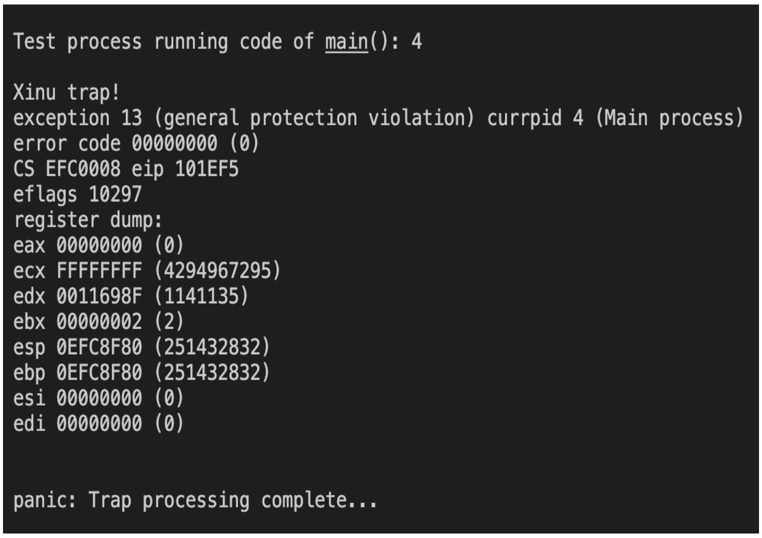

Figure 5: trapping when greaterfirst1 is called from testgreaterfirst

way when the machine instruction **call greaterfirst2** is completed, the control flows to main(). Below is the snippet from testgreaterfirst.S

```
unsigned long *ebpInCaller=NULL;
unsigned long *returnAddressToCaller=NULL;
unsigned long *returnAddressToMain=NULL;

asm("movl %ebp,ebp");

ebpInCaller = (unsigned long *)*ebp;

// ebp + 4 has return address
returnAddressToCaller = ebp + 0x1;
returnAddressToMain = ebpInCaller + 0x1;

*returnAddressToCaller = *returnAddressToMain;
```

6. Modifying interrupt handling behavior

```
process main(void)
{
    kprintf("\nBefore divide by zero \n");
    int o = 100;
    o = 5 / 0;
    kprintf("\nValue of k:%d \n", o);
    kprintf("\nAfter divide by zero \n");
    return OK;
}
```

The above code is used to test system's response to divides by zero operation. When Version1 of _Xint0 is used, the system gets stuck in an infinte loop. This is confirmed from the Figure 6. The print statements immediately after $o = 5/0$ are not printed, indicating that the process is in an infinite loop stuck at $o = 5/0$ statement.

When Version2 of _Xint0 is used, the process returns to the next statement after $o = 5/0$. Consequently, the print statments are executed and can be seen in Figure 6. This indicates that the process makes progress unlike the previous version and doesn't get stuck. In Version2 we know that the return address is at the top of the stack, we store this address into ecx register and add offset of 2 bytes to ecx (size of idiv instruction, See line 2c below). And we put it back on top of the stack without modifying anything else.

```
29:   c1 fa 1f                sar    $0x1f,%edx
2c:   f7 f9                   idiv   %ecx
2e:   89 44 24 1c             mov    %eax,0x1c(%esp)
```

Version1:

```
   .globl _Xint0
_Xint0:
    iret
```

Version2:

```
   .globl _Xint0
_Xint0:
    popl %ecx
    addl $2,%ecx
    pushl %ecx
    iret
```

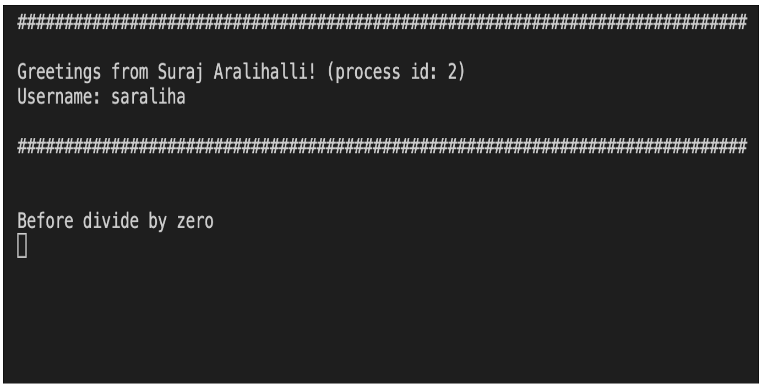

Figure 6: Version1 _Xint0: stuck in infinite loop

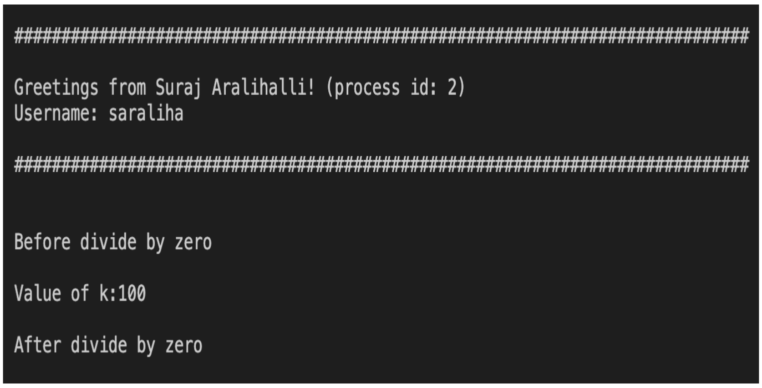

Figure 7: Version2 _Xint0: Process proceeds to next statement

Bonus problem

Code:

```
int createapp(char *cmd, char **argv)
{
    int pid = fork();
    //Child
    if (pid==0)
    {
        nice(10);
        // sleep(10);
        execvp(cmd, argv);
        // comes here only if execvp fails
        exit(1);
    }
    //Parent
    else
    {
        int status;
        // blocking call
        waitpid(pid, &status, 0);

        if ( WIFEXITED(status) )
        {
            int exit_status = WEXITSTATUS(status);
            if(exit_status==1)
            {
                return -1;
            }
        }
    }
    return pid;
```

```
}
```

My approach to fix the flaw where createapp() would return the PID of the child even if execvp() in the child fails, is to use a blocking system call from the parent that waits until the child executes successfully (exit status 0) or terminates (exit status 1). Based on this exit status, the parent

1. returns **pid** of the child if execvp() in child succeeds
2. returns **-1** if execvp() in the child fails

The same behaviour is also found in system() in Linux which returns only after the command has been completed. I have included the code snippet below where I use waitpid() with third argument 0. This makes the waitpid() a blocking system call and it returns only after the child is terminated. The status (success/failure) of the child process is captured in the exit_status variable. If the exit_status is 1, then we return -1. Otherwise, we return pid of the child as instructed in the question.

I tested the successful execution of execvp() by passing the valid arguments to createapp() function. On the other hand the I simulated the failure execution of execvp() by passing invalid arguments to createapp(). For example, in the first test case I passed valid arguments (See Testcase 1 below). I could see the pid of child process printed on the terminal as expected. When I passed invalid arguments in testcase 2 to intentionally fail the exevp() command, the parent process waits until the completion of the child process and identifies that the exit_status of child is 1 and prints -1 on terminal.

Testcase 1:

```
int main()
{
    char* command = "ls";
    char* argument_list[] = {"ls", "-l", NULL};
    int i = createapp(command, argument_list);
    printf("\npid: %d\n", i);

}
```

Testcase 2:

```
int main()
{
    char* command = "qdhsjjdsaj";
    char* argument_list[] = {"qdhsjjdsaj", "-l", NULL};
    int i = createapp(command, argument_list);
    printf("\npid: %d\n", i);

}
```