

3.1. Tested by running `asm("movl %ecx, %cr1");` in `main.c`, when `invalopcode` equals 5, it jumps to `trap6()` as expected. See the result in Figure 1.

```
Trap6: invalopcode=5

panic: Invalid opcode exception occurred 5 times - halting XINU.
```

Figure 1

3.2. Test what happens if `main()` executes `int3` using inline assembly. (We observed debugging information such as trap information, exception type, and register contents, as shown in Figure 2) What happens if you modify `_Xint3` so that its first instruction is `iret`? (It has no output. The main program continued to run after triggering `iret`) What does the observed behavior imply about the EIP value saved on the stack when exception 3 is generated? (This implies that the EIP saved on the stack was the address of the instruction following `int3`. Therefore, when the `iret` instruction restores registers, control is transferred to that next instruction, allowing the program to continue normally without entering the breakpoint exception handler) Discuss your findings in `lab2ans.pdf`.

```
Xinu trap!
exception 3 (breakpoint) currpri 3 (Main process)
CS 8 eip 10246F
eflags 202
register dump:
eax 00000000 (0)
ecx 00000000 (0)
edx 00000000 (0)
ebx 00000000 (0)
esp 0EFC8FE4 (251432932)
ebp 0EFC8FE4 (251432932)
esi 00000000 (0)
edi 00000000 (0)

panic: Trap processing complete...
```

Figure 2

3.3. In 3.1 and 3.2 we used x86 instructions `mov` and `int3` to generate exceptions. Another way of generating synchronous interrupts is via instruction `INT`, also called software interrupt. Inserting `asm("int $6")` in `main()` will generate exception 6, the same invalid opcode exception generated by `asm("movl %ecx, %cr1")` with the main difference that the EIP value saved on the stack is the address of the instruction following `"int $6"`. Describe in `lab2ans.pdf` how you would verify that this is indeed the case. (We have the handler do nothing but `iret` in `_Xint6`, and output test information before and after the instruction `"int $6"` in `main.c`. If we observe output after the

instruction, it means that EIP is the address of the next instruction and the execution is resumed (i.e. If the EIP is the address of the instruction itself, then repeated interrupts would have occurred). See the test result in Figure 3) Test your method and verify that it works as expected. Replace int3 in 3.2 with asm("int \$3") and describe what you find. (We ran the same test with "int \$3" and observe similar behaviour, meaning that EIP value was still the address of the following instruction. See the test result in Figure 4) Note that int3 is not the same x86 instruction (different opcode) as int.



```
Before int $6.  
After int $6, execution resumed.
```

Figure 3



```
Before int $3.  
After int $3, execution resumed.
```

Figure 4

A kernel may use the INT instruction (in general, software interrupts) in the upper half to invoke services of the lower half. This complements calling a lower half function directly from an upper half function which we will discuss when discussing input/output synchronization and coordination later in the course. In x86 XINU, a hardware clock that drives the 1 msec periodic system timer is mapped to interrupt number 32. Code main() such that it prints "Before start of processing loop" before entering an infinite loop. Within the loop insert an if-statement that checks if the global variable clktime which counts (in unit of second) the time elapsed since XINU bootloaded if it has reached 8 seconds. If so, print "8 second mark reached" then called exit() to terminate the process running main(). Test and verify that approximately 8 seconds elapses in-between the two output messages. (It took approximately 8 seconds before exiting) Modify main() so that inline assembly asm("int \$32") is inserted before the if-statement. Test your modified code. (In this case, the program exited almost immediately, way before 8 seconds. Normally, hardware triggers interrupt 32 at a fixed rate and the clock tick handler will eventually update clktime to represent elapsed seconds. However, when "int \$32" is added, the clock routine runs every single time the loop is executed, causing the 8 second mark to occur much faster than normal) Discuss in lab2ans.pdf your finding.

3.4. Perform an experiment where you use x86's int instruction to generate interrupt number 55. Describe what happens. Find out the underlying reason by looking up references for x86 and XINU's IDT configuration code in system/evec.c. (When calling "int \$55", XINU's IDT does not have a specific handler for interrupt 55, so the default handler is invoked, as shown in Figure 5)

```

Testing int $55.
Xinu trap!
exception 13 (general protection violation) curripid 3 (Main process)
error code 000001ba (442)
CS EFC0008 eip 102480
eflags 10246
register dump:
eax 00000001 (1)
ecx 00000000 (0)
edx 0011666A (1140330)
ebx 00000000 (0)
esp 0EFC8FCC (251432908)
ebp 0EFC8FCC (251432908)
esi 00000000 (0)
edi 00000000 (0)

panic: Trap processing complete...

```

Figure 5

We noted in class that XINU configures IDT such that external interrupts are not automatically disabled when an interrupt is generated. The responsibility to achieve mutual exclusion by disabling interrupts when processing of a previous interrupt is ongoing is delegated to kernel software. In XINU this means that interrupt handling code executes cli upon entry and sti before returning from an interrupt to reenale external interrupts. XINU configures the entries of IDT to be of type TRAP GATE to do so. Configuring the entries as INTERRUPT GATE delegates interrupt disabling to x86. Check system/evect.c to determine how entries are set to be of type TRAP GATE. Look up x86 references to determine what is needed to configure an entry as INTERRUPT GATE. Perform the modification to the clock interrupt only which is mapped to interrupt number 32. Rerun your test from 4.1, lab1, and describe your finding. (The values of clkcounterfine and clktime are still consistent, as shown in Figure 6) Given that interrupt disabling/enabling is by both x86 and clkdisp.S, does leaving interrupt disabling/enabling code in clkdisp.S influence correct behavior? (No, it does not. We changed it and ran it again, the result remained the same, as shown in Figure 7. The reason is that the interrupt flag is already cleared by the hardware, invoking another cli is just a redundant operation meant for robustness) Discuss your reasoning in lab2ans.pdf. With respect to IDT configuration for clock interrupts in XINU, note that XINU configures all relevant entries in a loop in evect.c, and separately for interrupt number 32 which overwrites the default configuration.

```

clkcounterfine = 2262
clktime = 2 seconds

```

Figure 6

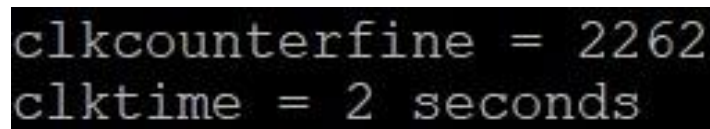
```

clkcounterfine = 2262
clktime = 2 seconds

```

Figure 7

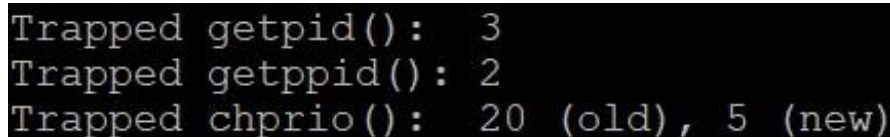
We discussed how XINU's configuration of x86's GDT data structure differs from that of Linux and Windows by not having entries for user mode code and data segments. In addition, XINU has a separate entry for the stack segment which Linux and Windows do not. As part of setting up a viable system configuration before running XINU kernel code proper, start.S calls XINU's basic configuration code to set up GDT (and IDT), loads gdtr to point to the beginning address in main memory where GDT has been configured, and sets up registers CS, DS, and SS to point to their respective entries in GDT. Modify the code in start.S so that SS points to the same entry as DS, the method employed by Linux and Windows. There is no need to remove the separate stack segment entry from GDT since we will just ignore it. Rerun the test of 4.1, lab1, and check that output is as expected. (We still get the same expected output, as shown in Figure 8. Using the same selector for DS and SS does not adversely affect memory accesses, since the DS descriptor is already configured with the necessary attributes for data access) Discuss your finding in lab2ans.pdf.



```
clkcounterfine = 2262
clktime = 2 seconds
```

Figure 8

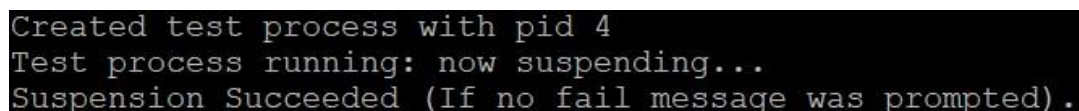
4. We call the wrapper functions in main.c to test its correctness. The test results are shown in Figure 9.



```
Trapped getpid(): 3
Trapped getppid(): 2
Trapped chprio(): 20 (old), 5 (new)
```

Figure 9

Bonus. We create a test process and suspend it, while writing lines to output suspension failure after calling suspend(). We observed that the fail message was not shown, meaning that our implementation worked correctly. See the test result in Figure 10.



```
Created test process with pid 4
Test process running: now suspending...
Suspension Succeeded (If no fail message was prompted).
```

Figure 10