3.1

I declared int invalopcode=0; in initialize.c and put extern int invalopcode; in prototypes.h.

To modify _Xint6, I push registers, compare invalopcode with 5, increment, then iret until it reaches 5, at which point we call trap6(). It prints invalopcode and calls panic(), hanging the system.

Then, In main(), I did asm("movl %ecx, %cr1");, an invalid instruction on x86. Because the CPU re-executes that instruction each time, invalopcode increments until 5, then trap6() is invoked. Because we never return from the invalid instruction or from panic(), so no subsequent lines in main() run.

New Xint6-

```
_Xint6:
        pushal
        cmpl   $5, invalopcode
        jl     .lessThan5
        call   trap6
        jmp    .

.lessThan5:
        incl   invalopcode
        popal
        iret
```

Testing-

```
process main(void)
{
    kprintf("About to trigger invalid opcode...\n");
    asm("movl %ecx, %cr1");
    kprintf("If you see this, #UD was not triggered.\n");
    return OK;
}
```

Output-

```
agraw185,Agrawal,Shivli,Spring,2025
About to trigger invalid opcode...
Trap 6 triggered; invalopcode = 5


panic: Invalid opcode threshold reached! System halting...
```

3.2

When I ran asm("int3"); with the default _Xint3, I saw XINU print a trap message (exception #3) and then panic. And never reached the line after int3 in main(). So I replaced _Xint3's body with a single iret and saw both 'Before int3' and 'After int3', no trap message, because _Xint3 immediately returned to the next instruction.

This implies that the CPU saves on the stack the address of the instruction immediately following the int3. Because when debuggers use a breakpoint instruction, they want to resume execution after the int3 once the debugger is done. So by design, the CPU sets EIP in such a way that an iret effectively steps over the breakpoint. This shows that the CPU saves the address of the next instruction on the stack for breakpoint exceptions, letting iret return to the instruction after the int3.

Testing-

```
kprintf("Before int3\n");
asm("int3");
kprintf("After int3\n");
```

Output-

```
agraw185,Agrawal,Shivli,Spring,2025
Before int3
After int3
```

3.3

On replacing asm("movl %ecx, %cr1"); with asm("int $6") I saw that int $6 triggers the same exception #6, but the CPU pushes the address of the next instruction on the stack. If the _Xint6 only did iret, we would return after the int $6 instruction and continue in main() rather than re-executing the invalid op. On using an actually invalid opcode like movl %ecx, %cr1, the CPU saves the address of the same failing instruction, so it re-faults on iret. So the software interrupt instructions save EIP pointing past the instruction which is not like hardware fault instructions that re-try the same failing instruction.

Then, I used asm("int3"); to generate exception #3. I replaced it with asm("int $3");. Both end up calling _Xint3.

int3 is a 1-byte opcode (0xCC) and int $3 is a 2-byte sequence (0xCD 0x03).

The CPU's behavior is the same for #3 in that it sets EIP to the next instruction on the stack. If _Xint3 does an iret, it resumes after the int $3. Debuggers insert the single-byte int3 (0xCC) at runtime into user code for breakpoints because it's easier to overwrite just one byte.

Now,, I wrote main() so it prints "Before start of processing loop", then enters a while(1) loop. Inside the loop, it checks the global clktime (in seconds). Once it reaches 8, we print "8 second mark reached" and call exit(). Initially, it takes almost 8 real‑world seconds to reach clktime = 8.

Then, I added asm("int $32"); at the top of the loop before checking clktime. That code often triggered the clock interrupt handler more frequently than the hardware's 1 kHz timer alone. As a result, clktime incremented faster in software, so the 8 second mark could be reached in much less than 8 real seconds. We understood that repeatedly forcing interrupt #32 in software can desynchronize clktime from real‑world time.

Testing-

```
process main(void)
{
    kprintf("Before start of processing loop\n");
    while (1)
    {
        asm("int $32");  // test with and without

        if (clktime ⩾ 8) {
            kprintf("8 second mark reached\n");
            exit();
        }
    }
    return OK;
}
```

Output-

first terminal output-

```
agraw185,Agrawal,Shivli,Spring,2025
Using int $3 instead of int3
Back from int $3
```

Second-

```
agraw185,Agrawal,Shivli,Spring,2025
Before start of processing loop
8 second mark reached
```
(in actually 8 seconds)

Third-

```
agraw185,Agrawal,Shivli,Spring,2025
Before start of processing loop
8 second mark reached
```
(the mark reached very quickly this time though)

3.4

Generating Interrupt #55, I placed asm("int $55"); in main(). Because XINU's IDT only configures entries for 0–47, interrupt #55 lacks a valid entry. The CPU detects an invalid interrupt vector and raises a General Protection Fault #13.

I also see a trap message and a panic, and never return to main() to print anything afterward. This happens because XINU calls set_evec(i, ...) in a loop only up to i < 48 in evec.c. The IDT has no entry for #55, so hardware triggers a GPF.

Testing-

```
kprintf("Before int $55\n");
asm("int $55");
kprintf("After int $55\n");
```

Output-

```
agraw185,Agrawal,Shivli,Spring,2025
Before int $55
Xinu trap!
exception 13 (general protection violation) currpid 3 (Main process)
error code 000001ba (442)
CS EFC0008 eip 10281F
eflags 10246
register dump:
eax 00000001 (1)
ecx 00000000 (0)
edx 00116A20 (1141280)
ebx 00000000 (0)
esp 0EFC8FCC (251432908)
ebp 0EFC8FCC (251432908)
esi 00000000 (0)
edi 00000000 (0)



panic: Trap processing complete...
```

Currently, pidt->igd_type = IGDT_TRAPG value = 15 decimal, i.e. 0xF, which is a trap gate. A trap gate does not automatically disable external interrupts on entry. Instead, clock code in clkdisp.S explicitly does cli on entry and sti before returning.

So to change clock interrupt (#32) from trap gate to interrupt gate I changed only the clock interrupt vector #32 to be an interrupt gate type=0xE, or 14 decimal, which automatically disabled interrupts on entry and re-enabled on iret.

```
pidt->igd_type = IGDT_TRAPG;
 if (xnum == 32) {
     pidt->igd_type = 0xE;
 }
```

On rerunning the Lab 1.4.1 clock experiment, the clock still incremented normally, and timing was correct. The code in clkdisp.S that does cli/sti became somewhat redundant because hardware was already disabling interrupts upon entry. Given the same results as before, confirm that leaving cli/sti calls in clkdisp.S does not conflict with the CPU's automatic disabling in an interrupt gate. The kernel code remains correct and functional.

Then, I saw that selectors for DS -> 0x10 and SS -> 0x18 in the GDT. This is different from Linux/Windows, which unify them in 32-bit mode.

So I replaced the line that loads SS with the same selector as DS in start.S:

```
asm
 movl  $0x10, %eax
 movw  %ax, %ds
 movw  %ax, %es
 movw  %ax, %fs
 movw  %ax, %gs
 //movl $0x18, %eax
 movw  %ax, %ss
```

And left the separate stack descriptor in the GDT unused. Now on rerunning the same Lab 1.4.1 clock test, the system behaved exactly the same. Because XINU does not rely on hardware segmentation for separate user vs. kernel stacks, unifying SS and DS caused no visible differences.

4.2.1

For this part, I started with creating system/getppidtrap.c and defined it in prototypes.h 'pid32 getppidtrap(pid32 pid);'. Then I implemented getppidtrap() to function as the system call API. It accepts one argument, a pid32 value. Then loads the system call number (21) into register EDX. On executing the trap instruction int $35, it switches to kernel mode. After the trap completes, it obtains the return value from EAX and returns it.

First implementation-

```
#include <xinu.h>
#include "prototypes.h"

pid32 getppidtrap(pid32 pid)
{
    pid32 retval;  (Variable to store the return value)
    asm volatile (
        "movl $21, %%edx\n\t"  (Place system call number 21 into EDX)
        "int $35\n\t"          (software interrupt)
        "movl %%eax, %0\n\t"   (return value from EAX into retval)
        : "=r" (retval)
        :
        : "edx", "eax", "ecx", "ebx", "esi", "edi"   (clobbered registers)
    );
   return retval;
}
```

Then I realized that my clobber list conflicts with the chosen input/output registers, because it marked EAX as both clobbered and the register output. So I specified a more precise output constraint-

```
#include <xinu.h>

#include "prototypes.h"
pid32 getppidtrap(pid32 pid)
{
    pid32 retval;
    asm volatile (
        "movl $21, %%edx      \n\t"
        "int $35              \n\t"
        : "=a" (retval)
        :
        : "edx", "ecx", "ebx", "esi", "edi", "memory"
    );
    return retval;
}
```

This approach came after a bit of brainstorming, where the line "=a"(retval) tells GCC the output is in register EAX. So there's no separate instruction inside the assembly block like movl %%eax, %0.

Because now EAX is an output, I don't list eax as a clobbered register. Instead, let GCC handle it via the output operand. I still declare EDX as clobbered because the assembly modifies it. And also add memory to the clobber if the kernel code might change memory.

Testing for this-

```
process main(void)

{
    pid32 ppid;
    kprintf("Invoking trapped getppid system call...\n");
    ppid = getppidtrap(getpid());
    kprintf("Trapped getppid returned: %d\n", ppid);
    return OK;
}
```

Initial output-

```
agraw185,Agrawal,Shivli,Spring,2025
Invoking trapped getppid system call...
Xinu trap!
exception 35 currpid 3 (Main process)
CS EFC0008 eip 101BFD
eflags 206
register dump:
eax 00000003 (3)
ecx 00000000 (0)
edx 00000015 (21)
ebx 00000000 (0)
esp 0EFC8F9C (251432860)
ebp 0EFC8F9C (251432860)
esi 00000000 (0)
edi 00000000 (0)


panic: Trap processing complete...
```

This tells that the system call wraps getppidtrap()) to do int $35, but Xtrap → trap(), which panics by default. So the code never actually calls getppid(). So I call a Dispatcher Function from _Xint35. It reads the system call number from EDX, calls the corresponding internal kernel function getppid(), returns the result in EAX, and then irets.

```
.globl  _Xint35
.extern syscalldisp

_Xint35:
    pushal
    call   syscalldisp
    popal
    iret
```

Syscalldisp.c-

```
#include <xinu.h>
extern pid32 getppid(pid32);

void syscalldisp(void)
{
    uint32 scnum;
    asm("movl %%edx, %0" : "=r" (scnum));

    switch(scnum) {
    case 21:
        {
            pid32 arg;
            asm("movl 44(%%esp), %0" : "=r" (arg));
            pid32 ret = getppid(arg);
            asm("movl %0, %%eax" : : "r" (ret));
            break;
        }

    default:
        asm("movl $-1, %eax");
        break;
    }
}
```

Now on testing again with the above main, I see-

```
Invoking trapped getppid system call...
Trapped getppid returned: 3
```

Which means the trapped system call is working. getppidtrap() executed int $35, the system call dispatcher at _Xint35 recognized the system call number 21, fetched the argument from the user stack, called getppid(), and put its return value in EAX. Control returned to getppidtrap(), which printed the parent PID 3. No panic or trap message means successfully bypassing the old jmp Xtrap path and using the new dispatcher logic.

4.2.2

For this the system call dispatcher, the second software layer I had to handle disabling interrupts on entry. Then, checking the system call number stored in register EDX. Also, extracting arguments from the user stack, and calling the corresponding upper-half kernel function, like getppid(). Finally, return the function's result in EAX and ensure the stack is in the correct state before executing iret to return to user mode since this ensures that system calls from user mode are correctly forwarded to their kernel-mode implementations. So I modified _Xint35 in intr.S to handle system calls, which previously jumped to Xtrap causing a panic, with a custom system call dispatcher. cli is used to prevent unwanted stack shifts.

```
        .globl  _Xint35
        .extern getppid
_Xint35:
        cli
        pushal
        cmpl   $21, %edx
        je     .do_getppid
        movl   $-1, %eax
        jmp    .returnFromSyscall

.do_getppid:
        movl  44(%esp), %ecx
        pushl %ecx
        call  getppid
        addl  $4, %esp
```

```
.returnFromSyscall:
        popal
        iret
```

Initially, I assumed the first argument passed was located at 44(%esp), but I saw incorrect results (parent PID = 512, 518, 544, etc.). This was because disable() was adding extra function call overhead, shifting the expected stack layout. To debug and correct the offset, I printed out values at different stack locations-

```
   movl 36(%esp), %eax
    pushl %eax
    pushl $msg36     # Debug print: "36(%esp) = %d\n"
    call kprintf
    addl $8, %esp

    movl 40(%esp), %eax
    pushl %eax
    pushl $msg40     # Debug print: "40(%esp) = %d\n"
    call kprintf
    addl $8, %esp

    movl 44(%esp), %eax
    pushl %eax
    pushl $msg44     # Debug print: "44(%esp) = %d\n"
    call kprintf
    addl $8, %esp
```

Output-

```
36(%esp)= 251396104
40(%esp)= 518
44(%esp)= 0
48(%esp)= 0
Returned parent pid: 544
```

After testing, I confirmed that 44(%esp) was the correct location for the pid argument when cli was used instead of call disable().

Then on final testing-

```
process main(void)
{
    kprintf("Calling getppidtrap()...\n");
    pid32 ppid = getppidtrap(getpid());
    kprintf("Returned parent pid: %d\n", ppid);
    return OK;
}
```

Output-

```
Calling getppidtrap()...
Returned parent pid: 3
```

This confirmed that the system call successfully traps into _Xint35 using int $35 and the dispatcher correctly extracts the user argument from the stack. The dispatcher invokes getppid() and retrieves the correct result. And, EAX holds the correct parent PID (3) upon return.

Now, I had to support 0, 1, or 2 argument system calls as the current only supported system calls with a single argument like getppid(pid32 pid). So I checked the system call number stored in EDX and correctly branch to:
.do_getpid for 0 arguments.
.do_getppid for 1 argument.
.do_chprio for 2 arguments.

I started by finding that the first argument is located at 44(%esp) after pushal. And the second argument is at 48(%esp). The arguments are pushed right-to-left before calling the function. So after the function call, I adjusted the stack using addl $X, %esp to remove the pushed arguments.

Final implementation-

```
_Xint35:
        cli
        pushal
        cmpl   $20, %edx
        je     .do_getpid
        cmpl   $21, %edx   # check 2
        je     .do_getppid
        cmpl   $22, %edx   # check 3
        je     .do_chprio
        movl   $-1, %eax
        jmp    .returnFromSyscall
# 0 arguments (getpidtrap)
.do_getpid:
        call   getpid
        jmp    .returnFromSyscall
# 1 argument (getppidtrap)
.do_getppid:
        movl  44(%esp), %ecx
        pushl %ecx
        call  getppid
        addl  $4, %esp
        jmp    .returnFromSyscall
# 2 arguments (chprio)
.do_chprio:
        movl  44(%esp), %ecx
        movl  48(%esp), %ebx
        pushl %ebx
        pushl %ecx
        call  chprio
        addl  $8, %esp
        jmp    .returnFromSyscall
.returnFromSyscall:
        popal
        iret
```

Then I ran a final test-

```
process main(void)
{
    kprintf("Testing getpidtrap()...\n");
    pid32 mypid = getpidtrap();
    kprintf("Returned pid: %d\n", mypid);

    kprintf("Testing getppidtrap()...\n");
    pid32 parent = getppidtrap(getpid());
    kprintf("Returned parent pid: %d\n", parent);
    kprintf("Testing chpriotrap()...\n");
    pri16 old_prio = chpriotrap(getpid(), 5);
    kprintf("Returned old priority: %d\n", old_prio);

    return OK;
}
```

Output-

```
agraw185,Agrawal,Shivli,Spring,2025
Testing getpidtrap()...
Returned pid: 1
Testing getppidtrap()...
Returned parent pid: 3
Testing chprio()...
Returned old priority: 5
```

This confirmed that the system calls correctly transition from user space to kernel space and back.

4.2.3

As stated and implemented above, I reused XINU's existing system calls—getpid(), getppid(), and chprio(), as upper-half kernel functions. These functions execute the service on behalf of user-mode applications that invoke system calls via int $35. The system call dispatcher (_Xint35) calls these upper-half functions while ensuring that EAX is not overwritten before executing iret and the CDECL calling convention is followed to correctly pass and return arguments.

As implemented above, I updated _Xint35 to invoke getpid(), getppid(), and chprio() based on the system call number in EDX and each function executed in C and returned a value in EAX, which is preserved for the user wrapper. _Xint35 also ensured that after a system call like getppid(), EAX is not altered before returning to user mode. The final _Xint35 implementation correctly calls kernel functions and returns via iret, preserving results. I modified getppidtrap() to return from the trap, it copies the value in EAX to a local variable, which is then returned to the main.

In the getppidtrap() implementation, EAX is preserved by _Xint35 and read back into retval by getppidtrap()and retval is returned to main(), completing the required flow.

```
#include <xinu.h>
#include "prototypes.h"
pid32 getppidtrap(pid32 pid)
{
    pid32 retval;
    asm volatile (
        "movl $21, %%edx      \n\t"
        "int $35              \n\t"
        : "=a" (retval)
        :
        : "edx", "ecx", "ebx", "esi", "edi", "memory"
    );
    return retval;
}
```

4.3

Trapped system call support for these XINU calls:

getpid() – 0 arguments – system call number 20
getppid() – 1 argument – system call number 21
chprio() – 2 arguments – system call number 22

Corresponding wrapper functions:

getpidtrap() in system/getpidtrap.c
getppidtrap() in system/getppidtrap.c
chpriotrap() in system/chpriotrap.c

In include/process.h I added-

```
#define SYSCALL_GETPID  20
#define SYSCALL_GETPPID 21
#define SYSCALL_CHPRIO  22
```

so _Xint35 knows how to dispatch calls based on EDX.

4.4

As I have shown testing in all the above sections, I will conclude by highlighting that I tested the trapped system calls by calling-

```
pid32 mypid   = getpidtrap();
pid32 parent  = getppidtrap(mypid);
pri16 oldprio = chpriotrap(mypid, 5);
```

and verifying that mypid is a sensible PID & parent is the parent's PID & oldprio matches the process's old priority.
I confirmed that no panic or trap message appears, meaning _Xint35 is correctly extracting arguments and returning EAX unmodified.

```
/*  main.c  - main */
#include <xinu.h>
process main(void)
{
    kprintf("Testing getpidtrap()...\n");
    pid32 mypid = getpidtrap();
    kprintf("Returned pid: %d\n", mypid);
    kprintf("Testing getppidtrap()...\n");
    pid32 parent = getppidtrap(getpid());
    kprintf("Returned parent pid: %d\n", parent);
    kprintf("Testing chprio()...\n");
    pri16 old_prio = chpriotrap(getpid(), 5);
    kprintf("Returned old priority: %d\n", old_prio);
    return OK;
}
```

Output-

```
agraw185,Agrawal,Shivli,Spring,2025
```

```
Testing getpidtrap()...
Returned pid: 1
Testing getppidtrap()...
Returned parent pid: 3
Testing chprio()...
Returned old priority: 5
```

Bonus-

For this, I extended the trapped system call implementation to support suspend(). suspend() is a system call that puts the current process into a suspended state, removing it from the ready queue and forcing a context switch via resched(). A suspended process remains inactive until it is explicitly resumed.
Then I implemented suspendtrap(), and assigned it system call number 23. On being callled, the wrapper places the value 23 in register EDX and issues an int $35 instruction. The system call dispatcher then recognizes this call and invokes the suspend() function as the upper-half kernel function. This function changes the process's state, removes it from the ready list, and returns the process's old priority code..

```
 syscall suspendtrap(pid32 pid)
   {
       syscall retval;
       asm volatile (
           "movl $23, %%edx      \n\t"
            "int $35              \n\t"
           : "=a" (retval)
           : "b" (pid)
           : "edx", "ecx", "esi", "edi", "memory"
       );

       return retval;
   }
```

I extended the dispatcher in _Xint35 to handle system call number 23. For a suspend call, the dispatcher extracts the single argument from the user stack and calls the suspend() function like getppidtrap. I made no changes to the suspend function. Instead, repurposed it as the upper-half kernel function for the trapped suspend system call. The dispatcher calls suspend with the target PID, and its return value is passed through to the suspendtrap.

This ensures that the dispatcher does not disturb the contents of EAX after the suspend returns so that suspendtrap can correctly retrieve the old priority value.

To test-

```
process test_suspend(void)
{
    while (1) {
        kprintf("Running process %d\n", getpid());
        sleep(2);
    }
    return OK;
}

process main(void)
{
    pid32 testpid = create(test_suspend, 8192, 20, "test_suspend", 0);
    resume(testpid);
    sleep(3);
    status s = suspendtrap(testpid);
    kprintf("Suspended process %d ⇒ status %d\n", testpid, s);
    return OK;
}
```

I then concluded that if a process suspends itself or is the only runnable process, the scheduler might immediately reselect it. So the bonus problem implements that.