CS 354 Spring 2025

Lab 2: Trapped System Call Implementation in x86 XINU using Software Interrupt [280 pts]

Due: 2/19/2025 (Wed.), 11:59 PM

1. Objectives

The objectives of this lab are to understand XINU's basic interrupt handling on x86 Galileo backends for synchronous interrupts where the source of an interrupt is either a fault (or exception) or software interrupt int. The latter has been used as the traditional mechanism for system call implementation in Linux and Windows kernels. We will utilize the INT instruction to change XINU system calls from regular function calls into trapped system calls which follows the traditional design. It is not a complete implementation that introduces simplifications, necessitated by content that will be covered under process management where process creation, context-switching, and scheduling are discussed.

---

2. Readings

- Read Chapter 4 from the XINU textbook.

When working on lab2:

For the written components of the problems below, please write your answers in a file, lab2ans.pdf, and put it under lab2/. You may use any word processing software as long as they are able to export content as pdf files using standard fonts. Written answers in any other format will not be accepted.

Please use a fresh copy of XINU, xinu-spring2025.tar.gz, but for preserving the output of from lab1 identifying authorship, and removing all code related to xsh from main() (i.e., you are starting with an empty main() before adding code to perform testing). As noted before, main() serves as an app for your own testing purposes. The GTAs when evaluating your code will use their own main() to assess your XINU kernel modifications.

3. Handling of synchronous x86 interrupts [80 pts]

3.1 Invalid opcode exception and modifying kernel response

Although gcc is unlikely to generate machine code that triggers an invalid opcode exception which is a synchronous interrupt (i.e., "synchronous" in the sense that an instruction of the current process is the cause as discussed in class), writing assembly code or embedding assembly in C code can result in interrupt number 6 if not careful. A case in point is accessing x86 control registers CR0-CR8, some of which are reserved and not allowed to be accessed. For example, embedding

```
asm("movl %ecx, %cr1");
```

into C code that tries to copy the content of ECX to CR1 will result in interrupt number 6. The operation as a whole comprised of opcode and operand is considered invalid. Trying to use a general-purpose register such as %EAF which does not exist is detected by gcc. CR1 exists but is not used in x86, a condition that is not checked by gcc.

Modifying the content of control registers is a task performed by operating system kernels and associated system tools. For example, during bootloading x86's protected mode that separates kernel mode/user mode is enabled by setting the PE bit (first bit) of CR0 to 1. In the graduate version of CS 354, CS

503, the last lab assignment involves a 4-week virtual memory lab where paging -- a topic we will discuss in the second half of the course but not implement -- is enabled after bootloading by setting the PG bit (last bit of CR0) to 1. Our XINU version runs with protected mode enabled (i.e., PE = 1) but paging disabled (i.e., PG = 0).

The seventh entry in IDT set up by XINU during system initialization (the same goes for Linux and Windows) for interrupt number 6 will contain a function pointer to an interrupt handler that x86 will jump to in order to execute lower half kernel code. The function pointer -- an entry point or label that represents an address in main memory -- is _Xint6 in system/intr.S. Modify the code at _Xint6 so that instead of jumping to Xtrap which calls trap() in evec.c _Xint6 saves the content of x86's 8 general purpose registers on the run-time stack by executing the pushal instruction. Recall from our discussion that before the interrupt handling code at _Xint6 is reached x86 hardware has already pushed the content of EFLAGS, CS, and EIP registers onto the stack. After executing pushal _Xint6 should compare a global variable, int invalopcode = 0, to be declared in initialize.c, against constant 5. If invalopcode is strictly less than 5 then _Xint6 increments

invalopcode, executes popal to restore the 8 general purpose registers saved on the stack, then executes iret. iret atomically pops and restores the values of EFLAGS, CS, EIP registered saved on the stack.

To determine what happens next, we need to check what value was saved by x86 when it pushed EIP onto the stack upon encountering the invalid opcode interrupt. In most cases, the address of the instruction which caused the exception is stored which facilitates re-running the instruction if the cause of an exception can be fixed by the interrupt handling routine. This is also the case for interrupt number 6. Hence upon executing iret which loads the program counter (i.e., instruction pointer) EIP with the saved value on the stack, the same instruction "movl %ecx, %cr1" will be executed again causing another invalid opcode exception.

Implement _Xint6 so that if it _Xint6 determines that invalopcode equals 5, it jumps to trap6(), void trap6(void), to be coded in system/evec.c, which outputs the value of invalopcode using kprintf(), then calls panic() with a suitable message which will cause XINU to hang. Test by generating exception number

6 from main() and verify that XINU's modified lower half works as expected.

3.2 Breakpoint exception 3

In x86 exception number 3, called the breakpoint exception, may be used by debuggers to insert breakpoints at run-time in user code that allow inspection of the state of a process to aid resolve bugs. Exception 3 can be generated using the int3 instruction

```
asm("int3");
```

which will cause XINU's lower half to execute code at _Xint3 in system/intr.S. Test what happens if main() executes int3 using inline assembly. What happens if you modify _Xint3 so that its first instruction is iret? What does the observed behavior imply about the EIP value saved on the stack when exception 3 is generated? Discuss your findings in lab2ans.pdf.

3.3 Generating synchronous interrupts using instruction INT

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. Test your method and verify that it works as expected. Replace int3 in 3.2 with asm("int $3") and describe what you find. Note that int3 is not the same x86 instruction (different opcode) as int.

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. Modify main() so that inline assembly asm("int $32") is inserted before the if-statement. Test your modified code. Discuss in lab2ans.pdf your finding.

3.4 Modifying IDT and GDT

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.

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 on-going 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 reenable 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/evec.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. Given that interrupt disabling/enabling is by both x86 and clkdisp.S, does leaving interrupt disabling/enabling code in clkdisp.S influence correct behavior? 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 evec.c, and separately for interrupt number 32 which overwrites the default configuration.

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. Discuss your finding in lab2ans.pdf.

---

4. Trapped system call implementation [200 pts]

4.1 Objective

XINU system calls are regular C function calls that do not contain a trap instruction to switch between user mode and kernel mode. We will implement trapped versions of XINU system calls that follow most of the way Linux and Windows traditionally implement system calls on x86. Our implementation does not go all the way as we haven't yet covered scheduling and the mechanics of process context-switching which are needed for a newly created process to execute in user mode. Problem 4 covers the bulk of traditional trapped system call implementation in x86 Linux and Windows where a system call uses software interrupt INT to trap to the lower half of the XINU kernel, changes to a different stack (i.e.,

kernel stack), and jumps to a kernel function in the upper half to carry out the request task. Upon completion, the upper half kernel function returns to the lower half which switches from kernel stack back to user stack and untraps by returning to user code.

4.2 Three software layers

As discussed in class, we consider three software layers when implementing trapped system calls.

4.2.1 System call wrappers

The first layer is the system call API which is a wrapper function running in user mode in Linux and Windows that ultimately traps to kernel code in kernel mode via a trap instruction. For example, fork() in Linux is a wrapper function that is part of the C Standard Library (e.g., glibc on our frontend Linux machines). Similarly for CreateProcess() which is part of the Win32 API in Windows operating systems. XINU's getpid() system call is not a trapped system call since it does not execute a trap instruction to jump to kernel code in kernel mode. The same goes for the new system call, pid32 getppid(pid32), that you implemented in Problem 3.1 of lab1.

We will implement a wrapper function, syscall getppidtrap(pid32 pid), in system/getppidtrap.c that acts as the API that programmers use to invoke the kernel service. That is, as with fork(), getppidtrap() will be the system call API used to trap to kernel code via trap instruction INT coded using inline assembly. We will code using extended inline assembly which allows us to copy the value of C variables into registers before inline assembly code is executed, and vice versa after inline assembly has completed. Without this added capability the benefit of embedding assembly code within C code would be limited.

We will use interrupt number 35 which legacy XINU does not use as the entry of IDT (interrupt descriptor table) -- the interrupt vector of x86 -- to jump to the system call dispatcher code in XINU's lower half. Interrupt numbers 0-31 are reserved in x86 for backward compatibility. XINU uses 32 to service clock interrupts for implementing the 1 msec system timer, interrupt number 42 is used by TTY, and 43 by Ethernet. Traditionally interrupt number 46 is used by Windows to implement trapped system calls whereas Linux uses 128.

XINU's system call wrappers need to communicate which system call is being invoked (i.e., system call number) to the system call dispatcher. We will do so by copying the system call number into register EDX before executing, int $35, via extended inline assembly in the system call wrapper function. If system calls have arguments they must be passed to the system call dispatcher as well. For example, the system call API getppidtrap() has one argument of type pid32. We will follow the CDECL convention for x86-32 by pushing arguments onto the stack.

Last but not least, when coding extended inline assembly use the clobber list to convey to gcc which registers in the set EAX, EBX, ECX, EDX, ESI, EDI we may be modifying so that gcc can prevent conflicts from arising. Do not specify registers in the clobber list that are used as input and output by the assembly code. Using the clobber list obviates the need to save/restore register values used by the system call dispatcher which reduces programming.

4.2.2 System call dispatcher

The second software layer is the system call dispatcher in the lower half of the XINU kernel. Entry 35 in XINU's IDT has been

configured to point to label _Xint35 in system/intr.S. You will modify the assembly code at _Xint35 so that it performs the tasks of a system call dispatcher which include checking which system call is being called, making a call to an internal kernel function in the upper half to carry out the requested kernel service. As noted above, _Xint35 will assume that EDX contains the system call number, and arguments are passed following CDECL.

Upon entering _Xint35 we disable interrupts since XINU's IDT has not been configured to automatically do so. In legacy kernels such as Linux and Windows, the run-time stack is changed to a kernel stack since using the user stack to manage kernel function calls presents a vulnerability that attackers may exploit. We will forgo this step for simplicity.

_Xint35 checks the system call number contained in EDX and calls the associated internal kernel function in XINU's upper half to carry out the requested service. The system calls we support can have 0, 1, or 2 arguments. We will reuse the legacy XINU system call as the internal kernel function of the upper half. For example, getppidtrap() traps to _Xint35 which then calls getppid(). Since getppid() is coded in C, _Xtrap35 must abide by

CDECL when passing arguments to getppid(). Then _Xint35 invokes getppid() by executing the instruction, call getppid. Upon completion, getppid() returns to its caller _Xint35.

The system call dispatcher ensures that the return value from getppid() is preserved in EAX, then makes sure that the run-time stack is in the same state as when the trap was generated by "int $35" before executing iret to return to user code getppidtrap(). Even though a clobber list is used to avoid register conflicts, it is important to double-check that upon untrapping by executing iret the wrapper function, i.e., system call getppidtrap() runs correctly not adversely affected by the trap.

4.2.3 Re-purposing legacy XINU system calls as upper half kernel functions

The third software layer are internal kernel functions in the upper half that perform the service requested of the kernel by the system call wrapper function. As noted above, we will repurpose and use XINU's legacy system calls coded in C as internal kernel functions in the upper half. Thus legacy system call, pid32 getppid(pid32), is utilized as an upper half kernel

function that the system call dispatcher _Xint35 calls to carry out the actual service requested by a process.

System calls, by default, have a return value since, at a minimum, failure to complete a system call for myriad reasons by returning -1 (i.e., SYSERR in XINU) must be supported. Since legacy XINU system calls, including getppid() that you implemented in lab1, are coded in C, values are returned to the caller through register EAX per CDECL. _Xint35 needs to communicate the return value of getppid() to the wrapper function getppidtrap() which is coded in C augmented by extended inline assembly. getppidtrap(), in turn, returns a value to its caller in register EAX per CDECL. Hence _Xint35 must not disturb the content of EAX returned by getppid() before executing iret. EAX must not be saved and restored before _Xint35 untraps to getppidtrap().

Code extended inline assembly in the system call getppidtrap() so that after returning from the trap it copies the value in EAX to a local variable of the wrapper function which is then returned to caller of getppidtrap(), i.e., an app function. In our case, by default, main() since we use it as a means to perform app layer testing of kernel modifications.

4.3 Supported system calls

We will provide trapped system call support for three XINU system calls: getpid(), getppid(), chprio(). The number of arguments are 0, 1, and 2, respectively. Their wrapper functions are getpidtrap() in system/getpidtrap.c, getppidtrap() in system/getppidtrap.c, chpriotrap() in system/chpriotrap.c. Define the system call numbers for the three system calls as 20, 21, and 22 in include/process.h.

4.4 Testing

Test and verify that your trapped system call implementation works correctly. Discuss your method for assessing correctness in lab2ans.pdf.

Bonus problem [20 pts]

Extend the trapped XINU system calls to include suspend() with wrapper function suspendtrap() in system/suspendtrap.c. Assign system call number 23. Test and verify that your implementation works correctly.

Note: The bonus problem provides an opportunity to earn extra credits that count toward the lab component of the course. It is purely optional.

Turn-in instructions

General instructions:

When implementing code in the labs, please maintain separate versions/copies of code so that mistakes such as unintentional overwriting or deletion of code is prevented. This is in addition to the efficiency that such organization provides. You may use any number of version control systems such as GIT and RCS. Please make sure that your code is protected from public access. For example, when using GIT, use git that manages code locally instead of its on-line counterpart github. If you prefer not to use version control tools, you may just use manual copy to keep track of different versions required for development and testing. More vigilance and discipline may be required when doing so.

The TAs, when evaluating your code, will use their own test code (principally main()) to drive your XINU code. The code you

put inside main() is for your own testing and will, by default, not be considered during evaluation. If your main() is considered during testing, a problem will specify so.

If you are unsure what you need to submit in what format, consult the TA notes link. If it doesn't answer your question, ask during PSOs and office hours which are scheduled M-F.

Specific instructions:

1. Format for submitting written lab answers and kprintf() added for testing and debugging purposes in kernel code:

- Provide your answers to the questions below in lab2ans.pdf and place the file in lab2/. You may use any document editing software but your final output must be exported and submitted as a pdf file.
- For problems where you are asked to print values using kprintf(), use conditional compilation (C preprocessor directives #define combined with #ifdef and #endif) with macro XINUTEST (in include/process.h) to effect print/no print depending on if XINUTEST is defined or not. For your debug statements, do the same with macro XINUDEBUG.

2. Before submitting your work, make sure to double-check the TA notes to ensure that any additional requirements and instructions have been followed.

3. Electronic turn-in instructions:

i) Go to the xinu-spring2025/compile directory and run "make clean".

ii) Go to the directory where lab2 (containing xinu-spring2025/ and lab2ans.pdf) is a subdirectory.

For example, if /homes/alice/cs354/lab2/xinu-spring2025 is your directory structure, go to /homes/alice/cs354

iii) Type the following command

turnin -c cs354 -p lab2 lab2

You can check/list the submitted files using

turnin -c cs354 -p lab2 -v

Please make sure to disable all debugging output before submitting your code.

Back to the CS 354 web page