CS 503 Fall 2022

Lab 2: Trapped System Call Implementation and Process Monitoring (240 pts)

Due: 09/28/2022 (Wed.), 11:59 PM

1. Objectives

The objectives of this lab are to extend XINU system calls -- which are regular C function calls -- into trapped system calls using x86 hardware support. We will also monitor CPU usage of processes and its dependence on dynamic process behavior.

---

2. Readings

1. XINU set-up
2. Read Chapter 5 from the XINU textbook.

Please use a fresh copy of XINU, xinu-fall2022.tar.gz, but for preserving the myhello() function from lab1 and removing all code related to xsh from main() during testing. main() serves as an app for your own testing purposes. The TAs will use their own main() to evaluate your XINU kernel modifications.

---

3. Trapped system call implementation [180 pts]

3.1 Objective

As discussed in the lectures and evident by inspecting any XINU system call code, all XINU system calls are regular C function calls. That is, there is no special trap instruction in XINU system calls that switches a process from user mode to kernel mode, and via the lower half jumps to a kernel function in the upper half to carry out the requested service. In this problem we will implement a trapped version of XINU system calls that goes 85% of the way Linux and Windows traditionally implement system calls in x86. Our implementation does not go all the way as we have not yet covered the mechanics of process context-switching which is needed to start executing a new process in user mode.

That is, after a process calls create() to spawn a child in suspended state and calls resume() to ready it, system call resume() executes upper half code in kernel mode and calls XINU's scheduler, resched(), to determine which process to run next. If the scheduler determines that the newly created child process should run next, upper half code would need to checkpoint and context-switch out the parent so that the child process can be context-switched in. Just before a jump to the first instruction of the user code of the child process (i.e., function pointer in the first argument of create()), "untrapping" must happen where kernel mode changes to user mode along with atomic updates to CS, SS, EFLAGS registers so that the run-time stack is switched from kernel stack to user stack and the new process runs in user mode with interrupts enabled. Thus making a new process start its life in user mode is embedded in the mechanics of process creation and context-switching which falls under process management.

Our implementation covers the bulk of traditional trapped system call implementation in x86 Linux and Windows where a system call contains an instruction to trap to the lower half of the XINU kernel, switch to a different stack (i.e., kernel stack), and jump to the kernel function in the upper half to carry out a request task. Upon completion, the upper half kernel function returns to the lower half which switches back to user stack and untraps by returning to user code following the system call with interrupts enabled. We will update the content of CS, DS, SS registers, albeit pointing to separate but duplicate code, data, stack segments in GDT with DPL set to 00. A XINU process making a trapped system call starts its life in kernel mode, remains in kernel mode even though it switches to different code/data/stack segments while running kernel code, and returns to user code running in kernel mode with its original code/data/stack segments. Hence the 85% characterization.

3.2 Three software layers

Three principal layers govern the software side of trapped system call implementation. Modification of GDT and IDT is described in 3.3.

System call wrappers. The first layer is system call API which is a wrapper function running in user mode 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. We will implement a system call wrapper function for getpid(), pid32 getpidx(void), in C and use extended inline assembly to trap to kernel code by executing the (synchronous) software interrupt instruction, "int $46", where the operand specifies the interrupt number we will use to demultiplex system call requests in the lower half. Note that interrupt numbers 0-31 are reserved, XINU uses 32 to service clock interrupts, 42 is used by TTY, and 43 by Ethernet (e.g., set_evec() call in devices/eth/ethinit.c to modify IDT). Interrupt number 46 is used by Windows whereas Linux uses 128.

You will reconfigure entry 46 in IDT so that it references a different code segment from the original kernel code segment, but with the DPL bits still set to 00. Thus trapping will change the content of the CS register, albeit to a duplicate of the original kernel code segment at ring 0. Since a XINU process was running in kernel mode before calling getpidx(), the trap via instruction "int" did not result in a change in privilege level such as if the process had been running in user mode (DPL 3) and the trap resulted in a switch to kernel mode (DPL 0). Since privilege level did not change, x86 upon executing "int $46" will push onto the user's run-time stack the content of EFLAGS, CS, EIP, and the hardware will not institute a stack change. ESP will point to the top of the user run-time stack which contains the saved content of EIP.

System call dispatcher. Entry 46 in XINU's IDT has been configured to point to label _Xint46 in intr.S. You will modify the assembly code at entry point _Xint46 so that it performs the task of a system call dispatcher: check which system call is being called, then make a call to an internal kernel function in the upper half. _Xint46 will assume that the wrapper function getpidx() communicates using register EAX which system call is being requested. Arguments are passed by the system call wrapper by pushing them on the stack right-to-left as in CDECL before executing the "int" instruction. Hence after the trap EFLAGS, CS, EIP are preceded by the arguments of the system call.

Basic chores of _Xint46 include saving the state of the user code and disabling interrupts since XINU IDT entries are configured as TRAP gates where x86 does not automatically disable interrupts. Modify IDT selectively so that entries for interrupt numbers 32 (clock interrupt) and 46 are set to INTERRUPT gates where x86 disables interrupts upon executing "int $32" or "int $46". The default gate types of all XINU IDT entries are TRAP gates where set_evec() in evec.c sets the IDT field idt.igd_type to value IGDT_TRAPG (15). Set it to 14 for interrupt numbers 32 and 46 so that they are configured as INTERRUPT gates. When x86 pushes the content of EFLAGS onto the stack when "int $32" is executed its IF bit will be 1. The IF bit in the EFLAGS register after executing "int 32 will be 0. This obviates executing cli upon entry into clkdisp.

Note that software behavior in response to interrupts, unlike prologue/epilogue code inserted by gcc for function calls, do not obey CDECL caller/callee convention. Your _Xint46 dispatcher code in concert with wrapper system call code has to ensure that relevant user state is saved after trapping and restored before untrapping back to user code. The saved content of EFLAGS, CS, EIP will be atomically restored by x86 when iret is executed.

Before the system call dispatcher makes a call to a upper half kernel function to carry out a requested service, it will switch stacks by loading %esp with a per-process kernel stack. Modify create() so that per-process kernel stacks are allocated by calling getstk() which returns the base address of a memory block to be used as per-process kernel stack. Keep in mind that stacks grow from high to low memory. Fix the size of a kernel stack to 4 KB which will suffice for bookkeeping of nested kernel function calls within XINU. Store the base address returned by getstk() in a global array, uint32 *kstack[NPROC], which can be referenced by the dispatcher using currpid as index. Since the system call dispatcher must switch back to the original user stack before "untrapping" by executing iret, remember the top of the user stack address by pushing it onto the kernel stack. Thus at the bottom of the kernel stack is a pointer to the top of the user stack. If the system call has arguments, they are copied from user stack to kernel stack. After updating DS, SS registers %ds, %ss, _Xint46 calls the upper half kernel function which is assumed to be coded in C. The CS register has been automatically updated by x86 upon trapping, and the old CS value pushed onto the user stack along with EFLAGS and EIP.

Upper half internal kernel functions perform system call tasks. Using the Linux system call fork() as analogy, fork() is the user mode wrapper function that eventually leads to internal kernel function do_fork() that, along with other kernel functions, carries out the work of spawning a process. We will reuse XINU's original system calls as internal kernel functions in the upper half to carry out the requested services. In the case of getpidx(), we will treat getpid() as the upper half kernel function that the dispatcher calls. Since getpid() has no arguments, after _Xint46 determines which kernel service is requested and executes "call getpid" the kernel stack will contain the saved user stack ESP and return address when reaching the first instruction of getpid(). Although getpid() does not invoke disable()/restore() to disable/enable interrupts, by default, XINU system calls do. Since the IF bit of EFLAGS will be 0 before the upper half kernel function is reached, calling disable() within an internal kernel function (i.e., legacy XINU system call function) will not affect the IF flag. In the case of calling restore() at the end of legacy XINU system calls, restore() does not unconditionally enable the IF bit but uses a flag set by disable() to restore the IF bit to its value before disable() was called. Hence IF remains 0 when the upper half kernel function returns to _Xint46, and it is safe to reuse legacy XINU system calls as upper half internal kernel functions.

Since getpid() is compiled by gcc following CDECL, the return value is communicated to _Xint46 using EAX. Your implementation of _Xint46 and getpidx() must keep this in mind so that getpidx() compiled by gcc can return the value stored in EAX by getpid() to its caller (e.g., test function main()). After the upper half kernel function returns to _Xint46, it must switch the runtime stack back to user stack, and update SS and DS. Just before untrapping to user code by executing iret, _Xint46 must set ESP so that is points to the top of the user stack where EIP was pushed by x86. Upon executing iret x86 will atomically pop the stored EIP, CS, EFLAGS values from user stack and update registers %eip, %cs, %eflags. When control returns to the system call wrapper function, it must clean up the stack in case arguments were passed before returning to its caller.

3.3 Reconfiguring XINU's GDT and IDT

We will reconfigure legacy XINU's GDT so that it supports additional entries for kernel code, kernel data, kernel stack that are to be used upon entry into the system call dispatcher via trap instruction "int $46". This is in addition to reconfiguring the type of IDT entries 32 and 46 to be INTERRUPT gates. First, we will delete the 4th entry of XINU's GDT since a separate entry for the stack segment is superfluous. SS can point to the same entry as DS since their attributes (R/W and DPL 0) are identical. The content with which to initialize GDT is configured by setting up gdt_copy[] in meminit.c. The first three entries (null entry, kernel code, kernel data) we will keep. The entry marked "3rd" (i.e., kernel stack) we will overwrite by creating a duplicate entry of the entry marked "1st" (i.e., kernel code). We will add a new entry to be marked as "4th" which will be a duplicate of the entry marked "2nd" (i.e., kernel data). When a system call traps we will switch CS, DS, SS to point to the entries marked "3rd" (for CS) and "4th" (for DS and SS). No privilege level change occurs since all entries are configured with DPL 0. Size of GDT is specified by parameter NGD. Make sure to find all the parts of XINU code where NGD occurs and update its value to reflect that GDT has grown by 1 entry.

Since each GDT entry is of size 8 bytes, updating the content of CS with index 0x08 means the first kernel code segment (entry "1st"). Loading DS and SS with 0x10 means first kernel data or stack segment (entry "2nd"). Updating CS with index 0x18 means second kernel code segment (entry "3rd") and loading DS and SS with 0x20 means second kernel data or stack segment. As discussed in class, in x86 %cs cannot be directly updated using the "mov" instruction. Instead, we rely on int/iret (or special instructions that force update of %cs such as ljmp) to reload the CS register. Loading of %ss using "mov" is allowed but only if the DPL values of %cs and %ss are the same. This is the case in our trapped system call implementation. After reconfiguring gdt_copy[], update setsegs() in meminit.c which overwrites XINU GDT table, gdt, with values stored in gdt_copy.

XINU IDT is configured by initevec() in evec.c by calling set_evec(). Modify the value of the segment selector, igd_segsel, of the idt structure for entry 46 so that it points to the second kernel code segment entry in GDT. Upon executing "int $46" this makes x86 save the current CS value by pushing it onto the stack along with the values in EFLAGS and EIP, and reload CS with the selector specified in the IDT entry for interrupt number 46. Since we do not use a TSS entry in GDT, x86 does not provide stack switching support and your code must update %ss and %esp. DS must always be updated by software. As discussed in class, configuring a newly created process so that it executes in user mode when it is scheduled to run must await discussion of the mechanisms underlying context-switching in process management. Hence in Problem 3, lab2, a process continues to start its life in kernel mode and remains in kernel mode until termination. For trapped XINU system calls, hardware support is utilized to trap into kernel code -- legacy XINU system calls are regular C function calls -- and switch to different code/data/stack segments, albeit from code/data/stack segments with DPL 0 to code/data/stack segments with DPL 0. In the full version implementing isolation/protection, we start from code/data/stack segments configured with DPL value 3.

3.4 Supported system calls

Since Problem 3 is an exercise in understanding how trapped system calls can be implemented in x86, we will provide trapped system call support for three XINU system calls: getpid(), getmem(), chprio(). We will call their wrapper functions getpidx(), getmem(), chpriox(), and place them in getpidx.c, getmemx.c, chpriox.c under system/. Define their system call numbers in include/process.h as #define SYSGETPID 20, #define SYSMEMGET 21, #define SYSCHPRIO 22. These values are passed by the wrapper functions to the system call dispatch upon trapping through EAX. getmem() is similar to malloc() in that it returns a pointer to the start of a requested contiguous memory block. getmem() (as is the case for getstk()) allocates memory in unit of 8 bytes. Even if 6 bytes are requested 8 bytes are reserved in the heap area. A difference between getmem() and malloc() is that getmem() is a XINU system call whereas malloc() is a user library function that makes brk()/sbrk() system calls to request memory. getpidx() has no arguments, getmemx() has one argument, and chpriox() has two arguments. The three XINU system calls are nonblocking calls. Unlike blocking calls such as sleepms() there is no call to XINU's scheduler resched() which may context-switch out the current process. Context-switching out due to time slice depletion cannot occur since our trapped system calls will run with interrupts disabled which prevents clkhandler() from updating preempt.

3.5 Testing

Test and verify that your trapped system call implementation works correctly. Discuss your approach for gauging correctness in lab2.pdf.

---

4. User code CPU usage monitoring and process behavior [60 pts]

Implement a system call, syscall usercpu(pid32), that takes as argument a PID and returns the time (in unit of millisecond) that it has spent executing user code. In general, this would correspond to time spent by a process in user mode. usercpu() returns syserr if the system call fails, in this case, PID is invalid. Place the code of usercpu() in system/usercpu.c.

To measure the user CPU time of processes, add a new process table field, uint32 prusercpu, that is initialized to 0 upon process creation. This includes the idle/NULL process where the value in its prusercpu field will be used to estimate CPU utilization. Whenever a backend machine's clock interrupt is raised, clkhandler() will increment prusercpu of the current process. Since XINU system calls and interrupt handlers run with interrupts disabled (i.e., IF = 0 in EFLAGS), clkhandler() is invoked when the current process runs user code. The resultant update to prusercpu provides an approximation of a process's true time spent executing user code due to timer granularity. We will discuss accuracy and precision issues when discussing clock management under device management. To gauge CPU utilization, reuse the 1 msec granularity counter vfineclkcounter from lab1. Divide the idle process's prusercpu by vfineclkcounter and map to an integer in the range 0-100 (percent). Code a wrapper function, unsigned short cpuutil(void), in system/cpuutil.c, that calls usercpu() to get the user CPU time of the idle process and calculate CPU utilization.

To test and assess correctness of your implementation consider the following scenarios. In one test case, create 4 processes of equal priority each performing the same while loop (i.e., user code) without making a system call. Increment a counter within the while loop. Make sure the parent that creates and resumes them in succession has higher priority. Note that in addition to starting the child processes at about the same time, we want them to terminate at about the same time as well. You may use the while loop counter as a termination condition. In a second test case, make two of the four processes call getprio() within the while loop. Note that getprio() is not a blocking system call, and, in XINU, will return to calling process without context-switching since interrupts are disabled. In both test cases we want all four processes to terminate at about the same time. We also want to check how many while loop iterations each process performed. Describe your method for terminating the four processes, and discuss your results in lab2.pdf.

Bonus problem [25 pts]

Extend trapped XINU system call support to include usercpu() from Problem 4. Place usercpux() in system/usercpux.c. Specify #define SYSUSERCPU 23 in include/process.h. Test 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, in general, not be considered during evaluation.

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 lab2.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 #if 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-fall2022/compile directory and run "make clean".

ii) Go to the directory where lab2 (containing xinu-fall2022/ and lab2.pdf) is a subdirectory.

For example, if /homes/alice/cs503/lab2/xinu-fall2022 is your directory structure, go to /homes/alice/cs503

iii) Type the following command

```
turnin -c cs503 -p lab2 lab2
```

You can check/list the submitted files using

```
turnin -c cs503 -p lab2 -v
```

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

Back to the CS 503 web page