CS 354 Fall 2023

Lab 2: Synchronous Interrupt Handling in x86 XINU and Trapped System Call Implementation [255 pts]

Due: 9/27/2023 (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: source of interrupt is either a fault (or exception) or software interrupt int. We will then utilize the INT instruction to change XINU system calls from regular function calls into trapped system calls. which follows the traditional method for implementing systems calls in Linux and Windows on x86 computers.

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, lab2.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-fall2023.tar.gz, but for preserving the helloworld() function from lab1 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 evaluate your XINU kernel modifications.

3. Handling of synchronous x86 interrupts [75 pts]

3.1 Divide-by-zero fault and modifying kernel response

x86 generates a divide-by-zero fault or fault which is interrupt number 0, if an instruction tries to divide a number by zero. For example, C code

```
int main() { int x = 10, y = 0; x = x / y; return; }
```

when translated into machine code will trigger divide-by-zero fault. The first entry in IDT set up by XINU during system initialization (the same goes for Linux and Windows) for interrupt number 0 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 _Xint0 in system/intr.S.

Modify the code at _Xint0 so that instead of jumping to Xtrap which calls trap() in evec.c _Xint0 saves the content of x86's 8 general purpose registers by executing the pushal instruction which pushes their values onto the run-time stack. Recall from our discussion that before the interrupt handling code at _Xint0 is reached x86 has already pushed the content of EFLAGS, CS, and EIP registers onto the stack.

After executing pushal _Xint0 should compare a global variable, int divzerocount = 0, to be declared in initialize.c, against constant value 10. If divzerocount is strictly less than 10 then _Xint0 increments divzerocount, 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. Restoring the register EIP with the saved return address implies that x86 will execute the instruction at the address in EIP next.

For most synchronous interrupts the EIP value pushed by x86 onto the stack after pushing the content of EFLAGS and CS registers is the address of the instruction that caused the interrupt. This is useful for diagnostic purposes since the EIP value on the stack will identify which instruction generated a fault. In some cases, a fault may be fixable. Executing iret results in jumping to the instruction that caused the fault which is re-executed. The second time around executing the same instruction will not trigger a fault since the kernel has applied a fix. For example, in the case of page fault (interrupt number 14) which is generated when a valid reference to main memory fails because the kernel kept the referenced content on disk due lack of space in main memory, the lower half's job of a kernel is to bring the missing content back into main memory and re-execute the memory reference instruction which will now succeed.

After modifying _Xint0, executing iret returns from lower half to user code, in particular, the same instruction that caused divide-by-zero fault which will be executed again. Since _Xint0 did not apply a fix, executing the same instruction will result in generation of interrupt 0 again. This cycle will repeat until divzerocount reaches 10. Code _Xint0 so that if divzerocount is greater or equal to 10 then _Xint0 calls a new kernel function, void trap0(void), to be coded in system/evec.c. All that trap0() does is output the value of divzerocount using kprintf(), then calls panic() with a suitable message which will cause the backend machine to hang (check the code of panic() in system/panic.c).

3.2 Invalid opcode fault and modifying kernel response

Although gcc is unlikely to generate machine code that triggers an invalid opcode fault, writing assembly code or embedding assembly in C code can result in interrupt number 6 if not careful. One case is accessing x86 control registers CR0-CR8, some of which are reserved. For example, executing instruction

```
movl %ebx, %cr1
```

that tries to copy the content of register EBX to register CR1 will generate interrupt number 6. The operation is considered invalid by Galileo's CPU. Changing the content of control registers is a task performed by operating system kernels and some system tools. For example, during bootloading x86's protected mode is enabled by setting the first bit of CR0 (PE bit) to 1. This allows kernel mode/user mode support by x86 hardware that may be utilized to implement isolation/protection.

Code a function, void invalidop6(void), in system/invalidop6.S which executes instruction, movl %ebx, %cr1. Call invalidop6() from main(). Modify trap() in evec.c so that as its first task trap() outputs a special message using kprintf() if the first argument passed to trap() -- an interrupt number -- is 6. Then trap() returns to its caller. Xtrap in intr.S is not coded to expect trap() to return since trap() calls panic() which causes a backend to hang indefinitely. Modify Xtrap so that it cleans up the run-time stack, then executes iret which returns to the instruction that generated the invalid opcode fault. Re-executing the instruction generates interrupt number 6 again.

Note: As in lab1, when new functions including system calls are added to XINU add its function prototype to include/prototypes.h.

When modifying trap() in evec.c declare a static variable, int invalidopcount = 0, which is incremented if the first argument of trap() is 6. If invalidopcount exceeds 4, call panic() with a suitable message. Thus after responding to a synchronous interrupt caused by an invalid instruction 4 times the kernel will make a backend hang indefinitely. When a backend hangs, will XINU's global variable, uint32 clktime, that keeps track of time in second since a backend was bootloaded cease being updated? Explain your answer in lab2.pdf.

3.3 Generating synchronous interrupts using instruction INT

In 3.1 and 3.2 we used actual division by 0 and actual execution of an invalid instruction to generate faults. Another way of generating synchronous interrupts is via instruction INT, also called software interrupt. We can embed assembly code in C code directly without calling a function coded in assembly through inline assembly asm(). In the code

```
int main() { int x = 3; asm("int $0"); kprintf("%d\n", x); return; }
```

gcc is asked to embed assembly instruction, int $0, at the specified location in C code when translated into assembly. Note that gcc produces assembly code from C code as an intermediate step before generating machine code from assembly code. Inline assembly is a useful programming technique that avoids overhead of making function calls. It is suited when assembly code to be embedded is short.

An important difference with generating synchronous interrupt 0 by actually performing a divide-by-zero operation is that software interrupt INT causes x86 to push the address of the instruction following INT onto the run-time stack. Hence if _Xint0 executes iret it will cause a jump to the instruction following INT. Unlike 3.1 the instruction that cause a synchronous interrupt will not be re-executed. In the main() code above, iret results in calling kprintf("%d\n", x) upon returning from _Xint0. Repeat 3.1 and 3.2 but with driver code main() using inline assembly to execute instruction int with operands 0 and 6, respectively. Verify that lower half XINU code in intr.S is executed only once.

4. Trapped system call implementation [180 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 needed for a newly created process to execute in user mode. Our implementation 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 jump to the 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 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.

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. Similarly for the new system call, int32 listancestors(pid32), from that you implemented in 3.1(e) of lab1 that output all ancestor processes and returned their number. We will implement a wrapper function, syscall listancestorsx(pid32 pid), in system/listancestorsx.c that acts as the API that programmers use to invoke the kernel service. That is, as with fork() listancestorsx() will be the system call API used to trap to kernel code via trap instruction INT coded using extended inline assembly. The chief difference of extended inline over plain inline assembly is the ability to copy the value of C variables into registers before inline assembly code executes, and vice versa after inline assembly has executed.

We will use interrupt number 33 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, 42 is used by TTY, and 43 by Ethernet. Traditionally interrupt number 46 is used by Windows whereas Linux uses 128 to trap to system call dispatchers in their kernel lower halves. XINU's system call wrappers must 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 EBX before executing, int $33, 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 listancestorsx() has one argument of type pid32 (which is typedef int32 whose C type is defined in include/kernel.h). If a system call has one argument as is the case for listancestorsx() we will pass its value from the system call wrapper to system call dispatcher in register ECX using extended inline assembly. For system calls that have two arguments, we will communicate the value of the second argument through EDX. For system calls with three arguments we will commuicate the third argument by pushing its value onto the run-time stack. We will not consider system calls with four or more arguments.

Lastly, when implementing 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 conflict is avoided. Do not specify registers in the clobber list 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.

System call dispatcher. The second software layer is the system call dispatcher in the lower half of the XINU kernel. Entry 33 in XINU's IDT has been configured to point to label _Xint33 in system/intr.S. You will modify the assembly code at _Xint33 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, _Xint33 will assume that EBX contains the system call number, single arguments in system calls are passed through ECX, two arguments in ECX and EDX, three arguments by pushing the third argument onto the stack. For example, in a system call, syscall abc(int a, int b, int c), with three arguments, a is passed in ECX, b in EDX, and c is pushed onto the stack.

Upon entry _Xint33 disables interrupts since XINU's IDT has not been configured to automatically disable interrupts. Then _Xint33 switches from user stack to kernel stack which is necessary for isolation/protection. When a process is created, we will allocate a second stack per-process to be used as kernel stack. We will do so by modifying create() which will call getstk() a second time and store the address returned in a new process table field, char *prsyscallkstack. The content of the second stack when allocated by create() is empty. Fix the size of the second stack to 8 KB.

EFLAGS
CS } kernel stack
EIP

All chores and function calls inside the kernel will be managed using the kernel stack which starts out empty. Before updating ESP in the system call dispatcher to switch the run-time stack from user to kernel stack, we will saveguard the EFLAGS, CS, EIP values pushed by x86 onto the user stack by copying the values to the kernel stack in reverse order. That is, the bottom of the kernel stack will contain a copy of EIP, followed by CS, and EFLAGS. Use the movl instruction to perform copying, not pushl. Since the system call wrapper has specified ESI and EDI in the clobber list we may use them in the system call dispatcher for the copy operation between the two stacks. To copy the saved EIP value at the top of the user stack, copy its value using movl into ESI. Copy the address of the kernel stack (initially empty) into EDI. Use movl to copy the value in ESI to the kernel stack using EDI and indirection. Update ESP and EDI, then copy the saved CS value from user stack to kernel stack. Repeat for EFLAGS.

Then check EBX to determine if the requested system call has 3 arguments. If so, copy the third argument which is at the top of the user stack pointed to by ESP to the top of the kernel stack using movl. Update ESP and EDI so that ESP points to the top of the user stack and EDI points to the top of the kernel stack. Before switching from user stack to kernel stack by copying the value in EDI to ESP, we must remember the address of top of the user stack. We will do so by copying the value of ESP into a new process table field, char *pruserstack. Now the remainder of the system call dispatcher and kernel upper half code can use the kernel stack to manage function calls within the kernel. When the chores of a system call (see below) are completed, _Xint33 must copy the values of EIP, CS, EFLAGS saved at the bottom of the kernel stack to the top of the user stack. Use movl, not pushl, to perform this operation. Then update ESP to point to the top of the user stack which contains EIP. Lastly, reenable interrupts before executing iret. Since the top of the user stack contains EFLAGS, CS, EIP, executing iret will atomically pop the three values and return to the instruction following INT in the system call wrapper.

After switching stacks, _Xint33 checks the system call number contained in EBX and calls the internal kernel function in XINU's upper half to carry out the requested service. The system calls we support can have 0, 1, 2, 3 arguments. We will reuse the legacy XINU system call as the internal kernel function of the upper half. For example, listancestorsx() traps to _Xint33 which then calls listancestors(). Since listancestors() is coded in C, _Xtrap33 must abide by CDECL when passing the argument contained in ECX to listancestors() by pushing its value onto the kernel stack. Then _Xint33 can invoke listancestors() by executing the instruction, call listancestors. Upon entry into listancestors() in the upper half of XINU, the kernel stack will contain the saved values of ECX and EIP2, where EIP2 at the top of the stack is the return address pushed by x86 when _Xint33 executes instruction, call listancestors. If the requested system call has three arguments then the system call dispatcher must ensure that the arguments are prepared on the kernel stack following CDECL before calling the internal kernel function which is coded in C.

Upper half internal kernel functions perform actual system call work. 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. 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, int32 listancestors(pid32 pid), is utilized as an upper half kernel function that the system call dispatcher _Xint33 calls to carry out the actual service requested by a process.

System calls, by default, have a return value since, at a minimum, possible failure to complete a system call for myriad reasons by returning -1 (i.e., SYSERR) must be supported. Since legacy XINU system calls, including listancestors() that you implemented in lab1, are coded in C, values are returned to the caller through register EAX following CDECL. Since the machine code of listancestors() is generated by gcc obeying CDECL the caller _Xint33 can assume that listancestors()'s return value is stored in EAX. _Xint33, in turn, must communicate the return value of listancestors() to the wrapper function listancestorsx() which is coded in C with extended inline assembly. Hence the machine code of listancestorsx() produced by gcc will return its value to the app function that called it in EAX. Thus _Xint33 must not disturb the content of EAX before executing iret to untrap to the system call wrapper function. Blindly saving/restoring EAX in _Xint33 will introduce a bug. Code extended inline assembly in the wrapper function listancestorsx() so that it copies the value in EAX to a local variable of the wrapper function which is then returned to the app function, by default, main() that called listancestorsx(). Do not execute the ret instruction directly from inline assembly.

4.3 Supported system calls

We will provide trapped system call support for four XINU system calls: getpid(), listancestors(), chprio(), procrange(), where, syscall procrange(uint16, pid32, pid32), to be coded in system/procrange.c is a new system call that returns the number of processes in the process table that meet a condition specified by the three arguments. The first argument takes on values 0, 1, 2 where 0 means a ready process, 1 means sleeping process, 2 means suspended process. The second and third arguments specify a PID range where the third argument must be greater or equal to the second argument. procrange() inspects the entries in XINU's process table and counts how many processes meet the specified requirement. For example, procrange(0, 3, 20) asks to check how many ready processes have PIDs between 3 and 20. If the arguments passed are invalid, procrange() returns SYSERR.

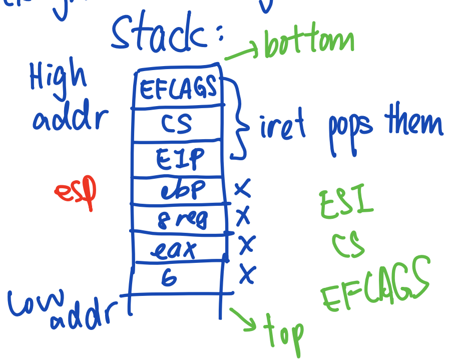
5 void 6 pid32 pid

The respective wrapper functions are getpidx() in system/getpidx.c, listancestorsx() in system/listancestorsx.c, chpriox() in system/chpriox.c, procrangex() in system/procrangex.c. Define the system call numbers for the four system calls as 5, 6, 7, 8, respectively.

7 pid32, pri16 8 uint16, pid32, pid32

System call dispatcher:

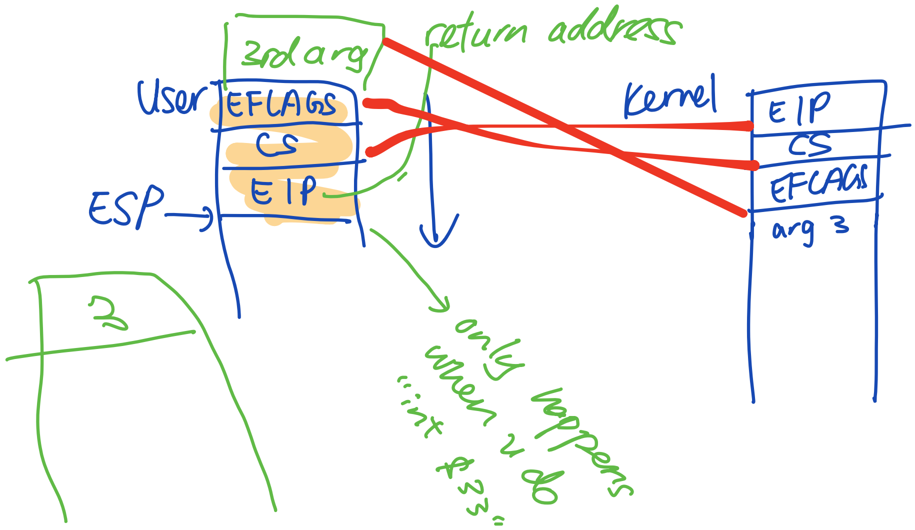
Stack grows from high to low



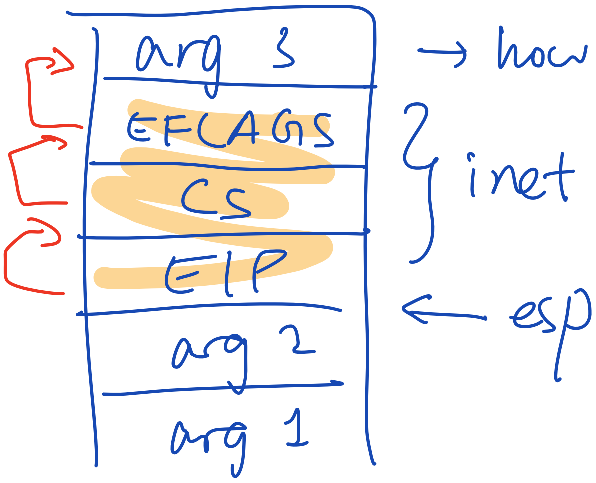
- Call getstk() in create() for kernel stack of size 8KB. ✓
  ↳ Returned in char *prsyscallkstack ✓
- Before switching to kernel, copy EFLAGS, CS, EIP in user to kernel
  - Copy EIP value (top of user stack) using movl into ESI ✓
  - Copy address of kernel stack into EDI ✓
  - Use movl to copy value in ESI to kernel stack using EDI & indirection (?) ✓
  - Update ESP & EDI → ESI (?) ✓
  - Copy saved CS value from user to kernel ✓
  - Repeat for EFLAGS ✓
- Check EBX to see if there's 3 arguments ⇒ If EBX==8, has 3 arguments ✓
  ↳ If yes, copy that from top of user stack (pointed by ESP) to top of kernel using movl ✓
- Update ESP & EDI so ESP points to top of user & EDI points to top of kernel (?)
- Copy value of ESP into char *pruserstack to remember address of top of user ✓
- Switch from user to kernel by copying the value in EDI to ESP ⇒ movl %EDI, %ESP ✓
- _Xint33 checks system call number in EBX & calls the internal kernel function in XINU's upper half
  ↳ E.g. → listancestorsx() traps to _Xint33 which calls listancestors()
    ↳ Pass the argument in ECX by pushing the value onto kernel stack
    ↳ call listancestors
    ↳ Kernel stack contains EIP2 at top which is the return address

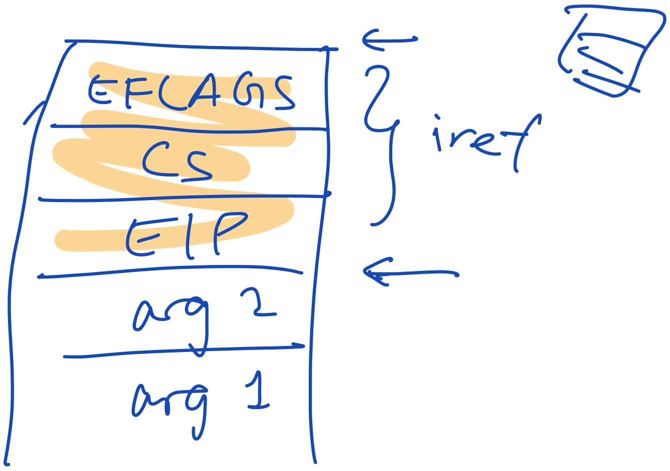
- listancestorsx() copies the value in EAX to a local variable that is returned

  * _Xint33 must not disturb EAX before executing iret
- _Xint33 copy values of EIP, CS, EFLAGS to top of user stack using movl
- Update ESP to point to top of user stack which contains EIP
- Reenable interrupts before executing iret

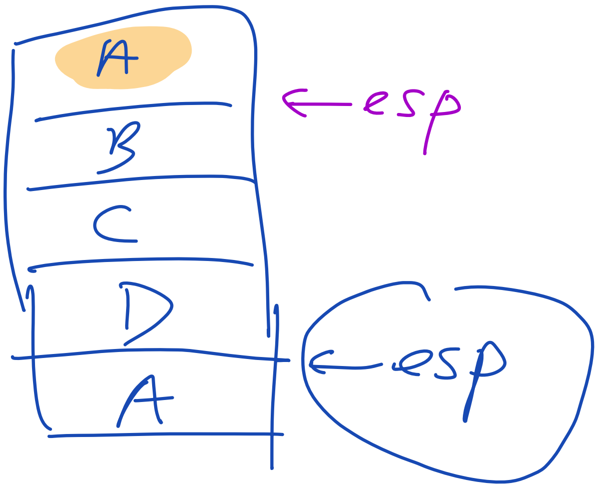


→ how to pop this?

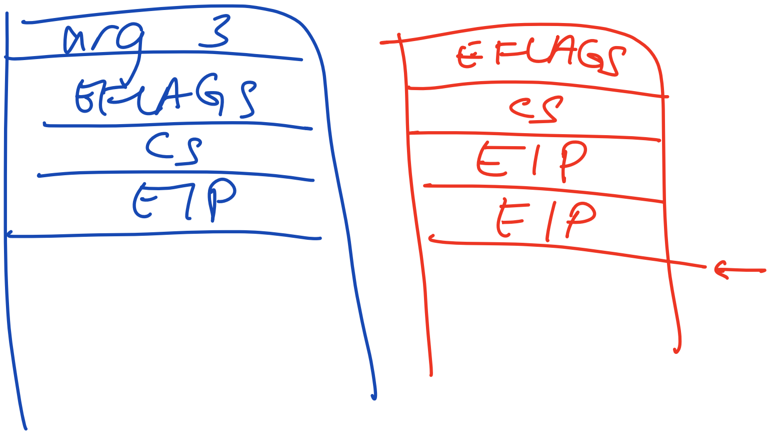




pushl 12(%esp)



add $12, %esp



4.4 Testing

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

Bonus problem [20 pts]

Extend the trapped XINU system calls to include getprio() with wrapper function getpriox() in system/getpriox.c. Assign system call number 9. 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 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 #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-fall2023/compile directory and run "make clean".

ii) Go to the directory where lab2 (containing xinu-fall2023/ and lab2.pdf) is a subdirectory.

For example, if /homes/alice/cs354/lab2/xinu-fall2023 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