CS 503 Fall 2022

Lab 1: Getting Acquainted with XINU's Software and x86 Hardware Environment (240 pts)

Due: 9/14/2022 (Wed.), 11:59 PM

1. Objectives

The objectives of the first lab assignment are to familiarize you with the steps involved in compiling and running XINU in our lab, and practice basic ABI programming that will be used as a building block in subsequent lab assignments.

---

2. Readings

1. XINU set-up
2. Chapters 1-4 from the XINU textbook.

For the written components of the problems below, please write your answers in a file, lab1.pdf, and put it under lab1/. You may use any number of 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.

3. Inspecting and modifying XINU source, compiling, and executing on backend machines

Follow the instructions in the XINU set-up which compiles the XINU source code on a frontend machine (Linux PC) in the XINU Lab, grabs an unused x86 Galileo backend machine, loads and bootstraps the compiled XINU image. Note that a frontend PC's terminal acts as a remote console of the selected backend Galileo x86 machine. If XINU bootstraps successfully, it will print a greeting message and start a simple shell called xsh. The help command will list the set of commands supported by xsh. Run some commands on the shell and follow the disconnect procedure so that the backend is released. Do not hold onto a backend: it is a shared resource.

3.1 Basic system initialization and process creation (45 pts)

(a) *XINU initialization, idle process, and process creation.* Inside the system/ subdirectory in xinu-fall2022/, you will find the bulk of relevant XINU source code. The file start.S contains assembly code following AT&T syntax that is executed after XINU bootstraps on a backend using the Xboot bootloader. Some system initialization is performed by Xboot, but most OS related hardware and software initialization is carried out by start.S and other XINU code in system/ after Xboot jumps to start in start.S. Eventually, start.S calls nulluser(), contained in initialize.c, which continues kernel initialization. nulluser() calls sysinit() (also in initialize.c) where updates to the data structure proctab[] are made which represents XINU's process table. In XINU, as well as in Linux/UNIX and Windows, a process table is a key data structure where bookkeeping of all created but not terminated processes is done. In our Galileo x86 backend which has a single CPU (or core), only one process can occupy the CPU at a time. In a x86 machine with 4 CPUs (i.e., quad-core) up to 4 processes may be running concurrently. Most of XINU's kernel code can be found in system/ (.c and .S files) and include/ (.h header files). At this time, we will use the terms "process" and "thread" interchangeably. Their technical difference will be discussed under process management.

When a backend machine bootstraps and performs initialization, there is as yet no notion of a "process." That is, the hardware just executes a sequence of machine instructions compiled by gcc and statically linked on a

frontend Linux PC for our target backend x86 CPU, i.e., the binary xinu.xbin. nulluser() in initialize.c calls sysinit() which sets up the process table data structure proctab[] which keeps track of relevant information about created processes that have not terminated. When a process terminates, its entry is removed from proctab[]. After configuring proctab[] to hold up to NPROC processes (a system parameter defined in config/), sysinit() sets up the first process, called the NULL or idle process. This idle process exists not only in XINU but also in Linux/UNIX and Windows. It is the ancestor of all other processes in the system in the sense that all other processes are created by this special NULL process and its descendants through system calls, e.g., fork() in Linux, CreateProcess() in Windows, and create() in XINU. The NULL process is the only process that is not created by system call create() but instead custom crafted during system initialization.

Code a new XINU system calls, create2(), that has the same function prototype as create() but behaves slightly different. First, a newly created process starts in the ready state. Recall that create() puts a new process in suspended state. Refer to include/process.h to determine how the ready state is internally represented in XINU. Also, check resume() which will indicate how a process is put in ready state which involves two steps: change the process state and insert the process in a priority queue of all ready processes. Second, if the value of the third argument of create2() is 0 the newly created process's priority is set to the parent's priority plus 1. Use the second example code in the lecture slides cs503-presentation-1.pdf where a process running main() creates two processes running sndA() and sndB() to test create2(). Instead of calling putc(), please call kputc() or kprintf() instead. That is, kputc('A') or kprintf("%c", 'A') in place of putc(CONSOLE, 'A') in sndA(), and analogously for sndB(). From the output of sndA() and sndB() (suitably modified since resume() will not be needed) observed on the terminal describe in lab1.pdf why the output is consistent with correct operation of create2(). You may want to run create() on the original example code main() is the lecture slides as a reference point. Use the modified XINU version of (b) to perform testing. Refer to the "Turn-in instructions" section on how to submit lab1.pdf.

Note: When new functions including system calls are added to XINU, make sure to add its function prototype to include/prototypes.h. The header file prototypes.h is included in the aggregate header file xinu.h. Every time you make a change to XINU, be it operating system code (e.g., system call or internal kernel function) or app code, you need to recompile XINU on a frontend Linux machine and load a backend with the new xinu.xbin binary.

(b) *Role of XINU's main and removing xsh.* After nulluser() sets up the NULL/idle process, it spawns a child process using system call create() which runs the C code startup() in intialize.c. Upon inspecting startup(), you will find that all it does is create a child process to run function main() in main.c. We will use the child process that executes main() as the test app for gauging the impact of kernel modifications on application behavior. In the code of main() in main.c, a "Hello World" message is printed after which a child process that runs another app, xsh, is spawned which outputs a prompt "xsh $ " and waits for user command. We will not be using xsh since it is but an app not relevant for understanding operating systems. Remove the "Hello World" message from main() and put it in a separate function, void myhello(void), in system/myhello.c. Call myhello() from nulluser() before it creates the idle process. Customize the "Hello World" message so that it contains your name and username. Don't forget to insert the function prototype of myhello() in include/prototypes.h. Modify main() so that it is completely empty but for a message that says "Test process running code of main():" followed by the PID of process. You may determine the PID of the current process by calling getpid(). Since the current version of XINU does not implement user mode/kernel mode (and memory) separation, you may also use the global variable currpid declared in initialize.c to reference the PID of the current process. Carry over these modifications to subsequent lab assignments unless specified otherwise. When performing testing in (a), use the XINU version of (b).

(c) *Process termination.* We will consider what it means for a process to terminate in XINU. Since the first argument of create() is a function pointer, normal termination implies execution of return (i.e., ret instruction in x86) by the function. The question, therefore, is where does the function return to since it was not called by another function. The technique used by XINU is to set up the run-time stack upon process creation so that it gives the illusion that another function called the function executed by create(). This function is identified by the macro INITRET whose definition can be found in one of the header files in include/. Playing detective, trace the sequence of events that transpire when the function executed by create() returns. Describe the sequence in lab1.pdf. Note that create() uses the internal kernel function getstk() (we will discuss its working later in the course under memory management) to allocate a run-time stack for a newly created process whose size is specified in the second argument of create(). During the termination sequence

of a process the memory used by its run-time stack is reclaimed (i.e., freed) by calling freestk(). The internal kernel function, resched(), implements fixed-priority scheduling as discussed in class. At the end of day, describe how process termination concludes and what happens after the current process has terminated? Ignore the stack content above INITRET (keep in mind that stack grows from high to low memory), i.e., at the top of the stack which we will consider when discussing context-switching.

3.2 Clock interrupt handling (20 pts)

Some system parameters in XINU are configured in config/Configuration which contains meta data from which C code (conf.c and conf.h in config/) is produced using automated translation tools. These parsing and lexical analysis tools covered in a compiler course will not concern us as they pertain to automated system configuration, not the architecture and implementation of the XINU kernel per se. The IRQBASE constant 32 in config/Configuration results in its inclusion as a C preprocessor directive in config/conf.h. IRQBASE defines the component of the x86 interrupt vector to which clock interrupts are mapped. Recall from computer architecture that the interrupt vector serves as an interface between hardware and software so that when an interrupt occurs it is routed to the responsible software component for handling it. That is, the interrupt handling code of an operating system that is tasked with responding to the specific interrupt. The clock interrupt on our x86 Galileo backends is configured to go off every 1 millisecond (msec), i.e., 1000 clock interrupts per second. Several years back a popular configuration was 10 msec. The default value varies across kernels and is, typically, configurable. We will investigate the role of system timers when discuss device management. x86 supports 256 interrupts numbered 0, 1, ..., 255 of which the first 32, called exceptions (or faults), are reserved for dedicated purposes. XINU chooses to configure clock interrupts at interrupt number 32.

When a clock interrupt is generated, it is routed as interrupt number 32 to XINU's kernel code clkdisp.S in system/ which, in turn, calls clkhandler() in system/clkhandler.c to carry out relevant tasks. Modify clkhandler() in clkhandler.c so that a global variable, int fineclkcounter, declared and initialized to 0 in clkinit.c, is incremented when clkhandler() is called every 10 times. Thus fineclkcounter keeps track of time (in unit of 10 milliseconds) elapsed since XINU was bootloaded on a backend machine. Declare a second global variable, int vfineclkcounter, in clkinit.c initialized to 0, that is incremented in the clkdisp.S after the call to clkhandler() returns. Hence vfineclkcounter keeps track of the number of milliseconds elapsed since XINU was bootloaded. Test that fineclkcounter and vfineclkcounter are being updated correctly by XINU's modified clock interrupt handling code by making main() sleep for 5 seconds by calling sleep(), then printing the value of the two global variables.

3.3 Time slice management (20 pts)

An important task carried out by XINU's clock interrupt handler is keeping track of how much time slice (i.e., quantum) remains for the current process. When a process runs for the first time after creation, its time slice is set to QUANTUM which is defined in one of the header files in include/. Its default value is on the small side. Reconfigure it to 20 (msec). Unspent time slice of the current process is maintained in the global variable preempt which is decremented by clkhandler(). If preempt reaches 0, XINU's scheduler is called so that it can determine which process to run next. Extend XINU's process table data structure, struct procent, by adding a new field, uint32 prcpuhungry, that is initialized to 0 in create(). Whenever a process depletes its time slice, increment the field prcpuhungry in clkhandler(). Assess if prcpuhungry accurately tallies how CPU intensive a process is by creating a test case using main(). Describe in lab1.pdf your testing method.

3.4 Process lifetime (20 pts)

The lifetime of a process is the time that a process from creation until now (or termination) has existed. Introduce a new process table field, uint32 prbirthday, which is set to fineclkcounter from 3.2 when a process is created. Implement a system call, int32 lifetime(pid32), that returns the lifetime of a process whose PID is specified in the argument. lifetime() returns SYSERR (i.e., -1) if the argument is invalid. Test and verify that the system call works correctly.

3.5 Energy conservation (15 pts)

After nulluser() sets up the first process and spawns a child process running the code of main(), nulluser() enters into an infinite while loop, hence does not terminate. The purpose of the idle or NULL process in XINU -- the same holds for Linux/UNIX and Windows -- is so that there is always a process to run on a CPU when no other processes require CPU cycles. It may be that all other processes are sleeping, suspended, or blocking on events such as a response from a server. When an interrupt occurs XINU temporarily ceases running the code of the process executing on Galileo's x86 CPU. With the help of x86's interrupt vector XINU jumps to its clock interrupt handling code. After completion of interrupt handling returns to the code of the process that was interrupted to resume where it left off. We adopt the view that the interrupt handling code has been executed by the process whose own code was temporarily put on hold. We say that the interrupt handling code was executed by borrowing the context, or state, of the current process occupying the CPU. Thus except during bootloading before the first process is created, all code executed on a CPU is viewed as being carried out by some process. This includes even interrupt handling code that may have nothing to do with the currently running process. To enforce this abstraction, we use an idle or NULL process that is always ready to run on the CPU in case no other processes require CPU cycles.

In mobile devices, conserving battery power is important, hence a different strategy is needed than to just execute an infinite while loop in the idle process which expends energy. In server farms powered by electrical grid, an important consideration is cooling to dissipate heat generated by machines in confined spaces. A CPU that executes instructions consumes energy which generates heat. We will implement an energy consumption reduction method by putting the CPU in a halted state using the x86 hlt assembly instruction. The hlt instruction puts the CPU in a hibernating state until an external interrupt occurs. On our backends, this will likely be a clock interrupt which is generated at 1 msec intervals. To prevent XINU's NULL process from continually executing unconditional jumps (which is what an infinite while-loop with an empty body does), we add x86's hlt instruction to the body of the while loop. Until an interrupt occurs, the NULL process puts Galileo's CPU into hibernation which reduces energy consumption and heat generation.

A simple and efficient way of embedding assembly code within C code is to use in-line assembly. By adding the statement asm("hlt") to the body of nulluser()'s infinite while loop, we instruct the C compiler gcc to insert the hlt instruction when translating the C code into assembly (an intermediate step before generating machine code from the assembly code). An alternative method is code a function in assembly which executes hlt and call the function from nulluser(). However, function calls incur overhead which in-line assembly allows us to avoid. Modify the code of nulluser() to utilize in-line assembly to execute hlt inside the infinite while loop. Use the tests of 3.3 and 3.4 to gauge that the energy efficient idle process appears to be working correctly.

4. Interfacing C and assembly code (40 pts)

4.1 Calling assembly function from C function (20 pts)

Some aspects of operating system code cannot be implemented in C but require assembly coding to make the hardware do what we want. In simple cases such as 3.5, in-line assembly allows efficient integration of assembly instructions in C code. In more complex cases, coding functions in assembly may be needed that are then called from C functions. The opposite scenario may also arise where a function coded in C is called from a function coded in assembly. This style of programming which falls under ABI (application binary interface) programming -- as opposed to API programming -- is necessitated in parts of kernel programming where C does not suffice. We will practice coding a function in assembly so that it can be called from a function coded in C. This is also an exercise in checking that you understand how the CDECL caller/callee convention for our 32-bit x86 Galileo backends works. You will pick up the basic elements of AT&T assembly coding on x86 CPUs.

You are given a function addtwo() of function prototype, int addtwo(int x, int y), coded in AT&T assembly in addtwo.S in the course directory. Copy the file into your system/ directory and update include/prototypes.h. Call it from main() and verify that it works correctly. Beyond the algorithmic aspect of addtwo() (i.e., performing addition of two integers passed in the argument), note that addtwo() saves the content of EFLAGS (by executing instruction pushfl) and EBX registers onto the stack and restores them before returning to the caller. In x86 CDECL the calling function is responsible for saving and restoring the content of registers EAX, ECX, EDX. Hence the callee may use these registers without worrying about disturbing

their content. EBX, if utilized by the called function, is the responsibility of the callee. Some arithmetic operation related bits of the EFLAGS register may be changed by addtwo(), hence addtwo() incurs overhead to preserve and restore its content.

Using addtwo.S as a reference, code addfour() with function prototype, int addfour(int a, int b, int c, int d), in AT&T assembly in system/addfour.S that returns the sum of the four integer arguments. Annotate addfour.S with comments, and add your name and username at the beginning of the file. You may confer reference material) for background on AT&T assembly programming. Although assembly code syntax varies across different hardware and software platforms, the underlying logic behind ABI programming remains the same. It is this understanding that is important to acquire. Test and verify that addfour() works correctly.

4.2 Calling C function from assembly function (20 pts)

Code a C function, int greaterfirst(int x, int y), in system/greaterfirst.c where greaterfirst() returns 1 if the first argument x is strictly greater than the second argument y, 0 otherwise. Code a function, int testgreaterfirst(int a, int b), in system/testgreaterfirst.S in AT&T assembly which calls greaterfirst() with arguments a and b. testgreaterfirst(), in turn, is called by main() which prints the value returned by testgreaterfirst(). Note that in x86 CDECL the return value is communicated from callee to caller through register EAX. gcc will ensure that greaterfirst() puts 1 or 0 (depending on the outcome of comparing its two arguments) in EAX before returning to its caller testgreaterfirst(). Your assembly code of testgreaterfirst() must make sure that the value contained in EAX is not disturbed and communicated back to its caller main(). The main chore of testgreaterfirst() is to access the two arguments passed by main() and pass them to greaterfirst() before calling greaterfirst(). Test and verify that your code works correctly.

5. Run-time manipulation of return addresses (60 pts)

5.1 Wrong turn (30 pts)

In 4.2 main() calls testgreaterfirst() which, in turn, calls greaterfirst(). main() and greaterfirst() are coded in C whereas testgreaterfirst() is coded in assembly. Our aim is to modify the return address of testgreaterfirst() so that accessing the modified return address leads to x86 GPF (general protection fault) which, by default, terminates the offending process. In particular, GPF maps to interrupt (i.e., exception) 13 of x86's interrupt vector which causes interrupt handling code in intr.S at label _Xint13 to be executed. The code at _Xint13 jumps to code at Xtrap which then calls C function trap() in evec.c. trap() prints state information that may be useful for debugging and calls panic() in panic.c which performs an infinite while-loop. GPF is the most common run-time error you will encounter during XINU kernel programming in CS503.

First, make a copy of greaterfirst(), call it, greaterfirst1(), and place it in system/greaterfirst1.c. Just before the return statement of greaterfirst1(), add code that modifies the return address in its stack frame. The original return address will point to the instruction in testgreaterfirst() following the instruction "call greaterfirst". Change the address to a different value (e.g., 0) that is likely to trigger GPF. To do so, we will need to know where in main memory the return address of greaterfirst1() is located. Consult stacktrace.c in system/ that uses in-line assembly to copy the value of the EBP register into a C variable. gcc in compile/Makefile is configured to use EBP as frame pointer (i.e., base pointer in x86 terminology) that serves to identify the boundary between the stack frames of the caller and callee. The return address of the callee is located 4 bytes above the address contained in EBP in main memory. Hence we may surgically modify the return address of greaterfirst1() without disturbing the rest of the stack content. Implement the return address modification in greaterfirst1() and test that it works as intended.

5.2 Expressway to main() (30 pts)

Make a copy of greaterfirst1() and call it greaterfirst2() (in system/greaterfirst2.c). Instead of overwriting the original return address with one that leads to GPF, modify it so that it contains the return address of testgreaterfirst() (viewed as callee) to main() (the caller). Hence instead of greaterfirst2() returning to testgreaterfirst() which then returns to main(), greaterfirst2() directly jumps to the instruction in main() after the call to testgreaterfirst(). To do so, you will need to find the address in the run-time stack where the return

address of testgreaterfirst() to main() is located. If you coded testgreaterfirst() in testgreaterfirst.S so that the caller's EBP has been saved (pushed "above" the return address), the saved EBP value can be used to find the boundary between main()'s stack frame and testgreaterfirst()'s stack frame. Even if you did not save EBP in testgreaterfirst.S, since you coded the assembly code and know how many bytes are occupied by testgreaterfirst()'s stack frame, you can add this count to the EBP value of greaterfirst2() to find the address where the return address of testgreaterfirst() to main() is located. Thus, you are overwriting the same memory location as in 5.1 but with a different value that may not lead to GPF. To facilitate a clean jump directly from greaterfirst2() to main() without going through testgreaterfirst(), greaterfirst2() will need to reset the stack pointer to point to the top of stack frame of main() since side effects may occur otherwise. In 5.2 we will ignore the issue.

6. Modifying interrupt handling behavior (20 pts)

In x86 when a processor detects a divide-by-zero attempt, interrupt 0 is generated, called divide-by-zero exeption. In XINU, this causes code in intr.S at _Xint0 to be executed which, as in GPF, leads to a call to trap() and panic(). When divide-by-zero generates interrupt 0, hardware support pushes the content of EFLAGS, CS, and EIP registers onto the stack before jumping to _Xint0. If the code at _Xint0 were modified to include only instruction iret (i.e., interrupt return), hardware pops the saved EIP, CS, and EFLAGS values atomically and jumps to the return address in EIP. For the divide-by-zero exception, the return address pushed onto the stack is that of the divide instruction that triggered exception 0. Hence, an infinite loop will result where the process will attempt to execute the instruction that divides by zero which, again, generates interrupt 0. For some synchronous interrupts, pushing the address of the instruction that triggered it leads to desirable system behavior. A canonical example is page fault (exception 14 in x86) which is generated when a process makes a memory reference whose content does not reside in main memory but on disk. After the missing content is retrieved from disk and placed in main memory, we would want the process to re-execute the memory access instruction that generated the page fault. Since the missing content is now in main memory, the instruction will succeed.

Modify _Xint0 and verify using test code that divides by zero that when code at _Xint0 executes iret the system gets stuck in an infinite loop. In a second version, before executing iret insert code that adds a positive constant to the EIP value pushed onto the stack by hardware (i.e., return address). The goal is adding an offset that causes iret not to return to the divide instruction that triggered exception 0 but the next instruction. x86 instructions are of variable length, hence determining correct offsets to jump to the next instruction is involved. This will not be an issue since outside of this exercise we will know the addresses to jump to when implementing ROP (return oriented programming) kernel code. Describe your finding in lab1.pdf.

Bonus problem (25 pts)

On a frontend Linux PC, code a function, int createapp(char *, char **), in createapp.c that spawns a child process using fork(). The child process calls execvp() and passes the arguments of createapp() to execvp(). For example, calling createapp("ls", argv) where argv[0] points to string "ls", argv[1] points to "-l", and argv[2] is set to NULL will result in the child executing the code of /bin/ls with command-line option -l. The parent does not wait for the child but continues executing the next statement of createapp() which returns the PID of the child. If execvp() fails, the child calls exit(1) to terminate. If fork() fails (which is rare in today's systems) createapp() returns -1. Before calling execvp() the child calls nice() with a positive offset to decrease its priority relative to its parent.

A flaw of the above specification is that createapp() would return the PID of the child even if execvp() in the child fails. Fix the flaw and describe in lab1.pdf your method. Place createapp.c in lab1/. Test and verify that createapp() works correctly. Describe your method for testing that createapp() works correctly. Note that createapp() is similar to system() in Linux and CreateProcess() in Windows but with nontrivial differences.

Note: The bonus problem provides an opportunity to earn extra credits that count toward the lab component of the course. It is intended to help reach the maximum contribution of the lab component (45%) more easily. 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 above in lab1.pdf and place the file in lab1/. 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 lab1 (containing xinu-fall2022/ and lab1.pdf) is a subdirectory.

For example, if /homes/alice/cs503/lab1/xinu-fall2022 is your directory structure, go to /homes/alice/cs503

iii) Type the following command

turnin -c cs503 -p lab1 lab1

You can check/list the submitted files using

turnin -c cs503 -p lab1 -v

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

Back to the CS 503 web page