CS 354 Spring 2025

Lab 1: Getting Acquainted with XINU's Software and Hardware Environment [290 pts]

Due: 2/5/2025 (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, simple features of XINU processes, 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 1-3 from the XINU textbook.

For the written components of the problems below, please write your answers in a file, lab1ans.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.

Note: We will not be using xsh in the actual assignment. XINU code that invokes xsh will be deleted. xsh is but another app, not relevant for understanding the XINU kernel.

3.1 Basic system initialization and process creation [125 pts]

(a) *XINU initialization.* Inside the system/ subdirectory in xinu-spring2025/, you will find the bulk of relevant XINU source code that determines how XINU functions. The file start.S contains assembly code following AT&T syntax that is executed after XINU bootloads 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 label *start* in start.S. Eventually, start.S calls nulluser() in initialize.c which continues kernel initialization. nulluser() calls sysinit() (in initialize.c) where updates to the process table data structure proctab[] are made. 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 performed.

In a 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 when we delve into 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 executable 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[]. proctab[] is configured to hold up to NPROC processes, a system parameter.

There are three places in the source code where the CPP directive #define is used to set the value of NPROC: include/conf.h, config/conf.h, config/Configuration where Configuration is a text file. Change NPROC to 15 in include/conf.h and config/conf.h. Modify main.c in system/ after the call to kprintf() outputting "...creating a shell" and before calling recvclr() to print the value of NPROC. What do you observe? Check the value of NPROC in include/conf.h and

config/conf.h. What are their values? Check the last modified timestamp of the two files and use the diff command to compare their content. What do you find? Based on the discussion in class, what is the correct method for modifying the system parameter NPROC that sets the size of the kernel's process table? Perform this modification and verify that main() outputs 15 when printing NPROC.

(b) *XINU idle process and process creation.* sysinit() sets up the first process, called the idle or NULL 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 they are created by this special NULL process and its descendants through system calls, e.g., fork() in UNIX, 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. It is ground zero from which all processes emanate.

Code a new XINU system call, createminpid(), that has the same function prototype as create() but differs in that it allocates the smallest unused PID. Code createminpid() in system/createminpid.c. Legacy create() uses kernel function

newpid() to find an unused PID starting at the last PID allocated by create() which is remembered using a static variable. Thus your mod entails changing newpid(). Accomplish this by introducing a new kernel function, local int newpidmin(void), coded in system/createminpid.c. Since "local" is not part of the C language, it must be a macro defined using #define. Search the files in include/ for the keyword "local" to determine its actual C type.

To assess correct functioning of your createminpid() implementation use main() in main.c as your application layer test code. The TAs will use their own main.c to test your kernel modification. Describe in lab1ans.pdf your approach for gauging correct operation of createminpid() using test code in main(). In general, determining correctness of code is an impossible task. All we can do is try to perform due diligence adhering to high standards to the extent feasible. In CS 354 all lab assignments are exercises to help you learn and test your understanding, not production code to be released to the outside world. This affords the luxury of not having to allocate the bulk of your time to testing. However, you should be able to devise test cases that catch the low hanging fruit and catch subtleties that lead to high assurance of correctness. This is one

of the hardest parts of implementing complex software and competence is gained through practice.

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.

(c) *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 evaluating the impact of kernel modifications on application behavior as noted above. 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 directly relevant for understanding operating systems. Remove the "Hello World" message and the code that creates xsh starting from recvclr() until return OK. This will be the blank main() function going forward that will be coded to performance myriad app layer testing.

To specify who the author of a specific XINU version is, modify startup() in system/initialize.c before it creates a process running main() so that your username, name (last, first), "Spring, 2025" are output using kprintf() on a separate line where the tokens are separated by commas. For example,

asmith2001,Smith,Alice,Spring,2025

By default, carry over these modifications to subsequent lab assignments unless noted otherwise. When testing the modifications in (a) and (b), use the XINU version of (c).

(d) *Process termination.* We will consider what it means for a process to terminate in XINU. There are two ways for a process to terminate. First, a process calls exit() which is a wrapper function calling kill(), a system call, with argument returned by

getpid(). In the relevant part of kill() the current process (i.e., process in state PR_CURR executing on Galileo's CPU) removes itself from the process table and calls the XINU scheduler, resched(), so that it may select a ready process to run next. As noted in class, XINU selects the highest priority ready process that is at the front of the ready list to run next. The ready list is a priority queue sorted in nonincreasing order of process priority. Since XINU's idle process is always ready when it is not running (it only runs when there are no other ready processes since it is configured to have strictly lowest priority), the ready list is never empty unless the idle process executes on the CPU. Other chores carried out by kill() include closing descriptors 0, 1, 2, and freeing the stack memory of the terminating process.

Second, a process terminates "normally" without calling exit() (i.e., kill()), i.e., it executes the x86 return instruction ret. The first argument of create() is a function pointer which the newly created process, when chosen by the scheduler to run, will execute. How this is achieved is technically subtle and will be discussed when we consider context-switching under process management. With respect to termination, when the last instruction, ret, of the function pointed to by the first argument of create() is executed, it will return to a dummy function

userret() (in system/userret.c) who job is to facilitate orderly termination. As with exit(), this is done by calling kill(getpid()). The jump to userret() upon executing ret is accomplished by manipulating the stack of the newly created process so that it makes it seem as if userret() had called the newly created function. Inspect the code of create() and specify in lab1ans.pdf which statement of create() helps implement this illusion. Additional information pushed onto the stack after the address of userret() relates to context-switching which will be discussed later. To check that this is the case, code a function, void myuserret(void), in system/myuserret.c, that instead of calling kill() calls kprintf() to print the message "In myuserret()" then enters into an infinite while-loop. Modify create() so that the newly created function jumps to myuserret(), not userret(), upon executing ret. Specify in lab1ans.pdf how you accomplish this task. Using main() as your test function, verify that the system hangs when main() completes execution.

Suppose main() calls a function, int abc(void), that prints message "In abc." then returns to its caller. In your test case, suppose main() prints "About to return from main()." after calling abc(). Will your modification of create() allow "About to return from main()." to be output, or will abc() returning cause

the system to hang? Test this scenario and explain in lab1ans.pdf the result you observe.

(e) *Parent process ID.* create() remembers the PID of the process that created a new process in the latter's process table field, pid32 prparent. Perform a quick search of the header files in include/ to determine the C type of pid32 and note it in lab1ans.pdf. Code a new XINU system call, pid32 getppid(pid32), in system/getppid.c that takes the PID of a process as argument, finds its parent's PID and returns the value. Since the argument is not guaranteed to be a valid PID, sanity checks must be performed which incurs overhead but is critical for system calls which cannot behave unexpectedly no matter what arguments are passed. That is, a system call running in kernel mode behaving erratically compromises the entire system. To determine if the passed PID is valid, check its entry in the process table data structure, proctab, where the process state field prstate has value PR_FREE if the PID is invalid (i.e., process does not exist). getppid() returns SYSERR if the argument is not a valid PID. Is this the only sanity check that needs to be carried out? Explain your reasoning in lab1ans.pdf and modify getppid() accordingly.

When extending a kernel by making changes such as introducing new system calls, compatibility with the legacy kernel and any unintended side effects need to be examined. create() populates the prparent field of a newly created process correctly. This is not the case for a NULL/idle process where inspecting nulluser() in initialize.c shows that the prparent field is not explicitly set. Before implementing additional kernel modifications to make getppid() backward compatible, we need to consider what options there are to assign meaningful behavior when getppid() is called with argument 0. Discuss in lab1ans.pdf two options, significantly distinct from each other, choose an option and modify XINU (e.g., getppid(), functions in initialize.c) accordingly. Ignoring the issue will not be considered a meaningful option.

3.2 Round robin scheduling in XINU [45 pts]

Implement a modified version of the sample code involving main, sndA, sndB in Presentation-1.pdf discussed in class. Note that a parent process executing main() spawns two child processes by calling create() which execute functions sndA and sndB, respectively. Add an additional function, sndC, that outputs character 'C' to console. First, let startup() in initialize.c

call create() to spawn a child process to execute main() where the child's priority is set to 25. In sndA, sndB, and sndC, use kputc() in place of putc() to output a character. Set the priority of the process executing sndB to 18. The priority of the process executing sndA remains 20. The child process (of the process executing main()) which executes sndC is spawned by main() after creating the child process running sndB. Its priority is set to 20. Test your XINU code on a backend and describe what you find in lab1ans.pdf. Explain the results based on our discussion of how fixed priority scheduling works in XINU.

Second, repeat the above with the priority of the child process executing sndA set to 30. Explain the output behavior in lab1ans.pdf.

Third, repeat the first scenario with the priority of the child process executing sndB() set to 20. Explain your finding.

---

4. Kernel response to interrupts [60 pts]

4.1 Clock interrupt and system timer

As discussed in class, XINU as well as Linux and Windows kernels, can be viewed conceptually as being comprised of two

main parts: upper half code that responds to system calls and lower half code that responds to interrupts. We introduced simple changes to XINU's upper half in Problem 3. Here we will consider a basic operation of the lower half involving clock interrupt handling. When discussing the code examples/forever.c we noted that even though forever.c does not contain any system calls, a process executing its code may still transition from user mode to kernel code due to interrupts while the process is running. For example, a hardware clock may generate an interrupt that needs to be handled by XINU's lower half. We noted that a process is given a time budget, called time slice or quantum, when it starts executing. Detecting that a process has depleted its time budget is achieved with the help of clock interrupts which perform a countdown operation.

In computer architecture the term interrupt vector is used to refer to an interface between hardware and software where if an interrupt occurs it is routed to the responsible component of the kernel's lower half to handle it. In our case, the interrupt handling code of XINU that is tasked with responding to the clock interrupt is called system timer. The clock interrupt on our x86 Galileo backends is configured to go off every 1 millisecond (msec), hence 1000 clock interrupts per second. System timers

and their default values are kernel dependent. They will be discussed later in the course under clock management. For now, we will be concerned with its basic operation which is needed for understanding how kernels, including XINU, behave.

Our x86 backend processor supports 256 interrupts numbered 0, 1, ..., 255 of which the first 32, called faults (or exceptions), are reserved, by convention, for dedicated purposes. The source of interrupts 0, 1, 2, ..., 31 are not external devices such as clocks and network interfaces (e.g., Bluetooth, Ethernet) but the very process currently executing on Galileo's x86 CPU. For example, the process may attempt to access a location in main memory that it does not have access to which will likely result in interrupt number 13, called general protection fault (GPF), to be generated. The process may attempt to execute an instruction that divides by 0 which maps to interrupt number 0. XINU chooses to configure clock interrupts at interrupt number 32 which is the first interrupt number that is not reserved for faults.

When a clock interrupt is generated, it is routed as interrupt number 32 with the help of x86's interrupt vector -- called IDT (interrupt descriptor table) -- to XINU's kernel code clkdisp.S coded in assembly in system/. clkdisp.S, in turn, calls

clkhandler() in system/clkhandler.c to carry out relevant tasks. Declare a new global variable, unsigned int clkcounterfine, in clkinit.c and initialize it to 0. Modify clkdisp.S so that clkcounterfine is incremented when clkdisp.S executes. Global variables are simple to access in AT&T assembly as they can be referenced without declaration.

clkcounterfine is a software clock driven by a hardware clock. It does not get updated if clkdisp.S is not executed which can happen if XINU's hardware clock is temporarily ignored. That is, software can command the the hardware to silence or ignore interrupts. Some interrupts on x86 cannot be disabled, called NMIs (non-maskable interrupts), which are generated when catastropic hardware failures are detected. The more XINU disables clock interrupts, for whatever reason, the more inaccurate clkcounterfine becomes. Legacy XINU uses a global variable, uint32 clktime, that is updated in clkhandler() to monitor how many seconds have elapsed since a backend was bootloaded. Test and assess if both clkcounterfine and clktime provide consistent time using test code in main(). Make main() execute a lengthy for-loop before printing clkcounterfine and clktime. Experiment with loop bound (in the millions) until

clktime outputs several seconds. Describe your finding in lab1ans.pdf.

4.2 Clock interrupt disabling

As noted in 4.1, software clocks such as counters clkcounterfine and clktime become inaccurate when hardware clock interrupts are disabled. In x86, clock interrupts can be disabled by executing the cli assembly instruction. One method for embedding assembly code within C code is through inline assembly. By adding the statement asm("cli") before the for-loop in main(), you instruct gcc to embed cli before the assembly code for the for-loop that gcc generates (which then gets translated into machine code). Re-enabling the clock interrupt can be done by executing the assembly instruction sti. As a variation of 4.2, encapsulate your for-loop in main() with inline assembly code that execute cli and sti so that while the for-loop is executing clock interrupts are disabled. Compare the values of clkcounterfine and clktime against their values from 4.1. Discuss your finding in lab1ans.pdf.

4.3 Time slice management

An important task carried out by XINU's clock interrupt handler is keeping track of how much of a process's time budget (i.e., time slice or quantum) has been expended. If the time slice remaining reaches 0, clkhandler() calls XINU's scheduler, resched() in system/resched.c, to determine which process to execute next on Galileo's x86 CPU. 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 of 2 is on the small side. Unused time slice of the current process is maintained in the global variable, uint32 preempt, which is decremented by clkhandler() each time it is invoked by clkdisp.S. If preempt reaches 0, XINU's scheduler is called.

Rerun the third scenario of 3.2 where the fixed time slice of XINU's round robin scheduler, QUANTUM (defined in a header file in include/), is increased from 2 to 8. Compare the two results and discuss your finding.

---

5. Interfacing C and assembly code [60 pts]

5.1 Calling assembly function from C function

Some aspects of operating system code cannot be implemented in C but requires coding in assembly to make the system do what we want. There are two methods for invoking assembly code from C code: calling a function coded in assembly from a C function and inline assembly. Here we will look at the former. Inline assembly, briefly utilized in 4.2, will be further utilized in lab2. When programming entails interfacing C code with assembly code it falls under ABI (application binary interface) programming as opposed to API programming. When tasks cannot be accomplished in C alone and requires coding some parts in assembly, ABI programming becomes an important tool. We will only be needing basic elements of AT&T assembly on x86 CPUs. We will practice writing functions in assembly so that they can be called from functions coded in C. Coding in assembly to interface with C functions is also an exercise that helps check that you understand how the CDECL caller/callee convention on our 32-bit x86 Galileo backends works.

You are provided a function addfour() of function prototype, int addfour(int, int, int, int), coded in AT&T assembly in addfour.S in the course directory. It adds four integer arguments and returns the result to the caller. Copy the file into your system/

directory on a XINU frontend. Please remember to update include/prototypes.h. Call addfour() from main() and verify that it works correctly. Beyond the algorithmic aspect of addfour(), note that addfour() saves the content of the base pointer ebp since it will be set to the start of the new stack frame belonging to addfour(). Before addfour returns to its caller, EBP is restored to it previous value. addfour() accesses the four arguments passed on the stack and stores them into registers EAX, EBX, ECX, EDX. The content of EBX, ECX, EDX are added to EAX which will contain their summation. Per CDECL convention, the return value of a function is assumed stored in EAX.

The code of addfour() in addfour.S modifies the content of EAX, EBX, ECX, EDX. In x86 CDECL the calling function is responsible for saving and restoring the content of registers EAX, ECX, EDX. Hence the assembly code generated by gcc when compiling the call to addfour() from main() will save the content of EAX, ECX, EDX before calling addfour() and restore their original values after addfour() returns. Per CDECL saving and restoring EBX is the callee's (i.e., addfour()) responsibility. Although calling addfour() from main() as is may work and return the correct addition result, addfour() contains a bug that can surface if main() has been using EBX whose original value is needed after

the call to addfour() returns. Modify addfour() in addfour.S to correct the above bug. Explain in lab1ans.pdf the detailed logic behind your fix. Verify that your modified code works correctly.

You may check 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.

5.2 Calling C function from assembly function

Code a C function, int addfourC(int, int, int, int), in system/addfourC.c where addfourC() performs the same operation as addfour() but coded in C. Code a function testaddfourC(void) in system/testaddfourC.S in AT&T assembly which calls addfourC() with four constants 2, 3, 2, 2 as arguments. Hence addfourC() should return the value 9. testaddfourC(), in turn, is called by main() which outputs the value returned by testaddfourC(). By fixing the arguments passed by testaddfourC() to addfourC() we obviate the need to pass arguments from main() to testaddfourC() for simplicity sake since you practiced passing arguments to C functions to assembly code in 5.1.

Since the return value is communicated from callee to caller through register EAX in CDECL, gcc will ensure that addfourC() puts its result in EAX before returning to its caller testaddfourC(). Your assembly code testaddfourC.S need not touch EAX so that the result returned by testaddfourC() is propagated back to the caller of testaddfourC() (i.e., main()). If your testaddfourC() modifies EBX then, as with 5.1, ensure that its original value is restored by testaddfourC() before returning to main(). If EBX is not modified there is no need to do so. Test and verify that your implementation works correctly.

Bonus problem [25 pts]

Implement a version of create(), int32 creates(), in system/creates.c, where the arguments are the same as create() but for stack size, priority, process name which are omitted. Stack size is set to a new system parameter, PROCSTACKSZ, defined as 8192 in include/process.h, priority is set to the parent's priority plus 1, and name is set to "NONAME". Test and verify that creates() works correctly, then rerun the third scenario of 3.2. Discuss your finding in lab1ans.pdf.

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 lab1ans.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 #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 lab1 (containing xinu-spring2025/ and lab1ans.pdf) is a subdirectory.

For example, if /homes/alice/cs354/lab1/xinu-spring2025 is your directory structure, go to /homes/alice/cs354

iii) Type the following command

turnin -c cs354 -p lab1 lab1

You can check/list the submitted files using

turnin -c cs354 -p lab1 -v

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

Back to the CS 354 web page