CS 354 Fall 2023

Lab 1: Getting Acquainted with XINU's Software and Hardware Environment [265 pts]

Due: 9/13/2023 (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, 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.

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

3.1 Basic system initialization and process creation [100 pts]

(a) *XINU initialization.* Inside the system/ subdirectory in xinu-fall2023/, 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 label *start* in start.S. Eventually, start.S calls nulluser() in initialize.c which continues kernel initialization. nulluser() calls sysinit() (also 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 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 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 defined in config/ as 100. Change NPROC to 10.

(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 all other processes 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.

Code a new XINU system call, createppid(), that has the same function prototype as create() but for an additional argument of type, pid32, that is passed as the fourth argument:

pid32 createppid(void *funcaddr, uint32 ssize, pri16 priority, pid32 reqppid, char *name, uint32 nargs, ...);

The argument reqppid specifies the PID (process identifier) requested by the calling process to be its parent. That is, irrespective which process called createppid() to spawn a child, the child process specifies who it wants to be the parent. In Linux/UNIX the parent of a process may change if a process becomes an orphan, i.e., its parent terminates first. Orphaned processes are adopted, by default, by process 1. In XINU as in Linux/UNIX and Windows, system calls to spawn a new process do not allow a process to request to be adopted. The modified system call, createppid(), will allow that.

create() remembers the parent PID of a newly created in process table field, pid32 prparent. createppid() checks if the process with the requested parent PID, reqppid, exists. If so, the child's prparent field is updated to reqppid. If not, createppid() returns SYSERR (defined as -1). Of course, createppid() must check that reqppid falls within the valid range of PIDs 0, ..., NPROC-1, and return SYSERR if it does not. Code createppid() in createppid.c under system/. Compile a new XINU binary xinu.xbin on a frontend machine. Use main() in main.c as your application layer test code to assess correct operation of createppid() as noted in (c) below. The TAs will use their own main.c to test your kernel modification.

Important: 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 relevant for understanding operating systems. Remove the "Hello World" message from main() and put it in a separate function, void helloworld(void), in system/helloworld.c. Call helloworld() from nulluser() before it executes an infinite while-loop as idle process. Customize the "Hello World" message so that it contains your name, username, year and semester. Don't forget to insert the function prototype of helloworld() in include/prototypes.h. Modify main() so that it is completely empty but for a message that says "Test process: executing main() in process " followed by the PID of process. You may determine the PID of the current process by calling getpid().

Carry over these modifications to subsequent lab assignments unless specified otherwise. When testing the modifications in (a) and (b), use the XINU version of (c). For example, for part (a) use create() to spawn multiple process and verify that create() fails when the total number of processes exceeds 10. As noted above, the GTAs will use their own main.c to test your kernel modifications.

(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 containing system call kill(), kill(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()). Since the first argument of create() is a function pointer, normal termination implies returning from create() to its caller by executing the ret instruction in x86. The question is where does create() return to since it was not called by another function. For example, if a process calls create(main, 1024, 20, "main", 0, NULL) then a child process is created which executes the code of function main(). When main() executes ret where does it return to? 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 create(). This fake caller is identified by the macro INITRET whose definition can be found in a header file in include/. Playing detective, trace the sequence of events that transpire when a process terminates normally by create() executing the x86 ret instruction. Describe your finding in lab1.pdf. Does the fake caller of main() specified by INITRET return? Explain your reasoning. Are there system calls in Linux that, upon success, do not return? If so, describe the system calls and explain why they may not return. Make sure to distinguish system calls from user library functions.

(e) *Ancestry lineage.* Code a new XINU system call, int32 listancestors(pid32), in system/listancestors.c that uses the prparent process table field to output all the ancestor process IDs of the process specified in the argument. listancestors() returns the number of ancestor processes. Annotate your code, test and verify that it works correctly.

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 sndA and sndB, respectively. Add two additional functions, sndC and sndD, that output characters 'C' and 'D' to console, respectively. First, let startup() in initialize.c call create() to spawn a child process to execute main() where the child's priority is set to 30. In sndA, sndB, sndC, and sndD, use kputc() in place of putc(). Test your XINU code on a backend and describe what you find in lab1.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 main() set to 10. Explain your results.

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

---

4. Kernel response to interrupts [45 pts]

4.1 Clock interrupt and system timer

As discussed in class, XINU as well as Linux and Windows, 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.

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. Several years back a popular configuration was 10 msec. The default value varies across kernels. We will investigate the role of system timers in more depth when discussing device management later in the course. 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 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). 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 in system/. Code in clkdisp.S, in turn, calls clkhandler() in system/clkhandler.c to carry out relevant tasks. Declare a new global variable, uint32 clkcountermsec, in clkinit.c and initialized it to 0. Modify clkdisp.S so that clkcountermsec is incremented when clkdisp.S executes every 1 msec due to clock interrupt. Note that the software clock, clkcountermsec, does not get updated if XINU's hardware clock that acts as the system timer is disabled. The more XINU disables clock interrupts the more inaccurate clkcountermsec becomes as a timer that counts how many milliseconds have elapsed since a backend was bootloaded. 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 whether both clkcountermsec and clktime keep the same time using test code in main(). Describe your method for testing and your finding in lab1.pdf.

Note: Testing and verification of correctness of real-world software is rarely perfect. What we aim for is achieving a high standard which develops through practice.

4.2 Wall clock time termination

Code a new system call, uint32 wallclkterm(uint32), in system/wallclkterm.c. wallclkterm() compares clkcountermsec with the argument that was passed. If clkcountermsec is strictly less, wallclkterm() subtracts clkcountermsec from the argument and returns the result. Otherwise, wallclkterm() calls exit() to terminate the current process. Test and verify that your implementation works correctly.

As part of testing and an application of wallclkterm(), modify the first benchmark scenario of 3.2 so that sndA, sndB, sndC, sndD are not in a never-ending loop whose execution you have to manually terminated. Instead, the functions replace while(1) with while(wallclkterm(5000)) so that each process terminates 5 seconds after bootloading. Note that wallckltlerm() helps achieve synchronized termination, albeit approximately.

4.3 Time slice management and its size

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 2 is on the small side. Unused 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.

Rerun the first 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 10. Compare the two results and discuss your finding in lab1.pdf.

---

5. Interfacing C and assembly code [75 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 in-line assembly. Here we will look at the former. In-line assembly will be considered 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 controlling computing systems using C is not possible, ABI programming becomes an important tool. We will only be needing basic elements of AT&T assembly coding on x86 CPUs. We will practice coding functions in assembly so that they can be called from a function 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 for our 32-bit x86 Galileo backends works.

You are provided a function addtwo() of function prototype, int addtwo(int x, int y), coded in AT&T assembly in addtwo.S in the course directory. It adds the two 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 addtwo() from main() and verify that it works correctly. Beyond the algorithmic aspect of addtwo(), note that addtwo() saves the content of the EBX register onto the stack and restores it 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. The sample code addtwo.S shows typical steps that called functions need to consider performing. It is not optimized code where all unnecessary steps are removed.

Using addtwo.S as a reference, code addsub() with function prototype, int addsub(int, int, int), in AT&T assembly in system/addsub.S that returns the sum of the second and third arguments if the first argument equals 1. If the first argument is 0 addsub() subtracts the second argument from the third argument and returns the result. If the first argument is not 0 or 1, addsub() returns -1. Annotate addsub.S with comments, and add your name and username at the beginning of the file. 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. Test and verify that addsub() works correctly.

5.2 Calling assembly function from C function: version 2

A flaw of addsub() in 5.1 is that the return value -1 is overloaded, i.e., it could indicate an error condition or the result of adding/subtracting the second and third arguments. Implement a variant, int addsubv2(int, int, int, int *), in addsubv2.S where the fourth argument is a pointer to an integer variable where the result of addition/substraction is to be stored. addsubv2() returns -1 if there is an error (first argument is not 0 or 1). addsubv2() returns 0 if the calculation stored in the fourth argument is valid. Annotate addsubv2.S with comments, and add your name and username at the beginning of the file. Test and verify that addsubv2() works correctly.

5.3 Calling C function from assembly function

Code a C function, int addsubC(int, int, int), in system/addsubC.c where addsubC() performs the same operation as addsub() but coded in C. Code a function testaddsubC(int, int, int) in system/testaddsubC.S in AT&T assembly which calls addsubC() with three arguments. testaddsubC(), in turn, is called by main() which outputs the value returned by testaddsubC(). In x86 CDECL the return value is communicated from callee to caller through register EAX. gcc will ensure that addsubC() puts its result in EAX before returning to its caller testaddsubC(). Your assembly code testaddsubC() must make sure that the value contained in EAX is not disturbed and communicated back to its caller main(). testaddsubC() must access the three arguments passed by main() and pass them to addsubC() by placing them in the correct order on the stack before executing "call addsubC". Annotate your code as before. Test and verify that your implementation works correctly.

Bonus problem [20 pts]

From the viewpoint of a system programmer compare XINU's create() system call to spawn a new process to fork()/execve() and clone() in Linux, and CreateProcess() in Windows. Make create() the reference point, and ignore features in fork()/execve(), clone(), CreateProcess() that go beyond the basic tasks carried out by create(): create a separate process, specify what code the child should execute, its stack as the principal private resource, and any optional arguments to be conveyed to the child. Shared file descriptors, virtual memory, etc. are topics to be ignored. We will cover them later in the course. Which of the methods is closest to XINU's create()? Discuss your reasoning in lab1.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 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 #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 lab1 (containing xinu-fall2023/ and lab1.pdf) is a subdirectory.

For example, if /homes/alice/cs354/lab1/xinu-fall2023 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