CS 354 - Lab 1

Noah Cornett
cornettn

Problem 3.1

(a) Inside the system/ subdirectory, you will find the bulk of relevant XINU source code. The file start.S contains assembly code that is executed after XINU loads on a backend. After performing bootstrap/initialization chores (the details are not relevant at this time), a call to function nulluser() in initialize.c is made. At some point, nulluser() calls sysinit() (also in initialize.c). Jump to the code of sysinit() 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 one of the key data structures of an operating system where bookkeeping of all created processes is performed. In our Galileo x86 backend which has a single CPU, 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/ and include/ (header files). Some configuration related parameters can be found in config/. The total number of processes supported in XINU is defined by the constant NPROC. Do some detective work to locate the places where NPROC is defined. Based on what you find, determine how many processes can exist in XINU.

Inside the include/conf.h, NPROC is defined as 100. However, since conf.h is a generated file, in the event that NPROC is not defined in the generation of conf.h, NPROC is defined to be 8 in include/process.h. Therefore, the number of processes that can exist at the same time is 100.

(b) 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 on a frontend Linux PC for our target backend x86 CPU (i.e., the binary xinu.xbin). This includes machine code translated from initialize.c. 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 freed in proctab[]. After allocating the proctab[] data structure to hold up to NPROC processes, sysinit() creates the first process, called the NULL (or idle) process. This NULL process which exists not only in XINU but also in Linux/UNIX and Windows is the ancestor of all other processes in the system. It is the only process that is not created by the system call create() but is "handcrafted" during system initialization. Locate where the constant NULLPROC is defined and determine its value. One of the properties of a process is its priority. In XINU, the larger the priority value (an integer) the higher preference a process is given during scheduling when the kernel decides which process to allocate a CPU next. Using detective work, determine

the C type of of the priority field in the process table of XINU. Do not just provide the final answer but describe the procedure you followed to arrive at the answer. Specify the maximum and minimum priority values, at least in principle, that a XINU process may take. Where in this range does the priority of the NULL process fall?

NULLPROC is defined in include/process.h and its value is set to 0.

To start to find some information about the priority system, I started by looking for the declaration and initialization of the proctab[] data structure. After discovering that it is located in system/initialization.c, I look for the initialization of the entries of the table. After noticing that the entries of the proctab[] are of type procent, I look for the declaration of the procent type. Within the declaration of the procent type in include/process.h, I see that the priority field is set to type pri16. Using grep, I find that the pri16 type is type defined as being a 16 bit integer in include/kernel.h. A 16 bit integer can range anywhere from -32768 to 32767. So, in theory, the minimum value that a priority can be is -32768, and the maximum value is 32767. However, in the create() function, if the priority is set to be less than 1, the function returns a system error. Therefore, the functional limit of the priority is 0 through 32767.

Since the NULL process is set to have a priority of 0, the priority falls at the bottom of the usable range.

(c) At some point, nulluser() calls create() where the first argument, *startup*, is a function pointer to startup() whose code is contained in initialize.c. System call create() finds a free entry in proctab[] and creates a new process. In XINU, all processes created using create() start off in a suspended state which means they are not assigned to the CPU. The system call resume() changes the state of a newly created process to a ready state which allows it to compete with other ready processes for CPU cycle allocation. In our version of XINU, the only task performed by startup() is to create (and resume) a process running the function main() whose code is in main.c. Since creating a process running startup() adds overhead, create a process that runs main() from nulluser() directly. This will be your first modification of XINU which changes the source code in initialize.c. When making changes to source code, it is good practice, even if changes are minor, to annotate the code using comments what was done, by whom, when. Follow this practice when making mods to XINU. After making the change to nulluser(), verify that it works correctly. Any time changes to XINU or application processes running under XINU are made, XINU must be rebuilt on a front-end machine and reloaded on a backend machine. The process running main() prints a number of messages using kprintf() and then creates a process running a shell app using create(). Except in lab1, we will have no use for the shell app (it is just an application and not part of the kernel) nor the kprintf() messages at the start of main().

I will be using github as source control for myself.

Problem 3.2

- Remove the "Hello World!" welcome message that is output on console by main(). Instead, write a function, void mymotd(void), in mymotd.c under system/ which outputs using kprintf() a hello message containing your user name and name. Make a call to mymotd() from nulluser() just before it creates a process running main(). Since now you have added code residing in a new file, mymotd.c, under system/, the question arises as to whether changes need to be made to Makefile in compile/. In earlier versions of Makefile, new code files needed to be explicitly specified. In the current version, all files in system/ are automatically compiled and linked during a build. Verify that the mod works correctly. In lab assignments where new functionalities are added to XINU, this will entail adding files containing C and assembly code to system/ and header files to include/.

Problem 3.3

- In Linux/UNIX, to create a new process that runs an executable binary, say, /bin/ls, we first use the system call fork() to create a child process and then call the system call execve() with /bin/ls (or a.out) as an argument. As noted in Problem 3.1(c), in XINU a newly created process is put in a suspended state after it is created using create(). That is, the child process exists as an entry in the process table but it is marked as suspended (constant PR_SUSP in include/process.h). Being in suspended state has the consequence that XINU will not assign CPU cycles to the process until its state is changed to ready (PR_READY) by calling resume(). Two steps are needed to put a process into ready state so that XINU (i.e., its scheduler) eventually allocates CPU cycles to the process. The first step is to set the state of the process to PR_READY. The second step is to insert the process to a data structure called a ready list where other ready processes who are awaiting to receive CPU cycles are kept. Both steps are performed by the kernel function ready(). ready() is not a system call but a kernel function that is called by system calls and other kernel functions.
-
- Inspecting the code of resume() will show that it calls ready() after performing sanity checks to verify that the PID of the process to be resumed is valid and that the process in question is indeed in suspended state. Otherwise, an error status (value SYSERR) is returned. If resume() does not encounter a problem, it returns the priority of the readied process. The code of ready() shows that it changes the process state to PR_READY, calls kernel function insert() to add the newly readied process into XINU's ready list, and calls the kernel function resched() which determines if the CPU should be assigned to a different ready process. For example, if the just readied process has a higher priority than the current process occupying the CPU, then by the preference rule noted in Problem 3.1(b), the higher priority process is assigned the CPU and the current occupant is removed. That is, a context-switch is performed. Code a modified version of create(), pid32 rcreate(), with the same arguments as create() which puts the newly created process in ready state. To do so, at the end of create() just before calling restore(mask) (we will discuss the role of restore() separately), make a call to ready(). Do not call resume() from create(). Put the function rcreate() in rcreate.c under system/. Check that rcreate() works correctly. For example, instead of nulluser() calling resume(create((void *)main, INITSTK, INITPRIO, "Main process", 0, NULL)); it should suffice to call
-
- rcreate((void *)main, INITSTK, INITPRIO, "Main process", 0, NULL);
-
- rcreate() changes the flavor of XINU process creation to resemble the approach followed by Linux/UNIX and Windows where newly created processes are not suspended.

- In Linux where a child process is created using fork(), the question arises as to who runs first: child or parent? Experimentally gauge an answer to the question by writing and running test code on a front-end Linux machine in the XINU Lab where the parent process forks a child process. Make them perform tasks after fork() such as writing to terminal (file descriptor 1) using the write() system call. Output the priority of the parent and child processes by calling getpriority(). This helps gauge how Linux sets the priority of a child process. Keep in mind that a number of properties of the parent are inherited or shared by child processes. Also note that in Linux smaller integers mean higher priority. Do not try to find an authoritative answer from books or other formal sources. The point of this exercise is for you to experiment. For this task, it does not matter if your experiments lead to an incorrect conclusion. Put your test code, parentchild.c, in lab1/. In the rcreate() version of create() in XINU, which process will run first after rcreate() is called when the process running main() is created in 3.3(a)? Use detective work to find the definitive answer without relying on experimental evidence alone. This is different from the Linux exercise. Keep in mind the rule: if two processes are ready, the kernel picks one with higher priority. The case when priorities are equal will be discussed separately.

According to the documentation of the getpriority() function call, "a child created by fork() inherits its parent's nice value" which essentially means they have the same priority. From my investigation, the parent always runs first, followed by the child (in linux).

After rcreate() is called, the main process will run before the null process because it is set to have a higher priority. The main process is set to have priority INITPRIO, which is 20. This is a much higher priority than the null process, which has a priority of 0.

Problem 3.4

- When your XINU code running on a backend machine triggers a hardware interrupt, x86 is programmed by XINU to call interrupt handlers, mostly coded in assembly, in intr.S. For example, a divide-by-zero arithmetic operation is detected by the hardware as interrupt number 0 (interrupts are numbered 0, 1, 2, ..., 255) and x86 proceeds to consult a table called IDT (interrupt descriptor table) set up by XINU during initialization which specifies which interrupt handler to run. The same goes for Linux/UNIX and Windows. Interrupts that are generated as a result of the running program such as dividing by zero are called synchronous interrupts. "Synchronous" means that the currently running program caused the interrupt. In many cases, interrupts are asynchronous in the sense that their cause lies elsewhere such as a clock ringing, a message arriving, among other events. Intel uses the jargon "exception" or "fault" to denote the first 32 interrupts which are all of synchronous type. They include the familiar segmentation violation/fault which maps to interrupt, i.e., exception (or fault) number 13 that is triggered when a running program tries to reference memory that it does not have access to. The interrupt is synchronous to the instruction executed by the current process which caused it. Exception 13 is also called General Protection Fault.

- Interrupt handling code for interrupt number 0 (i.e., divide-by-zero) is located at label _Xint0 in intr.S. The interrupt handling code for exception number 0 is just 7 lines of assembly code. The last instruction, jmp Xtrap, is an unconditional jump/branch to Xtrap which contains 2 lines of code. The first instruction, call trap, is a function call from assembly code intr.S to C function trap() in evec.c. Thus the initial part of interrupt handling is coded in assembly and the latter part in C. Inspect the trap() function code in evec.c. When doing lab assignments, you will become enemy, and then friends, with code in intr.S and evec.c. In particular, a bug in your XINU code is likely to eventually end up as a call to trap(). The code in trap() prints select system state information that may help you track down bugs. trap() is passed two arguments by its caller Xtrap with help from assembly code at _Xint0 in intr.S. Explain how passing of the two arguments is accomplished following (i.e., being consistent with) the CDECL convention. Note that code in evec.c is translated in machine code by gcc. The code in intr.S that is translated into machine code by the assembler. They must interface correctly to work correctly. Modify trap() to print your user name, name, and the XINU system variable clktime which is a global variable that keeps track of the number of seconds that have passed since you bootstrapped XINU on a backend machine. Trigger interrupt 0 by performing divide by zero at the beginning of main(). Confirm that the system reaches the code in trap(). The last operation that trap() performs is to call the kernel function panic(). Based on the code your find in panic(), explain what happens to XINU at the end dividing by zero in an app generates interrupt 0.

When making calls in CDECL, arguments are passed into the stack in a right to left order. So in this specific example, trap(int inum, long *sp)'s arguments would be pushed onto the stack in this order: sp, inum. The arguments being passed from right to left are consistent with the CDECL convention.

Based off of the code that I found in panic(), panic() just sets the current process into an infinite loop that does nothing. After triggering the interrupt, I notice that panic just sets the whole system in halt via said infinite loop.

Problem 3.5

- In Problem 3.4, we considered what happens to a process and XINU as a system when interrupt 0 is generated in x86 by executing a divide by zero instruction. In this problem we will consider what happens if a process does not generate a synchronous interrupt and terminates by completing its last instruction. Informally we would say that a process terminates normally. For example, in the case of a process created using create() and running function main(), we are interested in determining what happens after main() returns. In XINU, there is nothing special about main() -- it is just another function -- and a process can be created to run any function. We are interested in determining what happens when this arbitrary function returns as a result of completing the last instruction to be executed by the process. Keep in mind that XINU is compiled using gcc and follows the CDECL function call convention.

- To answer this question, we have to look at how the create() system call creates a new process which entails myriad tasks. When looking at create(), in the example where the function pointer of main() is passed as first argument of create(), we find that XINU creates the illusion that main() was called by another function by pushing the function pointer (i.e., address) of this virtual calling function onto the stack of the process running main() as the return address of main(). Looking at the code of create() where content is pushed onto the run-time stack of a process when it is created, determine the name of the function that XINU sets up as the function that called main(). Inspect the code of this virtual caller in system/ and determine what it does. In the special case where the process that terminates is the last process outside the NULL process, determine what happens by continuing to follow XINU's source code. In the chain of functions being called starting with the virtual function to which main() returns, what is the last function that is called, where is its source code, and what does it do? In environments where power consumption is an important issue -- e.g., battery powered mobile devices, or cloud computing servers where overheating must be prevented -- is XINU's approach to "shutting down" after the last process has terminated a good solution? What might be a better solution? Explain your reasoning.

The name of the function that XINU sets up as the calling function is userret(). When all of the functions are done, and userret is finally returned to, it kills the current process. The chain of functions that is called is userret() -> getpid() -> kill() -> xdone() -> halt(). So, the last function that is called is halt(), and it’s source code is found in system/intr.S. All that this function does, is recursively call itself forever causing XINU to do nothing forever.

This is not a very good solution to “shutting down” because XINU is still running assembly instruction over and over. Instead of this solution, calling a function that

physically powers off the the system instead of looping assembly instruction. This way, there is no extra power lost. Whenever the system needs to be rebooted, the whole entire initialization sequence would have to be launched.

Bonus

- In Linux/UNIX system programming, we say that the fork() is special in that it is the only system call that "returns twice." What does that mean? Why does this feature of fork() not apply to XINU's create()? We say that Linux/UNIX's execve() system call is special in that it is the only system call that does not return when it succeeds. What does this mean? Why does this not apply to XINU's create()?

Fork “returns twice” because both the child and the parent process return after the call to fork(). This feature does not apply to the create function because the create function sets up the processes return address to be a different function, so the create() function only returns once.

Execve() does not return because the current process’s stack is reinitialized to run the provided command. So, with the reinitialized stack, evecvp()’s return address is overwritten, thus it does not return. This does not apply to XINU’s create() because create still returns the PID of the process created.