Lab1 Han Wang(wang2786 0028451697)

Problem 3.1 (30 pts)

Follow the instructions in the XINU set-up description which compiles the XINU source code on a frontend machine (Linux PC) in the XINU Lab, grabs an unused x86 Galileo backend machine, and loads and bootstraps the compiled XINU image. Note that the frontend PC's terminal acts as a (remote) console of the dedicated backend Galileo x86 machine. If XINU bootstraps successfully, it will print a greeting message and start a simple shell called 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.**

Inside the system/ subdirectory, you will find relevant source code of XINU. 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() is made.

Locate where in system/ the source code of nulluser() is defined.

Ans: nulluser() is defined in initialize.c file in /system

Toward the end of nulluser(), it calls the function create() -- a system call as discussed in class -- to create a new running program (i.e., process) whose code is the function main() contained in main.c. All processes in XINU are created using create() except the process that runs the function nulluser() which is "self-made." That is, it is the ancestor of all other processes.

Find out where in the XINU source code this ancestor of all processes is located and what name, i.e., process ID (PID), it assigns itself. To do so, track down the header file (header files are located in the include/ directory) where the PID value of the ancestor process is specified.

Ans: Ancestor of all processes’s name is NULLPROC and it is located at /include/process.h or PID=0 PPID=0.

What does the ancestor process running the function nulluser() do for the rest of its existence?

Ans: The ancestor process keeps running for the rest of its existence, and it cannot be killed.

Does nulluser() ever return after being called by the code in start.S?

Ans: No, it does not because it is a void function and the process created by nulluser() is a process that keeps running and cannot be killed.

halt() is called after nulluser() in start.S. Where is the source code of halt() located and what does it do?

Ans: halt() is located in system/intr.S which will let the process do nothing forever. It halts the process.

What happens if you remove this function call from start.S?

Ans: It does not change the functionality of the os.

Does XINU run as before?

Ans: Yes.

What happens if you replace the while-loop at the end of nulluser() with a call to halt()?

Ans: It still works.

Problem 3.2 (50 pts)

In Linux/UNIX, to create a new process that runs an executable binary, say, /bin/ls (or a.out), 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. In XINU we said that a newly created process is put in a "state of limbo," i.e., suspended, 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 (PR_SUSP). The definition of all XINU process states is specified in process.h under include/. 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(). Look at the code of create() and resume() and determine how this is done.

What happens in Linux after a new process is created using fork()?

Ans: It makes two identical address space for the parent and the child, the processes start the execution after the fork() statement.

Note, create() specifies what code to run as the child process through its first argument (a function pointer). Determine who runs first: parent or child. Try to empirically gauge the answer by running test code on the frontend Linux PCs. (We will discuss scheduling

of processes in modern operating systems, a complex topic, when discussing process management.)

Ans: After testing, the results shows that both process run almost the same time, but parents always run first.

As an app programmer, do you have a preference as to which process -- parent or child -- should run next in Linux? Explain your reasoning.

Ans: It depends on the priority of the process and different functionality of the task. But in most cases, my personal preference would be the child process, since its easier to manipulate.

For a child process created using fork() in Linux, it has to call execve() to make the child run executable code contained in a file (e.g., /bin/ls) as part of a file system specified in the argument of execve(). In Linux, sometimes an app programmer just wants to create a new process to run an existing binary without going through the 2-step procedure of fork() and execve(). Write a wrapper function, int newProcess(const char *filename), that internally calls fork() and execve() to make this happen where filename specifies the full pathname of an executable binary. Place the code of newProcess() in newProcess.C under system/. Annotate your code with comments to facilitate readability.

How is newProcess() fundamentally different from the way XINU's create() works? Ignore stack size and process priority which are specified as arguments in create().

Ans: In newProcess, process are created immediately once they are created, and on the other hand, Create() is in a suspened mode waiting for resume. Furthermore, newProcess is working under 64-bit environment, and as XINU, its a 32-bit environment.

How does Linux's clone() compare to XINU's create()?

Ans: Linux’s clone() allows the child process to share parts of its execution context with the calling process(virtual address space, table of fd, table of signal handler) and on the other hand XINU create() has stack address, thread ID and pointer to the new block.

How does Linux's posix_spawn() compared to your newProcess() implementation?

Ans: Posic_spawn are more specified to provide a standardized method of creating new processes on machines that lack of the ability of support of fork(), which newProcess() uses. It is more suitable for small embedded systems like XINU. It combines fork and exec with other optional housekeeping steps in the child process before exec(). But

functionality wise, it is only a subset of the functionality of system calls (fork and execve) can provide.

Since the newProcess() we implemented did not use environment vars and func args(as NULL), the posix_spawn and my newProcess() implementation are the same.

Is there a best way to create processes? Explain your reasoning.

Ans: In my opinion, the best way of creating a process is not only the ability to generate separate child processes to execute other program, but also be able to track the process, control the state of the new process with sufficient cost using correct pointer and address table.

Problem 3.3 (5 pts)

Customize the welcome message printed by XINU in main() so that it prominently displays your name (last, first) and username. Preserve the new welcome message as part of your version of XINU during the rest of the semester and the labs to follow.

Problem 3.4 (30 pts)

Problems 3.1 and 3.2 looked at the overall picture and steps involved in starting XINU and running apps under XINU. To get a sense of how to terminate processes and XINU itself, create a process that runs the code of a function onandon() by calling create() with stack size 2048 and priority 30. onandon() should be an infinite loop that makes a XINU system call putc(CONSOLE, 'x') followed by system call sleep() with argument 2 within its loop. onandon() prints the character 'x', sleeps for 2 second, repeating this forever. The process running the function main(), after creating a child process that runs function onandon(), sleeps for 14 seconds, then calls XINU's kill() with the PID of the child process as argument. To verify that the child process has been terminated, check the process table data structure proctab. As in Problem 3.1, inspect the code where nulluser() resides to determine how proctab is initialized and how to access it. Put this version of main() in main1.c under system/. The same goes for onandon() in onandon.c.

Bonus problem [15 pts]

Ans:
My new command are:

1. a addition of integers command. Using “add val1 val2” to use. If there is a invalid input, user will get prompt. Documentation is in the source code.
2. a subtraction of integers command. Using “minus val1 val2” to use. If there is a invalid input, user will get prompt. Documentation is in the source code.
3. a multiplication of integers command. Using “times val1 val2” to use. If there is a invalid input, user will get prompt. Documentation is in the source code.
4. a division of integers command. Using “divided val1 val2” to use. If there is a invalid input, user will get prompt. Documentation is in the source code.

Note: All these commands are well edge-checked, if user input “add 134d 123” or “add 132 5s81”, program will count that as a invalid also, and prompt where went wrong.