ECE465/565 – Operating Systems Design: Project #1

Due date: September 17, 2023

Objectives

- To become familiar with the general structure of Xinu codebase;
- To become familiar with Xinu's process management;
- To practice writing and understanding test case files;
- To understand the handling of the stack.

Reminders on Code Reuse, Testing and Use of ChatGPT.

Code reuse: This course has a **strict no-code reuse policy**: ***consulting or reusing*** (even partially) code from other individuals or online resources is not allowed (and is considered an academic integrity violation).

Testing: Your code will be graded using the course's VCL image. Please make sure you test your code on the VCL before submitting.

ChatGPT use: The use of ChatGPT (https://chat.openai.com) is allowed with the restrictions indicated in the syllabus (please consult the syllabus for guidelines). If you use ChatGPT, please list all the questions entered in it in the following form: https://forms.gle/HgGTSVzF8CynFiCq5.

Getting Oriented in Xinu

Types and Constants

All header files are in the `include` directory. You may want to start from the following header files:

- `include/xinu.h:` unifies the inclusion of all necessary header files.
- `include/prototypes.h`: declares most system-call prototypes.
- `include/kernel.h`: contains definition of some important constants, types, and function prototypes.

For Debugging

See `kprintf()` system call in `system/kprintf.c`

Process-related code

- Process definition and PCB table: `include/process.h`
- Process creation/termination/suspension/resumption: `system/create.c`, `system/kill.c`, `system/exit.c`, `system/suspend.c`, `system/resume.c`

- Process scheduling: `include/resched.h, system/resched.c`
- Context switch and state change: `system/ctxsw.S, system/ready.c, system/sleep.c, system/unsleep.c, system/wait.c, system/yield.c, system/wakeup.c`
- Process queue management: `include/queue.h, system/queue.c, system/newqueue.c, system/getitem.c, system/insert.c`

Bootstrap procedure

The startup code (`system/start.S`) invokes `nulluser()` in `system/initialize.c` to initialize the system. Analyze the initialization code in `initialize.c.`

Questions (include the answers to these questions in the report):

Q1. What is the maximum number of processes accepted by Xinu? Where is it defined?

Q2. What does Xinu define as an "illegal PID"?

Q3. What is the default stack size Xinu assigns each process? Where is it defined?

Q4. Draw Xinu's process state diagram.

Q5. When is the shell process created?

Q6. Draw Xinu's process tree (including the name and identifier of each process) when the initialization is complete.

Q7. (Please see Q7 below)

Coding problems

P1: Cascading termination

The goal of this problem is to implement cascading termination. For this problem, we will group processes in two categories: *system processes* and *user processes*. We will call "system processes" the processes spawned by default by Xinu (e.g., `startup`, `main`, etc.), and "user processes" the ones spawned by the `main` process (and used to model user code). You can consider the `shell` to be a system process. To distinguish between the two categories of processes, add a `user_process` flag, of type `bool8`, to the PCB, and set the value of this flag to `TRUE` for user processes, and to `FALSE` for system processes. You can modify the `create` function to set this flag to `FALSE` by default, and later modify it as needed.

1. Modify Xinu to support cascading termination *for user processes only*.
2. Write a test case file to validate your implementation. Your test case file should:
   a. Spawn at least 15 user processes using the `create` system call. The structure of the process tree is up to you, but should be designed to allow for testing representative scenarios (e.g., termination of a leaf process, etc.)
   b. Kill select processes to test representative scenarios.
   c. Print the list of active processes before and after process termination. For each active process, print the identifiers of its children.
   d. Do **not** use the `shell` process in your test cases.

In your submission, the test case file should be named `main.kill`. Include in the report an illustration of the process tree used to test your code. If you don't know how to structure the test case file, have you look at `main.fork` as a reference. You will use `main.fork` to test your code for P2.

P2: Process creation and stack handling

Implement a *fork* system call similar to Unix's *fork*. The implementation should be in a separate `fork.c` file within the `system` folder. The function declaration must be as follows:

```
pid32 fork()
```

The fork primitive creates a new process (the child) by *almost* duplicating the parent process. Note that Xinu's processes are essentially threads: they share the memory address space but have *private stacks*.

Initialization: The child process should be initialized to have the same name, priority and stack length as the parent process. The *prsem*, *prhasmg* and *prdesc* fields of its process control block (PCB) should be initialized as during standard process creation (to `-1`, `FALSE` and `CONSOLE`, respectively). The child process should be set in READY state (and inserted in the ready list) upon creation. The ready list contains the processes that are eligible for execution. Have a look at `ready.c` (line 25) to see how to add a process to the ready list.

Return value: On success, *fork* must return the PID of the child process to the parent, and value `NPROC` to the child (note: differently from Unix's *fork*, the function you are coding does not return value 0 to the child because that's a PID in use). On failure, *fork* returns SYSERR.

Execution: After creation, *the child process should resume execution starting from the first instruction following the fork system call.*

Testing: Test cases for the *fork* system call are provided in `main.fork`. To run these test cases, rename `main.fork` into `main.c`, and compile the code. Note that you can selectively disable the test cases by commenting out the respective "`#define TESTCASEx`" macro definition at the beginning of the file. The expected output is contained in the provided `fork.output` file.

Hints: This problem requires a good understanding of the stack and its handling by Xinu. To this end:

- You can find information on the x86 architecture and assembly in the Intel Architecture Software Developer's Manual (available on the course website). Refer to Volume 1, Chapters 4.1-4.3 for information on the handling of the stack. You can also find a brief introduction on the runtime stack in the Xinu's book, Chapter 3.9.1.
- Have a look at `create.c` and `ctxsw.S` to see how the stack is initialized and how it is handled upon context switch.
- The `stacktrace` system call (in `stacktrace.c`) prints the stack backtrace for a given process, and can help you understand how to traverse the stack.
- Have a look at the three test cases in `main.fork` and make sure that you fully understand their expected output (in `fork.output`).
- If you want to the see assembly file corresponding to any Xinu file, you can proceed as follow. First, modify the `Makefile` (in the `compile` folder) and add "`-S`" to the `CFLAGS` compiler flag. Second, compile using "`make clean; make`;". After compilation, the `.o` files in the `binaries` subfolder will contain assembly (rather than object) code.

Question Q7: what is the effect of the "`receive()`" call in the test cases provided? What would happen if that function call was not present? *Hint*: see `receive.c` and `send.c` files. Include the answer to this question in the report.

Submission instructions

1. Create a `testcases` folder in the `system` folder and include the test case related files (`main.fork, fork.output and main.kill`) in it.
2. **Important:** You can write code in `main.c` to test your functions, but please note that, when we test your code, we may replace the `main.c` file. Therefore, do not implement any essential functionality in the `main.c` file. Also, turn off debugging output before submitting your code.
3. Go to the `xinu/compile` directory and invoke `make clean`.
4. Create a `xinu/tmp` directory and copy all the files you have modified/created (both .h files and .c files) into it (the `tmp` folder should have the same directory structure as the `xinu` folder). For example, if you have modified `Makefile`, `system/create.c` and `open.c`, your `xinu` directory will look like:

```
-xinu
   -[existing folders]
   -tmp [this is the folder you created]
     -compile
          Makefile
     -system
           create.c open.c
        -  testcases
            main.kill
```

   Note that the use of the `tmp` folder aims to help the TA to quickly identify what files have been modified. Please **do not delete** the files that you have created/modified from the original folders (i.e., the `xinu/include`, `xinu/system`, etc., you have been working on).

5. The project report should contain:
   - The answer to all questions posed in this assignment (Q1-Q7);
   - For each coding problem, a **brief** description of your implementation (which files have you added/modified? What are the main data structures used by your implementation?);
   - For P1, the illustration of the process tree used to test your code.
6. Go to the parent folder of the `xinu` folder. Compress the whole `xinu` directory into a *tgz* file.

   `tar czf xinu_project1.tgz xinu`

7. Submit your code and report (in pdf format) through Moodle. *Please do not include the report in the `xinu_project1.tgz` archive; submit code and report as two separate files.*