Homework Assignment #1

J. H. Wang

Mar. 16, 2023

Homework #1

- Written exercises:
  - Chap.1: 1.9(a)(b)(c)
  - Chap.2: 2.7, 2.10
  - Chap.3: 3.1, 3.11(a)(b)(c)(d)
- *Programming exercises:
  - Programming Problems: 2.15*, 3.14* (, 3.15**, 3.20**)
    - Note: Each student must complete all programming problems on your own
  - Programming Projects for Chap. 2* & Chap. 3*
    - [Team-based] To select at least one programming project from each chapter
- Due: two weeks (Mar. 30, 2023)

Written Exercises

- Chap.1
  - 1.9: *Direct memory access* is used for high-speed I/O devices in order to avoid increasing the CPU's execution load.
    - (a) How does the CPU interface with the device to coordinate the transfer?
    - (b) How does the CPU know when the memory operations are complete?
    - (c) The CPU is allowed to execute other programs while the DMA controller is transferring data. Does this process interfere with the execution of the user programs? If so, describe what forms of interference are caused.

- Chap. 2
  - 2.7: What are the two models of *interprocess communication*? What are the strengths and weakness of the two approaches?
  - 2.10: What is the main advantage of the *microkernel* approach to system design? How do user programs and system services interact in a microkernel architecture? What are the disadvantages of using the microkernel approach?

- Chap. 3
  - 3.1: Describe the differences among short-term, medium-term, and long-term scheduling.

- 3.11: What are the benefits and the disadvantages of each of the following? Consider both the system level and the programmer level.
  - (a) Synchronous and asynchronous communication.
  - (b) Automatic and explicit buffering
  - (c) Send by copy and send by reference
  - (d) Fixed-sized and variable-sized messages

Programming Problems

- Chap. 2
  - 2.15*: In Sec.2.3, we described a program that copies the contents of one file to a destination file.
    - This program works by first prompting the user for the name of the source and destination files.
  - Write this program using either Windows or POSIX API.
    - Be sure to include all necessary error checking, including ensuring that the source file exists.
      (… to be continued)

(… continued from the previous slide)

– Once you have correctly designed and tested the program, if you used a system that supports it, run the program using a utility that traces system calls.
  - Linux systems provide the *strace* utility, and
  - Solaris and Mac OS X systems use the *dtrace* command
  - As Windows systems do not provide such features, you will have to trace through the Windows version of this program using a debugger.

- Chap. 3:
  - 3.14*: The Collatz conjecture concerns what happens when we take any positive integer n and apply the following algorithm:

    $n = n/2$, if n is even
    $n = 3*n+1$, if n is odd

    The conjecture states that when this algorithm is continually applied, all positive integers will eventually reach 1.
    For example, if n=35, the sequence is: 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1.

    (…to be continued…)

- Write a C program using the fork() system call that generates this sequence in the child process. The starting number will be provided from the command line.
  - For example, if 8 is passed as a parameter on the command line, the child process will output 8, 4, 2, 1.
- Because the parent and child processes have their own copies of the data, it will be necessary for the child to output the sequence.
- Have the parent invoke the wait() call to wait for the child process to complete before exiting the program.
- Perform necessary error checking to ensure that a positive integer is passed on the command line.

- [optional] (3.15**): In Exercise 3.14, the child process must output the sequence of numbers generated from the algorithm specified by the Collatz conjecture because the parent and child processes have their own copies of the data.
- Another approach to redesign this program is to establish a shared-memory object between the parent and child processes. This technique allows the child to write the contents of the sequence to the shared-memory object. The parent can then output the sequence when the child completes.
- Because the memory is shared, any changes the child makes will be reflected in the parent process as well.

(…to be continued…)

- This program will be structured using POSIX shared memory as described in Sec.3.5.1. The parent process will progress though the following steps:
  - Establish the shared-memory object (shm_open(), ftruncate(), and mmap())
  - Create the child process and wait for it to terminate
  - Output the contents of shared memory
  - Remove the shared-memory object
- One area of concern with cooperating process involves synchronization issues. In this exercise, the parent and child processes must be coordinated so that the parent does not output the sequence until the child finishes execution. These two processes will be synchronized using the wait() system call: the parent process will invoke wait(), which will suspend until the child process exits.

- [optional] (3.20**): Design a file-copying program named filecopy using ordinary pipes.
  - This program will be passed two parameters: the name of the file to be copied, and the name of the copied file
  - The program will then create an ordinary pipe and write the contents of the file to be copied to the pipe
  - The child process will read this file from the pipe and write it to the destination file

  (…to be continued)

- For example, if we invoke the program as follows:
  - filecopy input.txt copy.txt
  - The file input.txt will be written to the pipe. The child process will read the contents of this file and write it to the destination file copy.txt.
- You may write this program using either UNIX or Windows pipes.

Programming Projects

- Programming Project for Chap. 2:
  Project 1: Linux Kernel Modules
  - Part I: Creating, loading, and removing kernel modules
    - Assignment: Proceed through the steps described above to create the kernel module and to load and unload the module. Be sure to check the counters of the kernel log buffer using *dmesg* to ensure you have properly followed the steps.

- Part II: Creating, traversing, and deleting kernel data structures
  - In the module entry point, create a linked list containing five *struct_birthday* elements. Traverse the linked list and output its contents to the kernel log buffer. Invoke the *dmesg* command to ensure the list is properly constructed once the kernel module has been loaded.
  - In the module exit point, delete the elements from the linked list and return the free memory back to the kernel. Again, invoke the *dmesg* command to check that the list has been removed once the kernel module has been unloaded.

Programming Projects

- Programming Project for Chap. 3 (Choose one)
- Project 1: UNIX Shell and History Feature
  - This project consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process
  - Part I - Creating a child process to execute a command
    - The first task is to modify the *main()* function is Fig. 3.35 so that a child process is forked and executes the command specified by the user.
  - Part II - Creating a history feature
    - The next task is to modify the shell interface program so that it provides a *history* feature that allows the user to access the most recently entered commands.

- Project 2: Linux Kernel Module for Listing Tasks
  - In this project, you will write a kernel module that lists all current tasks in a Linux system.
    - Be sure to review the programming project in Chap.2 before you begin this project
    - It can be completed using the Linux virtual machine provided with the textbook
  - Part I - Iterating over tasks linearly
    - Assignment: Design a kernel module that iterates through all tasks in the system using the *for_each_process()* macro. In particular, output the task name (known as executable name), state, and process id of each task. Write this code in the module entry point so that its contents will appear in the kernel log buffer, which can be viewed using the *dmesg* command.

- Part II - Iterating over tasks with a depth-first search (DFS) tree
  - Assignment: Beginning from the *init* task, design a kernel module that iterates over all tasks in the system using a DFS tree. Just as in the first part of this project, output the name, state, and *pid* of each task. Perform this iteration in the kernel entry module so that its output appears in the kernel log buffer.
  - To verify that you have indeed performed an appropriate DFS iteration, you will have to examine the relationships among the various tasks output by the ps command.

Notes for Programming Exercises

- Some programming exercises (programming problems 3.14, 3.15, and all programming projects in Chap. 2 and 3) require C programming in Linux/UNIX
- You are suggested to use the Linux virtual machine provided with the textbook

Homework Submission

- For hand-written exercises, please hand in your homework on paper in class
- For programming exercises, please upload your program to the iSchool+ as follows:
  - Program uploading: a compressed file (in .zip format) including source codes, execution snapshot, and documentation
  - The documentation should clearly identify:
    - Team members and responsibility
    - Compilation or configuration instructions if it needs special environment to compile or run
  - Please contact with the TA if you are unable to upload your homework

Thanks for Your Attention!