Assignment 2 - IPC
CS-351
Fall 2023

Due Date: 12/17/2023 at 11:59 pm

You may work in groups of 6

Only one group member needs to submit

Outcomes: after completing this assignment you will be able to:

1. Describe the function of signals, shared memory, message queues, and pipes interprocess communications mechanisms
2. Utilize POSIX shared memory, message queues, and pipes to implement solutions to practical problems
3. Design and implement a multiprocess applications that use interprocess communications for exchanging data.

Knowledge Gained/Exercised:

1. Knowledge of operating systems
2. Knowledge of operating system command-line tools
3. Knowledge of programming language structures and logic
4. Knowledge of software debugging principles
5. Knowledge of secure coding techniques

The knowledge and skills gained in this assignment are useful in the following work roles:

1. Software Developer
2. Secure Software Assessor
3. System Administrator
4. Cyber Defense Analyst
5. Data Analyst
6. Law Enforcement/Counterintelligence Forensics Analyst
7. Cyber Defense Forensics Analyst
8. System Administrator

9. Information Systems Security Manager
10. Cyber Defense Analyst
11. Cyber Defense Incident Responder
12. Vulnerability Assessment Analyst

Overview

In this assignment you will apply your knowledge of signals, POSIX shared memory, message passing, and pipes. To do this, you will implement three different multiprocess applications. Each application will comprise two processes that have a producer-consumer relationship. The first process, henceforth the sender, will read a user-specified file and then transfer the file contents to the second process, henceforth the receiver. The receiver will receive the file contents and save them to another file.
The difference between the three applications is the inter-process communications (IPC) mechanism they use to transfer the file contents. The first application will use the POSIX shared memory. The second application will utilize POSIX message queues. Finally, the third application will use POSIX ordinary pipes to transfer the file data from the parent to the child. All applications will implement signal handlers for the various signals to ensure a graceful termination. Furthermore, the security issues related to the IPC will be explored – including the permissions of the shared memory objects.
The sections that follow outline the differences between the programs.

Part I: Shared Memory

The sender and the receiver programs will work as follows:

The Receiver:

1. The receiver shall be started before the sender. It shall be invoked as `./recv` from the command line.
2. When launched, it shall override the default signal handler function for the `SIGUSR1` signal (the signal reserved for custom uses by user programs) with a function called `void recvFile(int sigNum)`. After doing so, the program shall go into a sleeping loop where it will sleep until a `SIGUSR1` signal arrives.
3. Once the `SIGUSR1` signal arrives, the program shall invoke the `recvFile` function. The function shall perform the following operations:
   (a) Get the shared memory ID for the shared memory segment named `cpsc351sharedmem`. If the segment does not exist, the receiver exits with an error ``Missing shared memory segment!"`. Otherwise, if the segment exists, the receiver continues to the next step.

(b) The receiver should assume that the shared memory segment in the previous step contains the data from a file that was put there by the sender. The receiver shall then get the size of the shared memory segment, read the data from the entire memory segment, and write the read data to a file called `file_recv`.

(c) Finally, the receiver shall deallocate the shared memory segment and exit.

The Sender:

1. The sender shall be started as `./sender <file name> <ID of the receiver process>`. For example, `./sender file.txt 1245`. The first argument represents the name of the file to send and the second represents the process ID of the receiver process. Please note: you can always get the process id of the receiver process using the `pidof` command. E.g., `pidof recv`. Of'course, this assumes that the receiver is already running.

2. The sender shall then allocate a shared memory segment named `cpsc351sharedmem` with permissions of `0600`. This is done to ensure that only the processes running with the UID of the current user can access the shared memory contents – important security consideration to ensure that processes from other users cannot spy on the current user through shared memory.

3. The sender shall then get the size of the file whose name was provided in the command line, set the size of the shared memory to the size of the file, and shall then read the contents of the file and write them to the shared memory segment.

4. Finally, the sender program shall send the `SIGUSR1` signal to the receiver program that the sender will assume is running. The sender can do this by invoking the `kill()` system call used for sending signals as `kill(SIGUSR1, <pid of the receiver>)`. The second argument is the process id of the receiver the user has provided at the command line (NOTE: you will need to convert the pid at the command line from string into integer. That can be done using the `atoi()` function. E.g., `int pid = atoi(argv[2])`.

The high-level skeletons for the sender and receiver can be found in the `skeletons/sharedmem`. Some of the command line argument parsing has already been done for you.

Part II: Message Queues

The following outlines the structural and functional requirements of the two programs:

The Receiver:

1. The receiver shall be invoked as `./recv`.

2. The receiver shall allocate a message queue named `cpsc351queue` and is configured to hold 10 messages with the maximum message size of 4096 bytes. The receiver shall than call the `mq_receive()` system call that shall do the following.

3. Next, the receiver shall block until the queue contains a message (this is the default behavior of `mq_receive()` as discussed in class).

4. When the message arrives, the receiver shall receive the message and check its length.

- If the length of the message is greater than 0, then the receiver shall write the message contents into file called `file__recv` and then go back calling `mq_receive()` to wait for another message.
- If the length of the message is 0, then this means the sender is done sending the message. The receiver should close the file, deallocate the shared memory segment, and exit.
- Each skeleton directory contains a Makefile that you can use to build all programs in that directory using the command `make` command.
- When there is a need to run the sender and receiver programs simultaneously, you can simply run them in different terminals at the same time.

The Sender:

1. The sender shall be invoked as `./sender <file name>` where `<file name>` is the name of the file to send to the receiver. For example, `./sender file.txt` will send the file named `file.txt`.
2. When invoked, the sender shall open the message queue named `cpsc351messagequeue`. If it does not exist, the sender shall terminate with an error.
3. Otherwise, the sender shall open the file specified at the command line and proceed as follows:
   (a) Read at most 4096 bytes from the file.
   (b) Send the bytes read through the message queue using `mq_send()` and a message priority of 1.
   (c) Repeat the previous steps until the end of file is reached.
   (d) When the end of the file is reached, send an empty message to the receiver with a priority of 2 to tell the receiver that the sending is done.
4. Terminate.

The high-level skeletons for the sender and receiver can be found in the `skeletons/mqueues`.

Part III: Message Pipes

The following outlines the structural and functional requirements of the pipes program. This program will be a little different from the above two as it will involve the transfer of file from the parent process to a child process across a pipe and therefore will require only one program.

The program shall be invoked as `./pipefile <file>` where `<file>` is the name of the file to transfer. For example, `./pipefile file.txt`.

The requirements of the parent and child are as follows:

The Parent:

1. The parent shall create an ordinary POSIX pipe using the `pipe()` system call.
2. The parent shall then fork a child process.

3. The parent shall then close the end of the pipe it does not need.
4. The parent shall then open the file specified at the command file.
5. The parent shall read the file 4096 bytes at a time, write the read bytes to the to the pipe, and repeat this step until the end of the file is reached.
6. Once the end of the file is reached, the parent shall close the end of the pipe it no longer uses and shall call `wait()` to wait for the child to terminate

The Child:

1. The child shall close the end of the pipe it does not need.
2. The child shall open a file called `file_recv` for writing.
3. The child shall read data 4096 bytes (maximum) at a and writing the bytes to the opened file until the parent closes its write end of the pipe (as discussed in class, the child will know this happened when the `read()` system call used to read data from the pipe returns 0 as the number bytes read)
4. The child shall close the file and exit.

The high-level skeletons for this part can be found in `skeletons/pipes`.

Additional Remarks

- To perform all file I/O operations, you should use the `open()/read()/write()` system calls.
- You can see some of these system calls being used throughout the skeletons.
- Additional sample file illustrating the use of `open()/read()/write()` can be found in `samples/copyfile.cpp`. The program accepts the source and destination file names from the command line and copies the source file to the destination file. For example, `./copyfile file1.txt file2.txt` will copy the contents of file `file1.txt` to `file2.txt`. The program can be compiled as `g++ copyfile.cpp -o copyfile`.
- File `samples/wp.txt` was provided to test drive the `copyfile` program. You can also use this file to test your programs to ensure that they work with large files. Your program will be tested on both large and small files.

BONUS

Rewrite the pipes program such that the parent does not use the `write()` system call to copy the data from the file to the pipe. Instead have the parent use the `sendfile()` system call. You can learn more about this system call by running the command `man sendfile` or use online research.

SUBMISSION GUIDELINES:

- ***This assignment MUST be completed using C or C++ on Linux.***

- You may work in groups of 6.
- ***Your assignment must compile and run on the Tuffix VM.***
- Please hand in your source code electronically (do not submit .o or executable code) through **CANVAS**.
- You must make sure that the code compiles and runs correctly.
- Write a README file (text file, do not submit a .doc file) which contains
  - Your name and email address.
  - The programming language you used (i.e. C or C++).
  - How to execute your program.
  - Whether you implemented the extra credit.
  - Anything special about your submission that we should take note of.
- Place all your files under one directory with a unique name (such as `p2-[userid]` for assignment 1, e.g. `p2-jarora`).
- Tar the contents of this directory using the following command. `tar cvf [directory_name].tar [directory_name]` E.g. `tar -cvf p2-jarora.tar p2-jarora/`
- Use CANVAS to upload the tared file you created above.

Grading guideline:

- Program compiles: 5'
- Correct use of message queues: 25'
- Correct use of shared memory: 25'
- Correct use of pipes: 25'
- All programs properly deallocate shared memory and message queues upon exiting 5'
- Correct signal handling: 5'
- All system calls are error-checked: 5'
- README file: 5'
- Bonus: 5'
- Late submissions shall be penalized 10%. No assignments shall be accepted after 24 hours.

Academic Honesty:

Academic Honesty: All forms of cheating shall be treated with utmost seriousness. You may discuss the problems with other students, however, you must write your **OWN codes and solutions**. Discussing solutions to the problem is **NOT** acceptable (unless specified otherwise). Copying an assignment from another student or allowing another student to copy your work **may lead to an automatic F for this course**. Moss shall be used to detect plagiarism in programming assignments. If you have any questions about whether an act of collaboration may be treated as academic dishonesty, please consult the instructor before you collaborate. Details posted at http://www.fullerton.edu/senate/documents/PDF/300/UPS300-021.pdf.