Carleton University
Department of Systems and Computer Engineering
SYSC 4001 Operating Systems Winter 2021

Assignment 3 – Concurrency, IPC, Semaphores, Shared Memory, Virtual Memory, Files

POSTED: March 16, 2021

DUE DATE: April 12, 2021 (9:00 PM) 2020 No late assignments are accepted.

THIS ASSIGNMENT MUST BE DONE BY TEAMS OF TWO STUDENTS. Only ONE student must submit the assignment, clearly stating who the members of the team are. This programming part of the Assignment should be done using the Pair Programming technique.

You can develop your application at home, but it is mandatory that the whole application runs in the course lab. Recommendation: reserve time to test the application the last days before the deadline (the lab might be crowded, and performance might be reduced for testing at the last minute).

Part I – Concepts [40 marks]

Answer the following questions. **Justify your answers. Show all your work.**

1. **[12 marks]** Consider the following page reference string in an Operating System with a Demand Paging memory management strategy:

201,302,203,404,302,201,205,206,302,201,302,203,207,206,203,302,201,302,203,206

(i) How many page faults will occur for the following page replacement algorithms when 3 frames are allocated to the program? Use
(a) LRU
(b) FIFO
(c) Optimal page replacement strategy.

(ii) Repeat (i) for 5 frames allocated to the program.

2**. [6 marks]** Consider a system with memory mapping done on a page basis and using a single-level page table. Assume that the necessary page table is always in memory and a memory reference takes 250ns.
a. How long does a paged memory reference take? **Explain** your answer.
b. If we add TLB that imposes an overhead of 30ns on a hit or a miss. If we assume that 80% of all memory references hit in the TLB, what is the effective memory access time? **Explain** your answer.
c. Why adding another layer, TLB, can improve performance? Are there situations where the performance may be worse with TLB than without TLB? **Explain.**

3**. [6 marks]** Consider a paged logical address space (composed of 32 pages of 2 Kbytes each) mapped into a 1-Mbyte physical memory space.
a. What is the format of the processor's logical address?
b. What is the length and width of the page table (disregarding all the control bits)?
c. What is the effect on the page table if the physical memory space is reduced by half? Assume that the number of page entries and page size stay the same.

4. **[4 marks]**
a. Explain what a Race condition is. Discuss a concrete example of a race condition.
b. Why disabling interrupts is not a good mechanism to prevent race conditions? Explain and justify.

5. **[2 marks]** Briefly explain how Semaphores work.

6. **[4 marks]** Explain, in detail, what an *open* operation must do. Consider the case of a file opened for APPEND, in a file system using a hierarchical directory structure.

7.
a) **[4 marks]** (from Silberschatz) Consider a file system that uses inodes to represent files. Disk blocks are 8Kb in size, and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, as well as single, double, and triple indirect disk blocks. What is the maximum size of a file that can be stored in this file system?

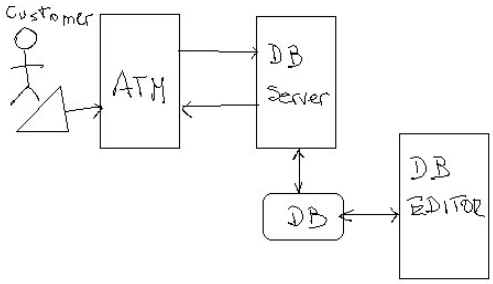
b) **[2 marks]** Explain what you can do in case (a) if you need to store a file that is larger than the maximum size computed. Give an example showing how you can define a larger file, and what the size of that file would be.

Part II – Programming [40 marks]

Problem description:

We will build a simple prototype of a program that mimics the work of an ATM system using concurrent processes and IPC services.

The system is organized in three concurrent processes, as shown in the following figure:



The Data Base (DB) is a text file that contains information about the customers using the following format:

Account No.	Encoded PIN	Funds available
5 char	3 char	Float

The initial database for the system contains the following information:

00001,107,3443.22
00011,323,10089.97
00117,259,112.00

The ATM component does the following:

1. Requests an account number from the customer
2. Requests a PIN from the customer
3. Transmits these two values to the DB Server ("PIN" message). If the DB server returns "PIN_WRONG", restart from 1.
4. After the 3rd trial, "account is blocked" is displayed on screen.
5. If the DB server returns "OK", the customer chooses a banking operation:
    a. Show balance: send a "BALANCE" message, check for the balance on the DB, and displays the balance

b. Withdrawal: send a "WITHDRAW" message, and the amount to withdraw. If there are not enough funds available, the DB server returns the message "NSF" (not sufficient funds). In that case, display that not enough funds are available. Otherwise, the DB Server responds with and "FUNDS_OK" message, and the new balance is displayed on screen.

The ATM component waits for the next customer (one transaction per customer; if you type "X" as customer number, the whole system ends).

The DB server does the following:

1. Waits for messages forever
2. When a PIN message is received, it gets the account number and the PIN. It searches the account number in the DB. It then subtracts 1 from the PIN number and compares the result with the number stored in the DB (this is how the PIN number is "encrypted" in the DB). If the account number is found and the PIN is correct, return an "OK' message, and saves the information on the account in a local variable. If there is a failure on the account or PIN numbers, it returns a "PIN_WRONG" message. After the 3rd attempt, the account is blocked. To do so, the first digit in the account number is converted to an "X" character.
3. If the message is "BALANCE", search the current account number, get the Funds field from the DB, and return it.
4. If the message is "WITHDRAW", get the amount requested, and check the funds available. If there are enough funds in the account, return "FUNDS_OK", decrement the funds available, and update the file. If there is not enough money, return "NSF"
5. If an "UPDATE_DB" message is received, the updated information is obtained and saved to the file (adding 1 to the PIN number before saving it, to "encrypt" the PIN).

The DB Editor will:

1. Loop forever
2. Request an account number
3. Request a PIN.
4. Request a value representing the funds available
5. Send this information with an "UPDATE_DB" message to the DB server.

We will build the application in incremental steps. The first part is not mandatory. If you can develop Part B from scratch, Part A should not be submitted, as what you will do in Part A will be also included in Part B. But Part A is available for those who prefer building the application in incremental prototypes. If you do not complete the whole assignment, the best prototype will give you marks according to the marking scheme provided.

Part A) [20 marks – OPTIONAL. You can submit Part B only and obtain the 40 marks. Part B includes Part A. Submit Part A) only if you feel you cannot work with Part B directly, or you want to obtain partial marks].

SUBMIT ONLY THE MOST ADVANCED WORKING PROGRAM and Part 6 (which is independent from the other 5 parts)

1. Run two concurrent processes in C/C++ under Linux

Using the *fork* system call (page 472 of the Linux book), create two independent processes. Process 1 will run forever and will display, on screen "I am Process 1". Process 2 will display, on screen "I am Process 2".

To finish the program, use the *kill* command (man pages), find the pid of both processes (*ps*) and kill them.

2. Extend the Processes. Now it will display the message on screen and will wait for an input: when you press "1", it will display the message again. Process 2 will wait for an input and will display the message

again when you press “2”. Use the *exec* system call to launch Process 2 (i.e., Process 2 should be a different program/executable).

3. Extend the processes above once more. Use the *wait* system call. Process 1 starts, and after *fork,* it displays a message, and waits for Process 2. Process 2 displays a message and waits for an input. If the input is “2”, then it displays the message again. If the input is “x”, it finishes, and exits. When this happens, Process 1 should end too.

4. Extend the processes above once more. They should now share memory. The primary functions are listed in the book and course materials. Using *shmget, shmctl, shmat,* and *shmdt*, add a common variable shared between the two processed. The variable contains the character typed: 1, 2 or x. Each of the processes should now react to the value of the shared variable, and display a message identifying themselves or exit.

5. Extend the processes above once more. They should now protect concurrent access to the shared memory position. On top of the *shm* instructions, you should protect the shared memory access using semaphores. Use *semget, semop, semctl* to protect the shared memory section.

As before, you now have a common variable shared between the two processes, and it is protected from concurrent access using semaphores. The shared variable contains the character typed: 1, 2 or x. Each of the processes should now react to the value of the shared variable, and display a message identifying themselves or exit.

6. Message queues practice: write a program with two children processes. Process 1 (the parent process) will display a number (start with a number 1, and increment it on every step), and then will send it to Process 2. Process 2 (first child process) will multiply this number by 2 and will display the total. The new number will be sent to Process 3 (second child process). Process 3 will add 1 to the number and display the result. Use *msgget(), msgsnd(), msgrcv()* and *msgctl()* to transfer messages between the processes.

Information about message passing is on Chapter 14 of the Linux reference book, and in the online materials posted.

Marks to be obtained in Part A:

If you deliver UP TO exercise number:	**You will obtain a mark of:**
1	3
2	6
3	19
4	12
5	15
5 and 6	20

Part B) [20 marks – if you only submit part B: 40 marks]

1. [10 marks **– if you only submit part B: 30 marks**]

Using all the code you wrote in Part A, write the solution to the problem. Here work incrementally too: first build a concurrent solution for the DB server and the DB Editor.

Your solution must have 2 processes communicating using IPC. Other Linux system calls may be used for process creation, termination, creation, and deletion of shared memory etc.

Test: use the application to create the initial DB discussed earlier using the DB editor.

2. [10 marks]

Add the ATM component executing concurrently. Your solution must have now three processes synchronized using IPC services. You must guarantee mutual exclusion when accessing the database. To test the application, open two windows (one for the ATM, and the second for the DB Editor) and check that the DB is properly updated while the ATM interface works properly. DO NOT WORRY ABOUT DEADLOCKS.

Marking Scheme:

Start by defining your system's architecture, showing the messages that you are going to use, the processes, how they are organized, and any other consideration (semaphores, shared memory, shared variables, etc.). This information will be useful to organize your work and will be evaluated (together with comments in your source code and any other form of documentation provided).

We will evaluate

- Correctness (including error checking): 70%
- Documentation and input/output: 20%
- Program structure: 5%
- Style and readability: 5%

Programs not compiling or those with rude comments/text will receive a mark of 0 (zero).

Keep in mind that you ALWAYS must double check your manual pages, in case that there are slightly different values in your installed software.