Operating Systems [2019-2020]

Assignment 04 – Shared memory and semaphores

Introduction

One of the quickest ways for two processes to communicate is by using shared memory. When two processes are using the same shared memory segment, the communication takes place directly writing and reading of the memory, without any need for calling the operating system routines. Among others, Linux support System V shared memory routines, which will be used in this assignment.

However, when more than one process shares the same memory segment, it is necessary to ensure that when one is writing in the memory, no other process will attempt to read or write to that same memory. It is therefore necessary to synchronize these processes memory accesses. One of the mechanisms that can be used are the IPC semaphores.

Objectives

Students concluding this work successfully should be able to:

- Create and use a segment of shared memory to share date between concurrent processes.
- Use semaphores to synchronize the access to shared resources from competing processes

Support Material

- K. A. Robbins, S. Robbins, “Unix Systems Programming: Communication, Concurrency, and Threads”, Prentice Hall:
  - Chapter 14 – Critical Sections and Semaphores
  - Chapter 15 – POSIX IPC
- “Programming in C and Unix”:
  - IPC: Semaphores
  - IPC: Shared Memory

Exercises

Previous Note:

- A file named *kill_ipcs.sh* is given with this assignment. It removes (only) SysV IPC structures created by you from memory. Use it to make sure you do not leave any IPC structures after running your programs.
- Use *ipcs* (in the Linux command line) to get info on SysV IPC structures.
- In these exercises both POSIX semaphores and SysV semaphore can be used. To ease the implementation of SysV semaphores the *semlib* library and its code are given with this assignment. Analyse the provided library and use it at will. For the ease of use, POSIX semaphores are advised.

1. Variable in shared memory

Write a program that creates N worker processes and a variable in shared memory (initially set as 1000). The number N of processes is given by command line. After being created every worker writes to the screen his PID. At a random time, ranging from 1 to 2 seconds, each of the worker processes will try to access the variable in shared memory and increment its value by one. After that, the process will die. In the end, after all worker processes have incremented the shared variable value, the main process writes the variable value, cleans all resources and exits.

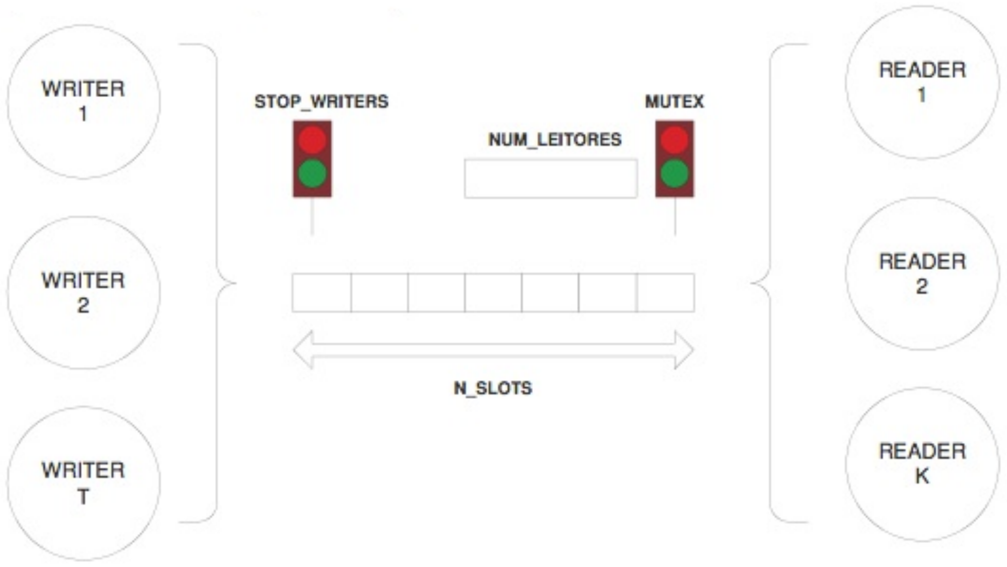
Complete the code supplied (file *sharedvariable.c*).

2. Stock market

In this exercise a set of brokers (writers) update the value of different stocks kept in shared memory. At the same time clients check the value of their stock (readers). Each broker and client is created as a different process. Since reading does not modify any data, reads can be performed simultaneously, provided that no writing is taking place. However, when writing, no other process can access the shared data, even for reading. Writing must be done in mutual exclusion.

Read the code (file *stockmarket.c*) supplied with this assignment and complete the missing sections according to the following rules and general architecture:

- The memory with the different stocks is implemented by the structure *mem_structure*. This structure will be placed in shared memory.
- When a writer changes the value of a stock (can be randomly generated), no other process can access the mem_structure. When there is a process reading, no writer can access, but other readers may read simultaneously. This behavior is achieved by using two semaphores, one to control writers' access to the shared memory (STOP_WRITERS) and another that controls readers access (MUTEX). In this version the readers should have priority over writers.



```
while (true) {
  sem_wait(STOP_WRITERS);

  write_data();

  sem_post(STOP_WRITERS);
}
```

```
while (true) {
  sem_wait(MUTEX);
  ++num_leitores;
  if (num_leitores == 1)
    sem_wait(STOP_WRITERS);
  sem_post(MUTEX);

  read_data();

  sem_wait(MUTEX);
  --num_leitores;
  if (num_leitores == 0)
    sem_post(STOP_WRITERS);
  sem_post(MUTEX);
}
```

Sections to complete:

- In the *main* function, add the necessary code to create the shared memory segment and semaphores.
- In the *reader_code* function, add the required code for the readers (clients). Use the function *read_stock()* to read a stock and *get_stock()* to select a random stock to read.
- In the *writer_code* function, add the required code for the writers (brokers). Use the available function *write_stock()* .

Notes:

- This program uses Ctrl+C to terminate. All the code concerning signals is given.

3. Lottery

Implement a system for automatic generation of lottery keys. The system has 1 producer and N consumers. A producer process should generate random numbers between 1 and 49. The values generated by the producer should be stored in a shared memory structure that the consumers will also access. Each consumer process will read numbers from the shared memory, one by one, until it gets a set of 6 distinct numbers. Each read number is added to a file opened by each consumer. The file names of the keys must have the format "key_x", **x** being the number of the consumer. When the key is complete, the file should be closed, the key should be printed in the screen and the consumer process terminated. When all consumers finish, the producer should also exit. Use semaphores to manage the access to the shared memory.

Your program should be prepared to finish without leaving resources allocated in the system and have no orphan or zombie processes. Also, deadlock conditions should be avoided.

4. Supermarket

Implement a program to simulate the weighing of fruit and vegetables in a supermarket. Customers need to weight their bags of fruit and vegetables using one of the scales available. The scales are controlled by an employee labels the price according to the weight. Customers need to wait for the labelling before leaving.

In our program, clients and supermarket employee will be represented by *producers* and *consumer* processes, and scales as an array in shared memory. Semaphores will be used to ensure that a scale is not used simultaneously by two clients, that customers do not remove the bags before they are weighed, that the employee only weighs a bag at a time and that the balance is not used without a bag.

Your program should be prepared to conclude without leaving resources occupied in the system and should not create orphan or zombie processes. Also, deadlock conditions should be avoided.