Synchronisation with semaphores

Orlando Sousa, António Barros, Luis Lino Ferreira, André Andrade,
Carlos Gonçalves, Jorge Pinto Leite, Nuno Morgado,
David Freitas e Armando Nobre

May 2, 2023

Instructions

All exercises must use semaphores for the synchronization between processes. The use of active waiting is discouraged. The evaluation of this part will mostly focus on the exercises on part 2.

For every exercise you must present three items:

1. The patterns are being used in a specific exercise. Consult the slides from the T or TP classes about synchronization.
2. The pseudo-code of the algorithm used on the resolution.
3. The implementation of the algorithm in C, *adequately formatted and commented*.

1 Exercises Part 1

1. Implement a program that creates eight child processes. Each child reads 200 integer numbers from the file **Numbers.txt** and writes those numbers along with its PID to the file **Output.txt**. Assuming a child process with PID 16734 is writing number 12, the respective output should be:

   ```
   [16734] 12
   ```

   Before generating the children, the father must delete any previous version of the file **Output.txt**. Check the manual entry for the `remove()` function for this purpose.

   *Suggestion: Use the Linux `seq` command to easily generate a sequence of numbers and store it in the "**Numbers.txt**" file. The command `seq 1 200 > Numbers.txt` should help.*

   (a) Develop a first version of a program where every child reads one number from **Numbers.txt** and writes it to **Output.txt** without interference from the concurrent siblings:

   ```
   #Start critical section CS
       Read value from Numbers.txt
       Write value to Output.txt
   #End critical section CS
   ```

   Run the program a few times and use the command `more Output.txt` to check the result for each run.

   (b) Based on the first version, develop a second version where a process reading from **Numbers.txt** does not prevent other process writing to **Output.txt**. Nevertheless, only one process can access each file at any time instant:

   ```
   #Start critical section CS1
       Read value from Numbers.txt
   #End critical section CS1
   #Start critical section CS2
       Write value to Output.txt
   #End critical section CS2
   ```

After all the children have completed, the father prints the content of **Output.txt**.

(c) Write a third version so that every child reads all numbers from **Numbers.txt** by order of their generation, i.e. from the first to the eighth process.
Hint: Use eight semaphores to synchronise between the child processes.

2. Develop an application that employs several concurrent processes to fill a shared memory area. The shared memory area has a capacity for 50 strings, each string supporting 80 characters.

   (a) Implement a first program, that writes text to the shared memory area. The program should behave as follows:
      i. Get exclusive access to the shared memory area.
      ii. In the first free string, add the following text line: `I'm the Father - with PID X`, where X is the PID of the process.
      iii. Sleep for a random time ranging from 1 to 5 seconds.
      iv. Unlock the shared memory area.
      v. Repeat the whole sequence.

   The program terminates when the shared memory limit of 50 strings is reached.

   (b) Implement a second program, that prints the current lines written in the shared memory area, and the respective total number of lines.

   Execute several concurrent instances of your code to test the solution.

3. Change the first program of the previous question by adding the following features:

   - Add an option to remove (i.e. to assume as *free*) the last written string from the shared memory. In each round, the probability of this option is 30%; consequently, the probability of writing a new line reduces to 70%.
   - Your program should wait at most 12 seconds to access the shared memory. The user must be warned every time the deadline is missed.
     Suggestion: Check the `sem_timedwait()` for this purpose.
     Attention: The macOS library does not support the `sem_timedwait()` function. If you are developing for this system, take notice that the `sem_wait()` unblocks when a signal (such as `SIGALRM`) is received, returning -1 and setting the `errno` variable with value `EINTR`.

   Once more, execute several concurrent instances of your code to test the solution.

   .

4. Implement a program that creates two new child processes. Each process is responsible for writing a message on the screen; however, the messages must follow a strict sequence. Use semaphores to synchronise their actions.

   (a) Write the program to adhere to this sequence:
      i. The first child process writes on the screen "1st child";
      ii. The father process writes on the screen "Father";
      iii. The second child process writes on screen "2nd child";

   (b) Change the previous program to execute 15 times the defined sequence.

5. Consider a program that spawns three children processes. Each child process must behave as follows:

Child process 1	**Child process 2**	**Child process 3**
printf("Sistemas ");	printf("de ");	printf("Computadores -");
printf("a ");	printf("melhor ");	printf("disciplina! ");

   Use the minimum number of semaphores required, so the result printed in the screen is:

```
Sistemas de Computadores - a melhor disciplina!
```

   Note: Use `fflush(stdout)` after every `printf()` so the output is not buffered.

6. Write a program that creates a new process. The father and the child should indefinitely write the letters 'S' and 'C', respectively:

Father process	**Child process**

while TRUE do:
  print "S"
	while TRUE do:
	  print "C"

   Implement a policy based on semaphores such that at any moment the number of 'S' or 'C' differs by at most two. The solution should output strings such as: "SCCSSSCCSCSCCC".

   Note: Use `fflush(stdout)` after every `printf()` so the text is not buffered and printed immediately.

7. Implement a program that creates a new process, such that the two processes behave as follows.

Process 1	**Process 2**

sleep(some random time);
buy_chips();
eat_and_drink();
	sleep(some random time);
	buy_beer();
	eat_and_drink();

   (a) P1 and P2 can only execute `eat_and_drink()`, after both have acquired the chips and the beer. Use semaphores to synchronize their actions and the most adequate pattern.

   (b) Modify the program to add 4 more processes which will also execute (randomly) `buy_chips()` or `buy_beer()`. The 6 processes can only execute `eat_and_drink()` when all other processes finished acquiring the chips and the beer. Be careful in order to produce and adequate output that shows clearly the execution of the processes.

2 Exercises Part 2

8. A company wants to develop an application to manage personal data records (*Number*, *Name* and *Address*) of clients. The company foresees that it will never reach more than 100 records, so this should be the maximum number of records supported by the application. The personal records must be stored in a shared memory region, so they are accessible to and shared by concurrent processes/programs.

   The application should be composed by a set of **three independent programs**, each one specialised on a specific task.

   - **Insert** : This program allows to add a client's personal record in the shared memory area.
   - **Consult** : This program allows you to consult a client's personal data, based on a given identification number.
   - **Consult_All** : Lists all data in the shared memory area. Only the records that contain valid data should be displayed. Multiple consult programs might be executed in parallel.

The application must support multiple concurrent *consults* and *inserts*. Simulate the time required by these operations. Use semaphores to synchronize their actions. Also include in your answer a description of the potential race conditions in this problem and how they were solved. Develop tests to verify that those potential race conditions never occur.

9. In order to determine the performance of a parallel program on a multi core system you were asked to:

   (a) develop a program which allows you to simulate the splitting of work between several processes. For that purpose each process should actively run between 100 us and 1S, according to the initial specification of the user.

   (b) your program should be evoked as follows: `testpar n time`, where argument `n` is the number of child and argument `time` is the execution time for each child

   (c) include a mechanism to measure with us precision de execution time of your software. From the line just before the cycle where you create all child until the father receives all joins from its child.

   (d) Test your program, at least 10 times for each configuration and register the values. The following configuration should be used:

   1/100 us, 1/500 us, 1/1ms, 1/10ms, 1/50ms, 1/500ms, 1/1s

   2/100 us, 2/500 us, 2/1ms, 2/10ms, 2/50ms, 2/500ms, 2/1s

   4...

   8...

   16/100 us, 1/500 us, 1/1ms, 1/10ms, 1/50ms, 1/500ms, 1/1s

10. Assume the Porto metro, where each train has a capacity for 20 passengers, and has three doors. Use semaphores to develop a solution that ensures:

    - the train will never be overloaded;
    - the passengers will not collide with each other when getting in or getting out off the train.
    - To simulate the problem assum a train, loaded with 15 passenger arrives at station A, where 5 will leave the train and 20 will try to enter the train, when full it goes for station B (needs 5 seconds in this trip). At station B, 10 will leave the train and 5 will enter.
    - Consider that each passenger is a process, which prints when it is waiting for a train, when it is riding the train, and when it leaves the train.
    - Each door allows one passenger to leave or to enter, at a time. This operation requires 200 ms. Assume that each passenger chooses a random door to enter and he leaves through the same door.
    - The output of your code should show the operation of the train.

11. Implement two programs, to simulate the selling of tickets to customers. You should simulate the customers (several processes), using semaphores to synchronize the access to a shared memory area for data exchange, which contains the number of the next ticket. Consider also the following:

    - Only the seller has access to the tickets, only he knows the number of the next ticket.
    - The clients should be served by their arrival order, blocking until getting their ticket.
    - The client should print the number of the ticket returned by the seller through shared memory. Only one ticket can be bought by client.
    - The behavior of *client* and *seller* should be implemented in separate programs.
    - Assume that each client needs a random number of seconds (1-10) to be served.
    - You should execute several clients concurrently, demonstrating the operation of the algorithm implemented.

12. Implement a program that creates two new processes. The child processes will have the role of producer, while the father will act as a consumer.

    In a shared memory area, there is a *circular buffer* with a capacity for 10 integers.

The producer writes 30 increasing values into the buffer, with a interval of 1s and prints them on the screen.

The consumer read values from the buffer every 2 seconds and prints them on screen.

Write a program that performs as described, following this set of functional restrictions:

(a) a producer blocks if the buffer is full;

(b) a producer only writes after guaranteeing mutual exclusion;

(c) a consumer blocks while there is no new data available.

13. Implement the following simulation, where:

    (a) **Reader processes**, read a string from a shared memory area: The readers do not change the shared memory area, therefore several readers can access the shared memory area, in parallel. Each reader should sleep for 1 second, print the string read from shared memory and the number of readers at that time instant. Readers can only access shared memory when there are no writers.

    (b) **Writer processes**, writes on the shared memory area: Writes its PID and current time. Only one writer can access the shared memory area at the same time. Each writer prints the number of writers and also the number of readers at that time instant.

    (c) Develop a new version of your code where Writers have priority over readers.

    Execute several readers and writers at the same time to test your software.

14. Consider a shared memory area (e.g. containing 10 integers) which allows only a one-way communication between a set of processes Ax and a set of processes Bx. Only one process, at a time, from Ax or Bx, is allowed to place its info on the shared memory. After one second the process should signal that other processes, from the same set, can place content on the shared memory. If there are no other processes ready to access the shared memory, then a process from the other set of processes can gain access to the shared memory. Demonstrate the operation of such a program by simulating a set of 3 Ax processes and a set of 2 Bx processes, each of them producing data (10 integers) every 5 seconds, and processes from Bx producing data every 6 seconds. Make sure your printing on screen clearly show the execution of the different steps in this program.

15. Consider a social club with a maximum capacity of 10 (just to simplify the simulation) clients.

    (a) While the full capacity is not reached, clients enter the club according to their arrival order. Whenever the club is full, the entrance of clients is conditioned to the exit of other clients. Simulate the entrance and leaving of clients.

    (b) consider now that there are three types of clients: **VIP**, **Special** and **Normal**. In this case, priority should be given firstly to waiting VIP, then Special, and only then to Normal clients, by this order. Print the necessary messages to clearly show the status of the club. The clients should be modeled as processes each of them with its type and the time during which it stays at the club.

16. From an initial vector of 10000 integers randomly chosen in the range 1-10000, it is intended to fill a final vector of 1000 positions with the moving average of 10 values and determine the largest value found (on the final vector).

    To resolve the problem, design a C program with the following requirements:

    (a) Five child processes (P1 to P5) must be created that fill the final vector according to the conditions stated above.

    (b) Each child process must process 1/5th of the vector.

    (c) Create another child process (Pmax) that, running in parallel with the two processes mentioned above, should print a warning message whenever a new larger value is found in the final vector. This process should not use active waiting.

    (d) The final vector and value of the largest value should be printed by the parent process after being sure that P1-5 have finished processing, correctly.

Note: In your solution you should take into account the correct use of processes, shared memory and semaphores, ensuring that the division of work between the processes is able to take advantage of all existing processors on the machine.

17. Unit X has developed a new product, which is intended to be monitored by a set of 5 sensors. Each of the sensors reads 6 values and ends. In each of the measurements, and depending on the value obtained, the sensor can be placed in an alarm state. You were asked to develop a solution that creates 6 processes: 1 "Controller" process and 5 "Sensor" processes.

    The "Controller" process shall be responsible for:

    (a) Create and initialize a shared memory region where the measurement results of each sensor will be saved;

    (b) Create the semaphores needed to implement the solution;

    (c) Print on the screen the result of the execution of each sensor, whenever values are received from the sensors;

    (d) Print on the screen the result of changing the number of sensors in alarm, whenever this occurs;

    (e) Finish at the end of receiving 6 measurements of each sensor, as well as remove the shared memory region and the created semaphores.

    Each "Sensor" process must:

    (a) Generate a random value between 0 and 100, thus simulating the result of a measurement;

    (b) Save this value in your respective area of the shared memory region;

    (c) Pass a sensor to "alarm" state, thus increasing the number of sensors in this state, if the measurement result exceeds 50 (consider that you should not trigger the same sensor in alarm, in duplicate);

    (d) Decrement the number of alarm sensors if the result of the current measurement and the previous measurement is less than 50;

    (e) Wait 1 second and repeat the operation again (a total of 6 times);

    The 5 "Sensor" processes must be performed simultaneously, ensuring correct access to the shared memory region through the use of traffic lights and without using active standby. You should also define the shared data structure that you consider most appropriate for troubleshooting.