Universidade de Coimbra
Faculdade de Ciências e Tecnologia
Departamento de Engenharia Electrotécnica e de Computadores

Sistemas Operativos - 2015/2016 – Practical Assignment Nº2
Inter-Process Communication: Semaphores, Shared Memory, Message Queues; Threads

1 Objectives

The objective of this practical assignment is to learn to work with inter-process communication mechanisms (IPC), and threads in Unix. The following IPC mechanisms are addressed: semaphores, shared memory, and message queues. Presentation of the classical "readers/writers" paradigm, using shared memory.

2 Material for Preparation and Reference

- [Robbins e Robbins, 2003,
  Chapter 4, "UNIX I/O";
  Chapter 5, "Files and Directories";
  Chapter 6, "UNIX Special Files";
  Chapter 8, "Signals";
  Chapter 12, "POSIX Threads";
  Chapter 13, "Thread Synchronization";
  Chapter 14, "Critical Sections and Semaphores";
  Chapter 15, "POSIX IPC";
  Section 8.6, "Handling Signals: Errors and Async-signal Safety";
  Appendix B, "Restart Library"].

- [Araújo, 2014,
  "UNIX Programming: Interprocess Communication";
  "UNIX Programming: Pthreads"
  "UNIX Programming: Memory Mapped Files";].

- [Threads - Referência Rápida, 2005,
  "Threads - Referência Rápida"].

- [Kernighan e Ritchie, 1988,
  Chapter 7, "Input and Output" (pp. 151-168);
  Chapter 8, "The Unix System Interface" (pp. 169-189);
  Appendix B, "Standard Library" (pp. 241-258)].

3 Introduction

One of the concurrent programming models that is currently often used in computer systems is called "Readers/Writers". In this model there are various processes (or threads) that are concurrently reading shared information and, from time to time, there are one or more processes (or threads) that update the information. Since readings do not change the data, several readings can be simultaneously performed, provided that no writing is being done. However, when a writing takes place, this should be performed in mutual exclusion with respect to the other writers, and also no readings can be allowed during that period of time. Figure 1 illustrates this principle.

Any current transactional system, and all database system, support a more or less elaborate version of this model. For example, in an air line ticket reservation system, it must be possible for all the

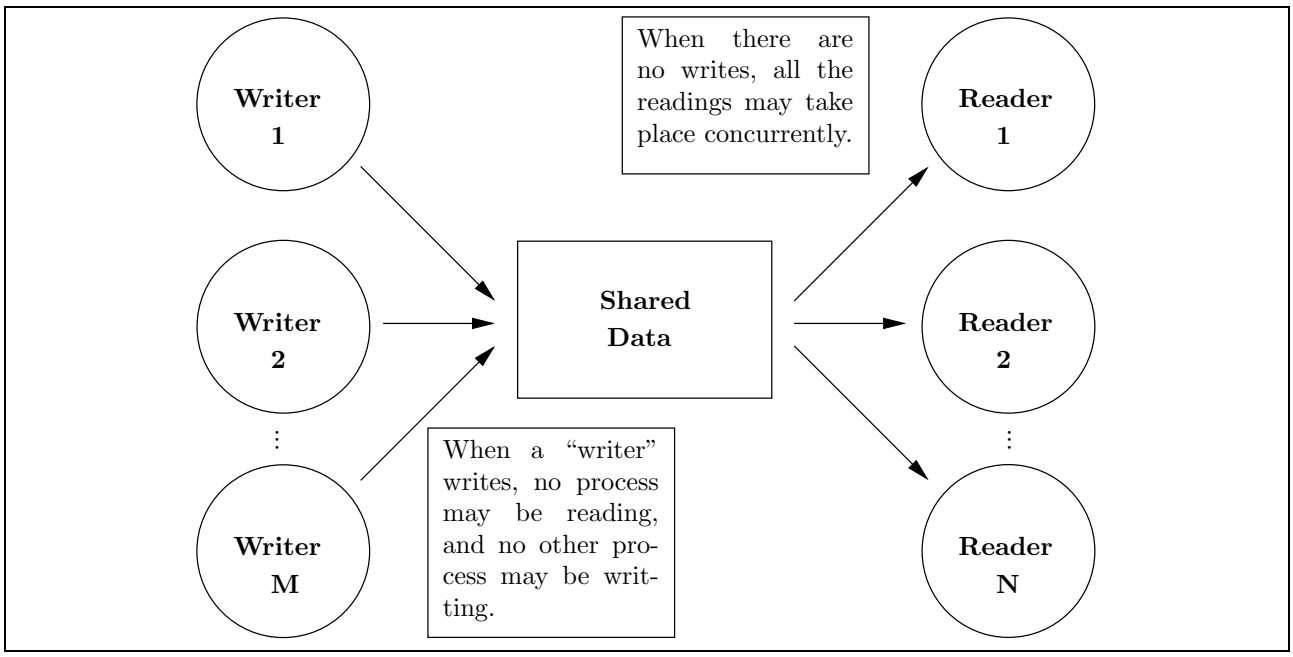

Figure 1: The readers/writers paradigm.

millions of terminals existing worldwide to read simultaneously what are the free seats on a given flight. However, when performing an update of one of the seats of the flight, the update must be atomically performed.

This practical assignment will deal with a simplified version of the "Readers/Writers" model, in the "priority to readers" variant.

An important point of this algorithm is that it is possible to have various simultaneous readings, without any reading stopping the others. This permits an increase in system performance. Whenever an update is performed, it is guaranteed that there are no other processes reading or writing.

4 The Readers/Writers Problem in This Practical Assignment

To start preparing your work, download the file `tpso1602.tgz` from the course web page. Extract the file to a local directory and start by analysing the code of the file `tpso1602.c`.

The program in file `tpso1602.c` intends to implement a simulator of queues in a *callcentre*. The program should create one "process of the writers" (POTW) and one "process of the readers" (POTR). Inside the POTW, there will be various writer threads that represent telephone centrals that are responsible for forwarding calls to the queue. Inside the POTR, there will be several reader threads that represent the call centre operators that answer the calls. The writer threads place calls in the queue and the reader threads remove entries from the queue. There is also a monitor process that, simultaneously, keeps reporting the number of calls in the queue. The monitor process is implemented by the initial process of the program (in the `monitor()` function). Therefore, the program will be constituted by a total of 3 processes.

The queue is implemented by a structure of type `mem_structure`. This structure is placed in a shared memory area and the following data items can be found inside it: `mutex` and `stop_writers` are semaphores used to implement the readers/writers paradigm; `head` is the current position where to insert new entries in the queue; `tail` is the current position where to remove entries from the queue (when `head = tail` the queue is empty); `readers` is the number of threads reading from the shared memory at the moment; and `slots` is an array of integers that represents the queue itself.

When a writer is placing a new entry into the queue, no other writer or reader thread can access the queue. When there is a thread reading, no thread can write but an unrestricted number of readings can be simultaneously performed. This behaviour is obtained using two semaphores: one controls the write

WRITER	*READER*
...	...
while(TRUE){	while(TRUE){
wait(stop_writers);	wait(mutex);
do_write(&queue);	local_tail = queue->tail;
signal(stop_writers);	(queue->readers)++;
...	if(queue->tail != queue->head)
}	queue->tail = next(queue->tail);
	if(queue->readers == 1)
	wait(stop_writers);
	signal(mutex);
	do_read(&queue);
	wait(mutex);
	(queue->readers)--;
	if(queue->readers == 0)
	signal(stop_writers);
	signal(mutex);
	...
	}

Figure 2: Readers/writers pseudo-code.

access to the queue (`stop_writers`); and the other (`mutex`) controls the access to the `readers` and `tail` variables. The application of these semaphores to solve this readers/writers problem is illustrated in the pseudo-code of Figure 2.

When a writer wants to insert a new entry it first reserves the semaphore `stop_writers`, then writes, and finally frees the semaphore. On the other hand, when a thread wants to read it reserves the `mutex` semaphore; saves the current value of `tail` on a local variable; increments the number of readers on the `readers` variable; increments the value of `tail` such that the next thread to read does not obtain the same call; and verifies if it is the first thread to read, and if yes, it reserves the `stop_writers` semaphore guaranteeing that there will be no writings while there are being readings. Before reading the value pointed on the queue by the local variable that took the value of `tail`, the thread frees the `mutex`. After reading, the reader thread reserves again the `mutex`, decreases the number of threads reading on the `readers` variable, verifies if there are no more threads reading, and if yes, then it frees the `stop_writers` semaphore.

5 Assigned Work

Problem 1. Readers/Writers using shared memory and semaphores. Complete `tpso1602.c`.

1.1 In file `tpso1602.c`, at the beginning of the `main` function, start by writing the code necessary to reserve a shared memory space to hold a structure of type `mem_structure` (defined at the beginning of file `tpso1602.c`). Use POSIX:MAP shared memory (`shm_open()`, etc). Note that the value of the file descriptor of this shared memory object should be saved in the local variable `shmfd`.

1.2 Write the code that will associate the memory area to the process address space, specifically to the variable `queue` defined at the beginning of the `main()` function.

1.3 Create the 2 unnamed POSIX:SEM semaphores (`sem_init()`, etc): the `mutex`, and the `stop_writers`. Adequately choose the initial values of the semaphores.

1.4 In the indicated places in the `main()` function do the following. Create the POTW, and the writer threads within the POTW. Create the POTR, and the reader threads within the POTR. The numbers of writer and reader threads are `NUM_WRITERS` and `NUM_READERS`, respectively (these numbers are '`#define`'d in `tpso1602.c`, and can be adjusted). Each writer thread shall receive the pointer to the shared memory area, and its writer-number (between `1` and `NUM_WRITERS`) through the input arguments to the corresponding thread on its creation. Similarly, each reader thread shall receive the pointer to the shared memory area, and its reader-number (between `1` and `NUM_READERS`) through the input arguments to the corresponding thread on its creation. This is the only method permitted for the each writer or reader thread to know these 2 values to be used inside the threads.

1.5 In the indicated zones in the source code, insert the implementations of the `reader()`, and `writer()` threads in order to implement the behaviour described in Figure 2. Use the adequate functions to manipulate/control the POSIX:SEM unnamed semaphores, and use the functions "`void do_read(int pos, int n_reader, mem_structure *queue)`", and "`void do_write(int n_writer, mem_structure *queue)`" to read and write on the queue.

Note: in the implementation of the work, make all the threads to be not-synchronised in terms of execution. Specifically, each thread, at the end of each iteration of the `while(TRUE)` cycle, should sleep during a period of time of random duration between 0 and 10 seconds ("`sleep(random()%11);`"). The random number generator should be separately initialised in each of the POTW and POTR processes ("`srandom(getpid());`").

1.6 The monitor process should access the queue in exclusive access mode. For this purpose, in the indicated places within the `monitor()` function, insert the code to perform the wait and signal operations on the `stop_writers` semaphore.

1.7 In the indicated place in the "`void cleanup(int signo)`" function associated to the `SIGINT` signal in the monitor process, do the following: (*i*) insert code to terminate the POTW and POTR, and to wait for their termination, and (*ii*) insert code to unmap and remove the shared memory area and to destroy the semaphores. Explain the possible importance that operation "(*ii*)" may have in these types of programs.

Compile and execute the program using `make`. Analyse the output of the program in order to validate the results. Use the `pause()` function in the critical zones of the program in order to validate the synchronisation mechanisms. Use `Ctrl-C` to terminate the program.

Problem 2. Message Queues. Write 2 processes, a `sender` and a `receiver`, where each process is a separate program. The `sender` should send an input file to the `receiver` using POSIX:XSI message queues (`msgget()`, etc). The `receiver` should save the received file in an output file. The maximum number of bytes in the data field of each message transmitted by the `sender` or by the `receiver` is given by a macro `MAX_BYTES`, which should be defined using a "`#define`" directive (for example: "`#define MAX_BYTES 10`"). The program should be compliant to the fact that the users may freely change the value of "`MAX_BYTES`" to any positive integer before compiling the programs. The sender should separately read from the input file the data bytes to be sent in each individual message. The receiver should separately write to the output file the data bytes received in each individual message. The input and output file names should be specified as the first command line argument to the sender process and to the receiver process, respectively. It is assumed that the names of the input and output files are different. After running the processes, the input and output files should have the same contents. An example of how the two processes can be run is as follows:

```
shell_1_prompt$ ./receiver file_out.txt
Receiver: file reception and saving is now complete.
      ...
            shell_2_prompt$ ./sender file_in.txt
            Sender: file reading and sending is now complete.
```

Report and Material to Deliver

A succinct report should be delivered to the professor in PDF format. The report should contain the following information in the first page: name of the course ("disciplina"), title and number of the practical assignment, name of the students, number of the class ("turma"), number of the group. It should be submitted through the Nónio system (`https://inforestudante.uc.pt/`) area of the "Sistemas Operativos" course in a single file in `zip` format that should contain all the relevant files that have been involved in the realisation of the practical assignment. The name of the submitted `zip` file should be "`tXgYrZ-so16.zip`", where "`X`", "`Y`" e "`Z`" are characters that represent the numbers of the class ("turma"), group, and practical assignment, respectively.

References

[Araújo, 2014] Rui Araújo. *Operating Systems*. DEEC-FCTUC, 2014.

[Kernighan e Ritchie, 1988] Brian W. Kernighan e Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc, Englewood Cliffs, New Jersey, USA, Second ed., 1988.

[Robbins e Robbins, 2003] Kay A. Robbins e Steven Robbins. *Unix Systems Programming: Communication Concurrency, and Threads*. Prentice–Hall, Inc, Upper Saddle River, NJ, USA, 2003.

[Stevens, 1990] W. Richard Stevens. *UNIX Network Programming*. Prentice-Hall, Inc, Englewood Cliffs, New Jersey, USA, First ed., 1990.

[The IEEE and The Open Group, 2013] The IEEE and The Open Group. [POSIX Specification] The Open Group Base Specifications Issue 7; IEEE Std 1003.1$^{\text{TM}}$-2013 Edition. 2013. [Online]. Available: `http://www.opengroup.org/onlinepubs/9699919799/` .

[Threads - Referência Rápida, 2005] Threads - Referência Rápida. *Threads - Referência Rápida*, 2005. Disponível na página da disciplina em: "`http://www.isr.uc.pt/~rui/so/thread_ref.pdf`".