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

Sistemas Operativos - 2019/2020 – 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 "producer/consumer" paradigm.

2 Material for Preparation

- [Stallings, 2004,
  Section 5.5, "Bounded-Buffer Producer/Consumer Problem Using Messages". pp. 240-241,
  Section 6.6, "Dining Philosophers Problem"].

- [Robbins e Robbins, 2003,
  Chapter 14, "Critical Sections and Semaphores";
  Chapter 15, "POSIX IPC", including
  Section 15.1, "POSIX:XSI Interprocess Communication",
  Section 15.2, "POSIX:XSI Semaphore Sets",
  Section 15.3, "POSIX:XSI Shared Memory",
  Section 15.4, "POSIX:XSI Message Queues";
  Chapter 12, "POSIX Threads";
  Chapter 13, "Thread Synchronization";
  Section 8.6, "Handling Signals: Errors and Async-signal Safety";
  Appendix B, "Restart Library"].

- [Marshall, 1999,
  "IPC: Semaphores";
  "IPC: Shared Memory";
  "IPC: Message Queues: `<sys/msg.h>`";
  "The POSIX Threads Library: `libpthread`, <pthread.h>"].

- [Leroy, 2001c,
  "The Linux Thread Library"].

- [Leroy, 2001a,
  "LinuxThreads README"].

- [Leroy, 2001b,
  "LinuxThreads Frequently Asked Questions"].

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

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

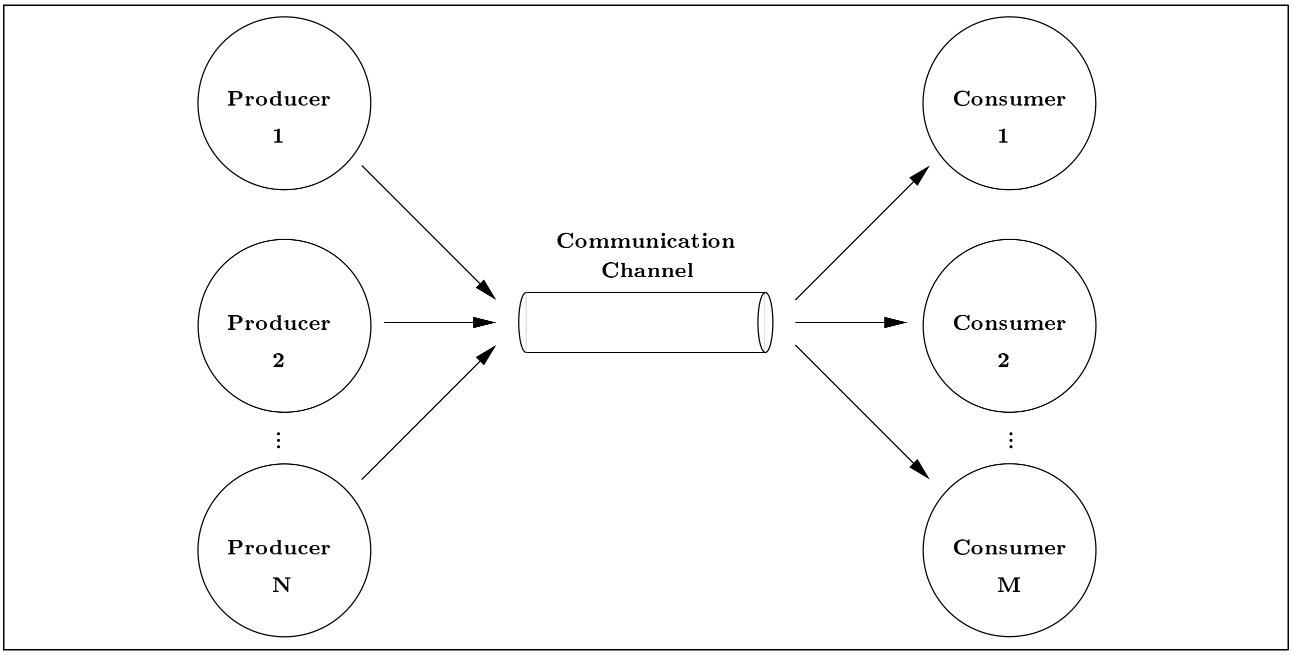

Figure 1: The producer/consumer paradigm.

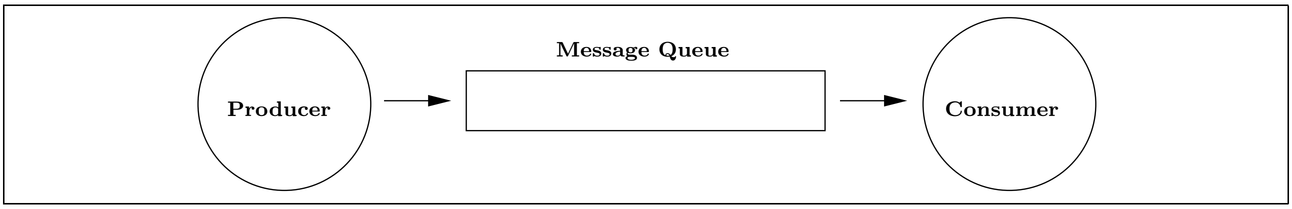

Figure 2: Producer/consumer with only two processes and a message queue.

3 Introduction

A well known concurrent programming paradigm is the "Producer/Consumer" paradigm. In this paradigm, there are one or more processes that produce information, and the information is processed, or consumed by one or more processes. Figure 1 presents an illustrative diagram of the paradigm. Problem 1 (Section 5) will deal with this paradigm.

On the other hand, one of the fastest methods for two processes to communicate is through the utilisation of shared memory. From the moment that two processes are using the same shared memory segment, the communication is performed by writing and reading directly in memory. However, it is required to maintain the coherence of the writing and reading operations on the shared memory area, through the use of some synchronisation mechanism such as, for example, semaphores. Problem 2 (Section 5) will deal with shared memory and semaphores. Problem 3 will deal with semaphores.

The practical assignment will also deal with threads.

4 How Does the Simple Producer/Consumer Work?

As an aid to the assigned work, it is suggested that students implement, before the work, a simple producer/consumer program[1]. Even if you decide not to implement, make an effort to understand very well its operation, as it will be helpful.

Figure 2 illustrates the working principle of a simple producer/consumer. There are two processes running, the **Producer** process, and the **Consumer** process. The producer process will read from a

[1] It is not obligatory to make the implementation of the simple producer/consumer. However, such implementation is very recommendable, and that it will help to make the assigned work.

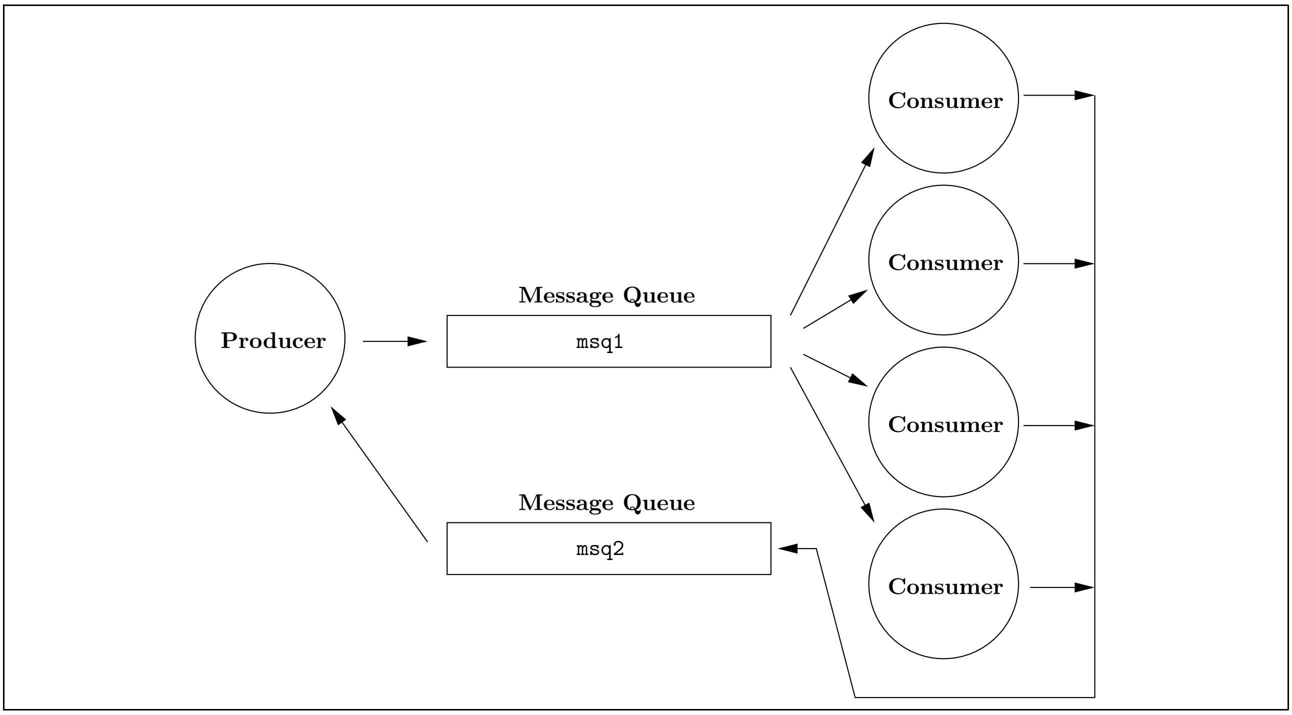

Figure 3: 1 Producer / N Consumers using message queues.

file (produce) a set of data that represents elements that should be processed (e.g. debit and credit operations on a bank account). Whenever it reads a data item, the producer process sends the information read (a data structure) to a message queue. When there is no more information on the file, the producer process sends a "virtual element" to the message queue with a *flag* indicating that there is nothing more to process.

The consumer process is continually waiting to receive, and receiving, information from the message queue. The consumer should terminate when it receives the virtual element indicating termination.

The program should be prepared to terminate without leaving occupied resources on the system, both on the normal operation/exit case, and in case of error. For this purpose, all the relevant signals should be redirected (see [`man 7 signal`], [Robbins e Robbins, 2003, Table 8.1, page 257]).

5 Assigned Work

Problem 1. Producer/Consumer using message queues. In this work a version of the Producer-Consumer paradigm using processes and message queues will be implemented.

The objective is to implement an automatic key generation system for a raffle. There will be a process, the producer, that generates random numbers between 1 and 49, and N consumer processes that will group these values into sets of 6 different numbers. The transmission of the generated numbers from the Producer to the Consumers will be performed using a message queue, `msq1`. Repeated numbers are not permitted in the key.

The producer process should be permanently generating random numbers, between 1 and 49. Each generated value should be to sent to message queue `msq1`. This mechanism permits to increase the program parallelism. The consumer processes keep receiving the values, one by one, from message queue `msq1`. Each different value that a consumer obtains is placed in a file that will save the final key, and written to the screen, correctly identifying the consumer that obtained it. When the key is complete, the file should be closed and the Consumer process terminates.

To test the system, we can define N (the total number of consumer processes) with a value of 4. The names of the files where the keys will be saved should have the format "`Key_`y`.txt`", where y is an integer value (between 1 and N) that identifies the consumer process. Figure 3 illustrates the working principle of the system.

Each message should have the capacity to hold one integer value, and the consumers read messages from the message queue `msq1`. The Producer process is continuously executing the following cycle: it generates a (random) value and sends a message with it to the message queue `msq1`.

The Consumer processes are continually executing the following cycle. Wait until there exists information to process (i.e. wait to receive from `msq1` a new message with the information, a random number), and receive such information. For every obtained value, the Consumer verifies if it is repeated or not, writes an informative message (identifying the process) on the screen, and in case the value is not repeated, then it appends the value to the corresponding file that will save the final key. When the key is complete, the Consumer closes the output file, writes a closing message to the screen, and terminates.

In addition to the above described behaviour, there is a second message queue, `msq2`, which transports messages carrying one integer named `mayproduce`. Before sending a new random number to `msq1`, the producer should receive from `msq2` one message with `mayproduce==1`. The producer, (only) at its very beginning should send to `msq2`, M messages with `mayproduce==1`. After a consumer receives one item from `msq1`, it should send to `msq2` one message with `mayproduce==1`. Each consumer, immediately before terminating, should send to `msq2` one message with `mayproduce==0`. To test the system, we can define M with a value of 40.

The program should terminate when all the consumer processes have terminated. The program should be prepared to terminate without leaving occupied resources on the system, both on the normal operation case, and in case of error. For that purpose it should redirect all relevant signals.

Work to be performed: Implement the above described application. Use a Makefile for compilation. Some of the required steps are as follows:

1.1 Start by writing the code necessary to create the message queues.

1.2 Implement the code for the producer and the consumers processes.

Criar signal handler

1.3 Implement in the program the code required to guarantee that the program terminates without leaving occupied resources on the system, both on the normal operation case, and in case of error. For that purpose all relevant signals should be redirected. fechar todos os ficheiros ...

1.4 Compile the program using `make`, and execute it. Analyse the output of the program in order to validate the results.

Problem 2. Shared memory and semaphores. Create two programs (thus involving two `main()` functions), a server and a client, that communicate using shared memory and semaphores. Suppose the existence of a file `input.asc` that contains floating point numbers (except the value `DBL_MIN`; see '`man float.h`') in ASCII format, one number per line. The client should read the `input.asc` file line by line (i.e. one number at a time) into a `double` variable, and then send the `double` to the server through the shared memory in binary format. At each moment, only one `double` variable should be in the shared memory. To properly communicate the values through the shared memory, the client and the server should synchronise their operations using POSIX:SEM named semaphores (see `sem_open()`, `sem_close()`, `sem_unlink()`, etc), and POSIX:MAP shared memory (`shm_open()`, `mmap()`, `ftruncate()`, `munmap()`, etc). After reading all the values from the file, the client should send the value `DBL_MIN` to the server to indicate that no more numbers will be sent, and then the client should terminate. For each number that the server receives from the client, the server should write it (append it in sequence) in binary format to the file `input.bin`. If this file does not exist it should be created; if it already exists, then it should be truncated to zero length before starting to write. After all the numbers have been written, the server should close `input.bin`, and then reopen it. Then, the server should read the all the numbers in `input.bin` into an array of `doubles` in memory with only one read operation. Next, the server should multiply by `4.0` each number of the array, and write (maintaining the order of the array of `doubles`) all the numbers to an ASCII file `output.asc` that has a format similar to `input.asc` (i.e. one number per line). Finally, the server should terminate.

Work to be performed: Implement the above described application. Use a Makefile for compilation. Some of the required steps are as follows:

2.1 Start by writing the code necessary to reserve a shared memory space to hold the shared data required by the application.

2.2 Then, write the code that will associate the shared memory area to the process address space.

2.3 Using the adequate functions define the POSIX:SEM name semaphores required by the program. The semaphores should be correctly initialised.

2.4 Implement the code for the client and server processes. Use the adequate functions to control the POSIX:SEM named semaphores.

2.5 Implement in the program the code required to guarantee that the program terminates without leaving occupied resources on the system, both on the normal operation case, and in case of error. For that purpose all relevant signals should be redirected.

2.6 Compile the program using `make`, and execute it. Analyse the output of the program in order to validate the results.

Mais simples

Problem 3. Threads. Make a program where the `main()` thread starts by asking the user to introduce two integers. Then, the program should create one server thread which should receive as its input arguments (using `pthread_create()`) two pointers to integers, one pointer to each of the integers introduced by the user. The server thread should sum and multiply both numbers. Upon termination, the server thread should return the calculated sum and product values through output arguments of the thread (using `pthread_exit()`).

The `main()` thread should wait for the termination of the server thread. After detecting the termination of the thread, the `main()` thread should print the sum and product values returned from the server thread (these values should come from the output arguments of the thread; no other method is permitted for the `main()` to receive these values), and then `main()` thread should terminate the program.

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-so20.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.

[Leroy, 2001a] Xavier Leroy. *LinuxThreads - POSIX 1003.1c Kernel Threads for Linux [README]*. INRIA, France, 2001a.
[Online]. Available: "`http://pauillac.inria.fr/~xleroy/linuxthreads/README`".

[Leroy, 2001b] Xavier Leroy. *LinuxThreads Frequently Asked Questions (With Answers) [for LinuxThreads Version 0.8]*. INRIA, France, 2001b.
[Online]. Available: "`http://pauillac.inria.fr/~xleroy/linuxthreads/faq.html`".

[Leroy, 2001c] Xavier Leroy. *The LinuxThreads Library*. INRIA, France, 2001c. [Online]. Available: "`http://pauillac.inria.fr/~xleroy/linuxthreads/`".

[Marshall, 1999] A. Dave Marshall. *Programming in C: UNIX System Calls and Subroutines Using C*. Cardiff University, UK, 1999. [Online]. Available: "`http://www.cs.cf.ac.uk/Dave/C/`".

[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.

[Stallings, 2004] William Stallings. *Operating Systems - Internals and Design Principles*. Prentice–Hall, Inc, Upper Saddle River, NJ, USA, fifth ed., 2004.

[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^{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`".