Universidade de Coimbra

Mestrado Integrado em Engenharia Electrotécnica e de Computadores

Sistemas Operativos

Practical Assignment nº2

Inter-Process Communication:

Semaphores, Shared Memory, Message Queues; Threads

Grupo 5 - PL3

Cristiano André Ramos Alves 2009109526

Luís Rafael dos Santos Afonso 2013135191

Devido ao uso de multiplos ficheiros e makefiles nos problemas, cada problema têm a sua pasta:

- Problem1
- Problem2
- Problem4

Todos os ficheiros foram comentados em inglês devido ao hábito e o facto de o enunciado estar em inglês influenciou isso inconscientemente.

Os códigos têm no cabeçalho um pequeno sumário funcional e como testar, assim nunca fica perdido do código.

Uso de makefiles:

Para facilitar o desenvolvimento e testes foi criado makefiles como mostrado nos slides, não só para compilar, mas para correr e limpar os ficheiros criados.

Com o terminal na directiva de cada problema basta executar:

- “make” ou “make all” para compilar
- “make play” para correr o executável - apenas para o problema 1

“make clean” para eliminar os códigos objectos e ficheiros criados em run time.

Problema 1

Objectivo (sumário):

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

Ficheiros:

- Source:
  - tpso1602.c
  - makefile - makefile para compilar, correr e limpar
- Executável:
  - tpso1602

Procedimento:

The instructions on the assignment we're followed and I think it's best to just see the comments on the code while reading the instructions. Procedures applied outside those instructions, or not that clear from reading them I will justify now.

For the semaphores a initial value of 1 was chosen since both of them is to stop reads/writes from any thread while another one is writing and also stop the "writers" when there are "readers".

The threads required some information from the process that created them. They required their number and the address of the shared memory mapped to the process memory. Since that's 2 arguments a struct called "thread_arg" was created for that purpose. There are 2 distinct types of threads in the exercise but both required the exact same type of arguments, so 1 struct was used for both types of threads.

For what was requested on the SIGINT handler called "cleanup", a global variable to hold the address of the shared memory was created since in the handler, to destroy the semaphores we needed that.
Also in the handler, the child process are killed by the parent as requested but it was probably not needed - the childs have the default SIGINThandler, they would have been terminated anyway.

Teste:

- Get a shell in the folder of the problem "Problem1" and execute "make" to generate the executable.
- Do "make play" to run the program.

Problema 2

Objectivo (sumário):

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.

Ficheiros:

- Source:
    - Problem2_receiver.c - source code for the receiver
    - Problem2_sender.c - source code for the sender
    - Problem2.h - common header file
    - makefile - makefile para compilar, correr e limpar
- Executáveis:
    - receiver
    - sender

Procedimento:

In my opinion there we're 2 big problems in this exercise. 1st, you can only have MAX_BYTES at a time in the queue. 2nd, how would the receiver know that he received the entire file? I could use a timeout, after x seconds of no data then the file would be finished - but errors could cause a timeout before the whole file was received so that was a no go.

So to solve this I decided that the first message would be the total number of bytes to be sent and that there would be packets (packets made more sense since I first used new message queues, not system V).

The sender will measure the file size in bytes, send that value to the receiver which then will know that only after receiving that number of bytes, the file will be completed.

Summary of process of each program:

Sender:

- Open existing key (number defined in the header)
- Set with msgctl() the msg_qbytes to MAX_BYTES
- Use lseek() to get file size
- Send file size by using each bytes as a part of a bigger number (byte 0 hold the least significant part while the rest holds the most significant part.)
- Read the file MAX_BYTES at a time and send them with msgsnd()

Receiver:

- Create existing key if it doesn't exist, if it does it needs to be changed in the header and do "make clean" followed by "make".
- Read the message with IPC_NOWAIT in a cycle with a sleep of 10ms. The cycle will occur enough times that it lasts the defined timeout value, if no message is received. If the timeout occurs the program will terminate and the file that would be created will not exist in the end.
- If it's the first message extract the number of bytes of the file
- The rest of the packets are written in order on the output file as they are received - since this was adapted from more recent message queues, measures we're taken so that the last packet would not cause programs (Ex: MAX_BYTES = 10, that means each packet is 10 bytes. If the packet only had 5 bytes with actual useful data then only 5 bytes will be writen into the file)
- After the number of packets we're all received the program will end.

Teste:

- Get a shell in the folder of the problem "Problem2" and execute "make" to generate the executable for the receiver and the sender
- Get a second shell in the same folder.
- In the first shell do "./receiver text_out.txt", where "text_out.txt" can be any name for a file to be created. The receiver will wait 5 seconds (can be changed) for a message to arrive.
- In the second shell do "./sender text_in.txt". "Text_in.txt" can be any file as long as it exists (.txt file is advisable).
- You should see the receiver reporting every time it receives a packet and when it's all done. The sender should also say when it's done. In the end "file_out.txt" should be a copy of "file_in.txt".