Pol Rubio
Rems Nalivaiko
24 de marc. de 2022
103221 - SISTEMES OPERATIUS
Fundació TecnoCampus Mataró-Maresme

LAB C 2

Pol Rubio
Rems Nalivaiko
24 de marc. de 2022

103221 - SISTEMES OPERATIUS

Fundació TecnoCampus Mataró-Maresme

Index:

Activity 1

Pol Rubio
Rems Nalivaiko
24 de març. de 2022
103221 - SISTEMES OPERATIUS
Fundació TecnoCampus Mataró-Maresme

activity1.c

Compile:

To compile our C programs, you only need to follow the following steps:
Open the terminal and type the following command:

```
gcc <name of the file>.c -o <name of the executable file> -lrt
-pthread
```

Execute program

Run the executable file by typing the following command:

```
./<name of the executable file>
```

Example

Here are some execution examples:

```
devasc@labvm:~/labs/sistemes-operatius-c2/activity_1$ ./a.out
bouncing for 8 times.
parent (pid=5686) begins
parent (pid=5686) bounce: 7.
child (pid=5687) begins
child (pid=5687) bounce: 6.
parent (pid=5686) bounce: 5.
child (pid=5687) bounce: 4.
parent (pid=5686) bounce: 3.
child (pid=5687) bounce: 2.
parent (pid=5686) bounce: 1.
parent (pid=5686) ends.
child (pid=5687) bounce: 0.
child (pid=5687) ends.
```

How does it work?

This is a C program that demonstrates the use of shared memory and semaphores between parent and child processes created using fork().

The program starts by including the necessary header files such as time.h, stdio.h, sys/types.h, stdlib.h, fcntl.h, unistd.h, sys/mman.h, and semaphores.h.

It then defines some constants such as MAX_NUM, SHARED_MEM_NAME, SHARED_MEM_SIZE, SEM_NAME1, and SEM_NAME2.

The main() function starts by seeding the random number generator using the current time.

It then generates a random number between 0 and MAX_NUM and prints a message indicating how many times it will bounce.

Next, it creates shared memory space, limits the space to 4 bytes, maps the memory space in the process, and generates a random number which is written to shared memory.

The program then creates two semaphores named SEM_NAME1 and SEM_NAME2 and initializes them using sem_open and sem_init functions.

The fork() function is then called to create a new process. The resulting_pid variable will contain the value of 0 in the child process and the process ID of the child process in the parent process.

The program then enters a loop that will continue as long as the value in shared memory is greater than 1. In each iteration of the loop, the process waits for its turn to write to shared memory using the sem_wait function.

After acquiring the semaphore, the process decrements the shared value by one, prints a message indicating its bounce number, and releases the semaphore using sem_post.

Finally, the program prints a message indicating whether the current process is the parent or the child and whether it's beginning or ending. The parent process waits for the child process to terminate before cleaning up the shared memory space and semaphores using the munmap, close, unlink, and sem_close functions.

Activity 2

Pol Rubio
Rems Nalivaiko
24 de març. de 2022
103221 - SISTEMES OPERATIUS
Fundació TecnoCampus Mataró-Maresme

activity2.c

Compile:

To compile our C programs, you only need to follow the following steps:
Open the terminal and type the following command:

```
gcc <name of the file>.c -o <name of the executable file> -lrt
-pthread
```

Execute program

Run the executable file by typing the following command:

```
./<name of the executable file> <number of iterations>
```

Example

Here are some execution examples:

```
devasc@labvm:~/Desktop/c/activity_2$ ./a.out 3
main: created pipe.
main: open pipe for read/write.
parent (pid=5423) iteration: 0
parent (pid=5423): 29 + 60 = ?
parent (pid=5423) iteration: 1
parent (pid=5423): 42 - 63 = ?
parent (pid=5423) iteration: 2
parent (pid=5423): 51 / 15 = ?
parent (pid=5423) ends.
child (pid=5424): 29 + 60 = 89
child (pid=5424): 42 - 63 = -21
child (pid=5424): 51 / 15 = 3
child (pid=5424) ends.
```

How does it work?

This C code demonstrates inter-process communication between a parent and a child process through a named pipe. The parent process creates the named pipe, sends random arithmetic operations to the child process through the named pipe, and waits for the child to complete the calculation. The child process reads the operations and operands from the named pipe, calculates the result, and sends it back to the parent process.

The code starts by including necessary header files for standard I/O, random number generation, inter-process communication, and file handling. The code defines some constants and initializes a few variables.

The parent process is implemented as a function named "parent." This function generates two random operands and an operator, writes them to the named pipe, and then waits for the child to complete the calculation. The child process is implemented as a function named "child." This function reads the operands and operator from the named pipe, calculates the result, and writes it back to the named pipe.

The main function reads the number of iterations to be performed from the command line argument, creates a named pipe, opens it for read/write, and forks a child process. If the fork fails, the program exits. If the fork is successful, the child process starts executing the "child" function, and the parent process executes the "parent" function. After the processes complete, the program closes the named pipe and removes it from the file system.

Activity 3

Pol Rubio
Rems Nalivaiko
24 de març. de 2022
103221 - SISTEMES OPERATIUS
Fundació TecnoCampus Mataró-Maresme

activity3.c

Compile:

To compile our C programs, you only need to follow the following steps:
Open the terminal and type the following command:

```
gcc <name of the file>.c -o <name of the executable file> -lrt
-pthread
```

Execute program

Run the executable file by typing the following command:

```
./<name of the executable file>
```

Example

Here are some execution examples:

```
devasc@labvm:~/labs/sistemes-operatius-c2/activity_3$ ./a.out
bouncing for 8 times.
thread1 begins, 8
thread2 begins, 8
thread1 bounce: 7
thread2 bounce: 6
thread1 bounce: 5
thread2 bounce: 4
thread1 bounce: 3
thread2 bounce: 2
thread1 bounce: 1
thread1 ends.
thread2 bounce: 0
thread2 ends.
```

How does it work?

This C program creates two threads that execute concurrently and access a shared resource using two mutex locks. The shared resource is a single integer variable located in a shared memory space that is created using the POSIX API calls shm_open(), ftruncate(), and mmap(). The main thread generates a random number that represents the number of times the two threads will access the shared resource in a ping-pong fashion, decrementing it until it reaches 1.

The handle_thread1() function is executed by the first thread, and it acquires the first mutex lock, prints a message indicating that the thread has started, and then enters a loop where it repeatedly decrements the shared integer variable, prints a message indicating the new value, and releases the second mutex lock. When the variable is equal to 1, the loop exits, and the function returns.

The handle_thread2() function is executed by the second thread, and it initially releases the first mutex lock (which is acquired by handle_thread1()), prints a message indicating that the thread has started, and then enters a loop similar to that of handle_thread1(), except that it acquires the second mutex lock before printing the new value of the shared integer variable. When the variable is equal to 1, the loop exits, and the function returns.

The main function creates the shared memory space and writes the random number to the shared integer variable. It also initializes the two mutex locks and spawns the two threads, passing a thread_data_t struct that contains a pointer to the shared integer variable and pointers to the two mutex locks. The main function then waits for the two threads to finish using pthread_join(), destroys the mutex locks, and frees the shared memory space.

Overall, this program demonstrates how to use POSIX threads and shared memory to coordinate access to a shared resource using mutual exclusion.

Bibliography

- https://stackoverflow.com/
- https://github.com/leonardjaner/OS-2022-2023/
- https://scholar.google.com/