Scaletta Gioele ID: 811106170

CSCI4730/6730 – Operating Systems

Project #2: Multi-threaded Web Server

Structure of the program (division of tasks between theads)

In the submitted program, almost all the work is accomplished by three functions (*thread_control, myhtread, listener*) which manipulate a bunch of global variables. Among those, it has to be emphasized an array which is our buffer structure for the consumer-producer problem. The position of the first and last inserted element are always monitored through two global variables (*in* and *out*). The function *thread_control*, which is called by the main takes care of creating and managing the listener thread and the consumers threads. Function *mythread* executes the requests which are present in the buffer. Function *listener* communicates with the client and put in the buffer all the requests/tasks.

The library *semaphores*.h is used to avoid race conditions and to regulate the access to the buffer as better explained later.

Control function (*thread_control*)

This function creates and sets all the thread in an optimal way for our producer-consumer situation. Moreover it takes care of the crash handling. Using *pthread_create* a listener thread, which will execute the *listener* function (passed as argument), is created. *Pthread_create* is also used in order to create as many thread as required which creating a pool of threads which compete for executing the tasks. In this case the function passed as a parameter is *mythread.*

Producer function (*listener*)

Every incoming request is saved by this function in the *array.* The variable in is incremented by one every time a request is added to the buffer so that the position of the last added element is always known. To avoid race conditions and exceeding the buffer size, semaphores are used. Before entering the critical section (manipulation of the *array*) it is necessary to call *sem_wait* both for *empty* (to signal that we are adding an element to the buffer and to stop in case the buffer is already full) and for *mutex* (this semaphore is the one which avoids the race condition and guarantee that the array is manipulated by only one thread at a time). After the critical section the "lock" *mutex* is realised and also the semaphore *full* is updated. Moreover it has to be mentioned that the semaphores are initialized using the function *sem_init*. As in the producer-consumer problem empty is initialized to *numThread* and full to 0.

Consumer function (*mythread*)

The function *mythread* executes the required tasks. All the producer threads will execute this function (maybe at the same time) but only one thread at a time will execute the critical part. In this case both the variable *out* and the *array* has to be protected. The procedure is very similar to the producer-consumer: *sem_wait* is applied to both *full* (if the buffer is empty we have to wait) and *mutex* (critical section "locked" as discussed before). While *sem_post* to *empty* (and of course to *mutex*). To make the whole process faster the function which manages the request (*process*) can be called after the critical section. As a matter of fact the critical issue is only related to *out* and *array,* once the value of the task that has to be completed has been extracted from the array.

Crash handling

The crash handling part is very straight-forward once the correct function to be used is found (*pthread_tryjoin_np*). As a matter of fact, this non-blocking function allows us to identify crashed threads as soon as possible and create them again. Being not necessary to complete the task which was being handled by the crashed process, the creation of the new process is identical to the one described before.

Conclusion

The above described implementation allows the task to be divided between more than one thread, making the whole process much faster. Semaphores are much more convenient in terms of performance with respect to any form of busy-waiting. Conversely to programming assignment one, in this case, communication between threads is very frequent and using inter process communication would have resulted in a much more complicated and tedious (and slower) program. In this project the better collaboration achievable by thread is definitely an advantage. Thus it is more convenient to use threads instead of processes. (Also because linux does not support semaphore among processes)

NB: The Makefile has been modified by changing –lpthread in –pthread in order to making the linking part effective. So a compilation error can be due to this modification, however the program was correctly running also on nike on the cluster vcf0.