PROGRAMACIÓN PARALELA
Linux pthreads

Linux processes are quite heavy for our purposes; managing them implies time overheads... On the other hand, threads might be better suited for what we are looking for; as a matter of fact, they are also called lightweight processes since they have a minimum set of attributes.

Working with pthreads

The first program you should execute is called `thread0`; it uses some system calls to show how to obtain the process and thread Ids. Once the `pthread0` process starts executing, one single thread (the *main thread*) executes the `main` function. When the `pthread_create` system call is executed, there is a second, independent thread, which starts executing the function called `thread_function`. The main thread waits for the spawned thread using the `pthread_join` system call. Note that the parameters in this call specify the thread id (`a_thread`), and the address of a variable to store the exit code of the spawned thread (`&thread_result`). The spawned thread, on the other hand, finishes its life when it executes `pthread_exit`. Note that the parameter for this function ("`Bye!`") is returned to the main thread. Finally, keep in mind that there are two threads (the main thread, which executes the `main` function; and the spawned thread, which executes the `thread_function` function) but just one single process.

The next program to test is called `thread1`. Since threads share the same address space of the parent process, global variables at the process level are shared by every thread that executes inside the process. Thus, in this case, since the `message` character array is declared outside any function, it has a global scope and therefore it can be shared by two threads: the main thread, which executes the `main` function and its spawned thread which executes the `thread_function` function. Note that in the `pthread_create` system call, the address of the string is passed as the last argument to the spawned thread. This thread, prints the original contents of the string, sleeps for 3 seconds, and then modify the string contents via the `strcpy` function.

The third program is called `thread2`. Here the global `run_now` variable is shared by two threads: the main thread, which executes the `main` function and its spawned thread which executes the `thread_function` function. The `run_now` variable is initialised to `1`. Therefore, the main thread, will put an `m` (its signature) in the screen and toggle the variable to `2`, thus enabling the spawned thread to put an `s` (its signature) in the screen; when the `run_now` variable is `2`, the main thread will sleep for `1` second. The code for the spawned thread follows the same principle. Therefore, the two threads will interleave execution: while one thread sleeps, the other runs and prints its signature.

The fourth program to try is called `thread3`; in this case, a semaphore is used to synchronise the execution of two threads. The `bin_sem` semaphore protects the `work_area` array and its initial

value is set to `0` using the `sem_init` function. Once the main thread creates a second thread, this thread gets blocked via the `sem_wait` function. The main thread waits for some keyboard input via the `fgets` function. Once the user enters some text, the main thread signals (via the `sem_post` function) the spawned thread that it can proceed to count the number of characters that were input by the user. Once the spawned thread finishes the count, it gets blocked again via the `sem_wait` function inside the `while` loop. Both threads continue executing until the user enters `end`.

The next program to try is called `thread4`; in this case, `10` threads are created. The first five threads execute function `thread_function1`; the remaining half execute function `thread_function2`. Threads use local variable `my_data` to distinguish themselves. To accomplish this, the main thread passes each thread a different value via the `&i` parameter (a pointer to an `int`; i.e., the address of a memory location that holds an integer). Since `pthread_create` expects a pointer to `void` as its last argument, we need to cast `&i` properly: `(void *) &i`. Each spawned thread in turn will receive the argument via the `arg_ptr` formal parameter. Since `arg_ptr` points to `void`, we need to cast it to the proper type (`int *` in this case): `(int *) arg_ptr`. Now we can access the argument sent by the main thread inside each thread by using the `my_data` variable: `my_data = *data_ptr`. Notice that to store the thread ids it is necessary to use an appropriate array (`pthread_t a_thread[10]`).

The last program to try is called `thread5`; in this case, we will create as many threads as logical cores in the executing machine exist, which can be obtained by reading the `_SC_NPROCESSORS_ONLN` Linux environment variable. As in the previous program, each spawned thread receives a different value via its `threadArg_ptr` parameter. The approach used in the code attempts to emulate the way *OpenMP* (a specification for shared-memory parallel programming) generates code. Since when using pthreads, we can only pass one parameter, if we need to pass more than a single value, a pointer to a C `struct` must be used. In this case, each spawned thread will receive its task number (`t_num`) as well as the total number of running tasks (`t`); this data will be packed inside a C `struct` (`threadData`).