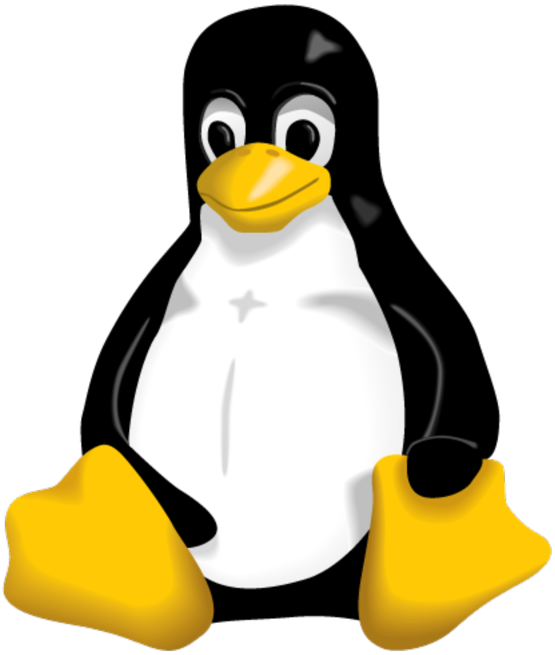
Report AOSV
A.Y. 2020/2021

Daniele De Turris (1919828)
Francesco Douglas Scotti di Vigoleno (1743635)

July 21, 2021



Contents

1 Introduction

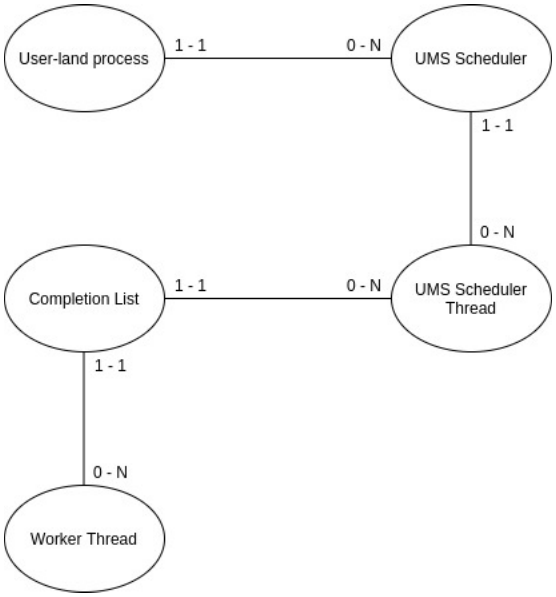
Our solution consists of a Linux LKM implementing a device driver, and a user mode library that interacts with it thus giving a programmer access to the facilities exposed by the module.
The development environment consisted of:

- OS: Ubuntu 20.04.2 LTS
- Kernel: 5.8.0-59-generic

The backend implementetion is based on 5 entities:

- User-land Process
- UMS Scheduler
- Scheduler Thread
- Completion List
- Worker Thread

and they are interconnected through these relationships:



1.1 UMS Scheduler

A UMS scheduler is used to keep track of groups of scheduler thread and to which process they belong.
Every process can have multiple UMS schedulers but if a fork occurs, a new one should be requested by the program.

1.2 Scheduler Thread

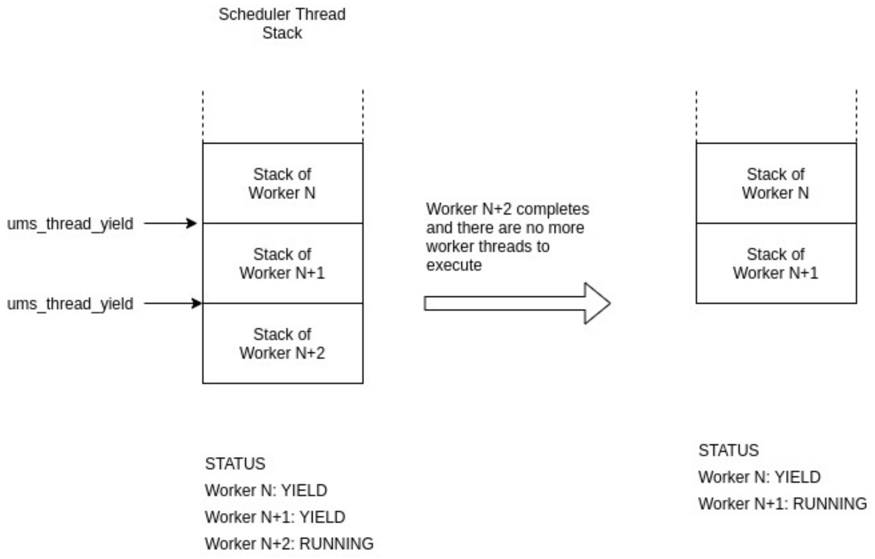
A scheduler thread is a standard pthread that has called the library function *enter_ums_scheduling_mode()*.
Immediatly after, it will start executing the entrypoint, that is programmer-defined function in charge of choosing the next worker thread to execute.
Once the worker thread to run has been chosen, the entrypoint will call *execute_ums_thread()* that:

- will flag the chosen worker thread as running and execute it immediately after if it is not running
- will return to the entrypoint if the chosen worker thread is already running. In this case the entrypoint will choose another worker thread to execute, having the guarantee that at least one of the workers returned by the *dequeue_ums_completion_list_itmes()* is runnable.

The scheduler thread will then start executing the worker thread, that in our implementation stands for calling the function pointer associated to that worker thread.
Once that occurs there only two possible scenarios:

- The worker thread completes, thus giving control back to the entrypoint function
- The worker calls a *ums_thread_yield()* that will suspend the execution of the current worker and puts it in the backlog of the current scheduler thread, and immediately after calls the entrypoint.

This mechanism is implemented by nesting the stack of a new worker thread inside the one that just called a *ums_thread_yield()*



1.3 Completion List

A completion list is a collection of worker threads.
Every scheduler thread has one completion list, but multiple scheduler threads can use the same completion list, as long as they belong to the same process, otherwise they wouldn't have in memory the function pointed by the worker threads' work function pointer.

1.4 Worker Thread

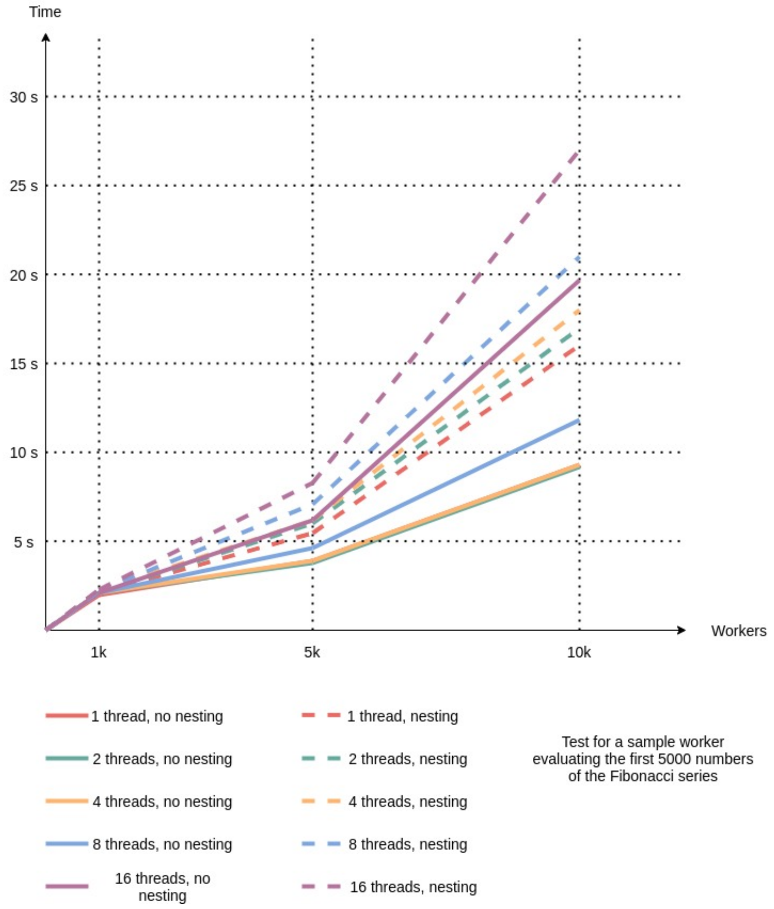
A worker thread is an abstraction of a job to be execute, thus, taking a hint from the *pthread_create()*, we opted to represent it via a function pointer.
Once a scheduler thread has decided it has to be executed, it just has to call the function. When the worker ends, control is given back to the entrypoint. If the worker calls a *ums_thread_yield()*, its execution will be stopped and put in the backlog of workers to be completed by the scheduler thread, resulting in the execution of entrypoint function.

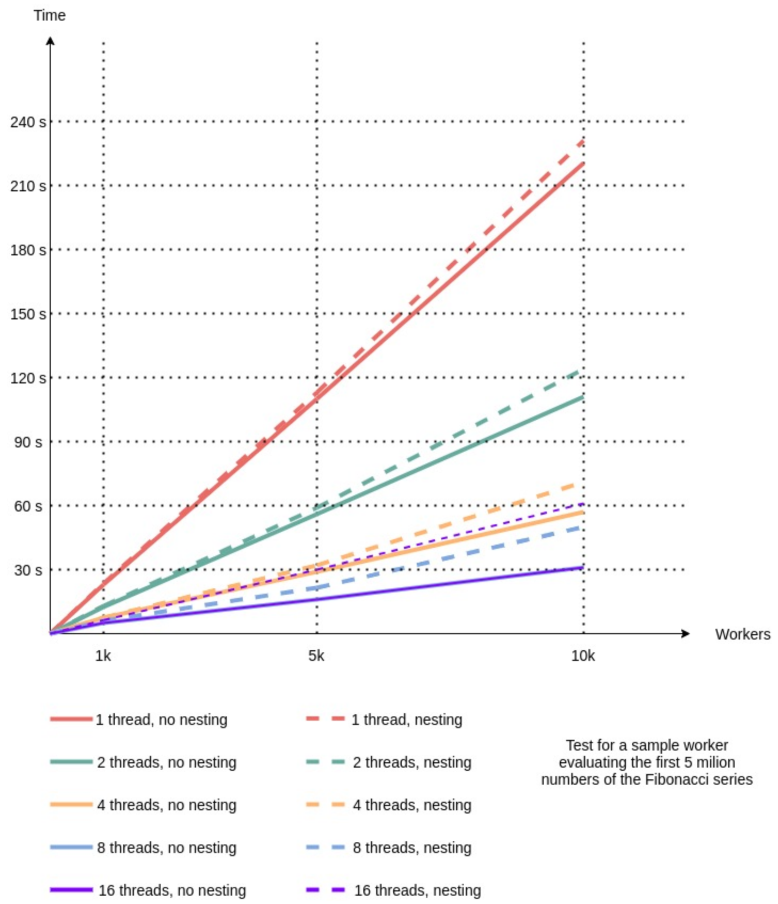
2 Results

The tests of the module were conducted on a i9 9880H limited to 2.3GHz with 8 cores to the VM, with a test user program using a single completion list to put it in its worst case scenario, but with two UMS scheduler having half of the Scheduler Threads each, thus resulting in a completion list being shared by Scheduler Threads pertaining to different UMS Schedulers.

2.1 Completion Time

By measuring the completion time of our sample test with various we can plot the results in the two following diagrams:





Test for a sample worker evaluating the first 5 milion numbers of the Fibonacci series

We can clearly see that with very light and fast workers the computational overhead overwhelms the module, resulting in degrading performances the more scheduler threads you spawn because of the synchronization required to keep everything consistent.
With a heavier workload we can see how the module scales almost linearly up until we reach a number of threads equal or greater than the number of cores of the system, where we see no further gains in terms of time, but a little loss when every worker yields.

In conclusion it must be noted that the test were performed with a single completion list, thus putting the module in his worst case scenario. As a result, the use of multiple completion lists and relative scheduler threads is highly recommended given how little the performance penalty is for using a number of scheduler thread greater than the number of cores.

All the datapoints used for these graphs have been collected in "doc/TestResultsSpreadSheet.xlsx".

2.2 Swithing Time

The mean switch time measured during our tests amounts to:

- between 2000 and 5000 nanoseconds in the case where no worker calls a *ums_thread_yield()*
- a maximum of 6-700000 nanoseconds (0.7 milliseconds) in the worst case scenario where every worker yields as the first instruction

This value can be also be observed in the
/proc/ums/SCHEDULER_ID/schedulers/SCHEDULER_THREAD_ID/info
entry of each scheduler thread, evaluated as the running mean of all the switching times.

3 Conclusions

Analyzing the constraints of our solution and its performance, it is pretty clear how an ad hoc solution for handling a specific multi-threaded problem could be faster then if implemented via our library,

For a general purpose project we can see how the use of our module could save the programmer's time by providing an easy to use interface that doesn't limit much his options, while still maintaining good performance for medium and heavy workloads.