Lab 2: Scheduling User-Level Threads and Comparing Them With Kernel-level Threads

CMPSC 473, SPRING 2019

Released on February 14, 2019, due on March 12, 2019, 11:59:59pm

Adrien Cosson, Yu-Tse Lin, Shruti Mohanty, Mitchel Myers, and Bhuvan Urgaonkar

1 Goals

This programming assignment (Lab 2) has three main goals. First, you will use the source code offered by us to understand how one may implement user-level threads. Second, you will implement the following CPU scheduling algorithms for these user-level threads: (i) round-robin (RR), (ii) vanilla Lottery Scheduler (vLOT) and (iii) a modified Lottery scheduler (mLOT) to be described in Section 3.1.1. You will validate the correctness of your implementation using some test cases that we will provide. Finally, you will design simple experiments to measure and compare the efficacy of user-level threads as a "vehicle of concurrency" vs. kernel-level threads along the following axes: (i) context switching overheads, (ii) handling of blocking IO, and (iii) ability to exploit multiple processors.

2 Understanding the Provided Code-Base

After accepting the invitational link to Lab 2 (please use the account for one of the members of your team and be sure to indicate the names of both members in all your files), you will be given access to your own private repository. Clone the repository for Lab 2 to obtain a collection of files containing our code and test cases. You will find the following files:

- The user-level threads implementation: `thread.c`, `thread.h`, `my_scheduler.c` available in the `User_Level_Thread` directory.
- Four parallel programs ("test cases") based on user-level threads (each contained within a separate file with a name such as `TestFile*.c`) for you to test your scheduler implementation. These files have definitions for functions that you will pass to the `CreateThread()` function of the user-level threads implementation. Each of these files has a `main()` function which calls the `CreateThread()` function multiple times to create a certain number of threads, and then calls the `Go()` function to start scheduling and executing them. You can find these files also in

the `User_Level_Thread` directory. Details on `CreateThread()`, `Go()`, and other associated functions will appear later in this section.

- Two functions that threads comprising your parallel programs will run called (i) `counter()` and `sleeping()`. You will find the prototypes and definitions of these functions in `functions.h` and `functions.c`, respectively, in the `Kernel_Level_Thread` directory. Both of these functions increment a counter variable for 10 seconds of real time. Whereas the `counter()` function increments the counter continuously, the `sleeping()` function sleeps for a short period of time (less than 10 seconds) before resuming counting. To use these functions, we have provided a program called `kt_test.c` which you can find it in the same directory. This file uses kernel-level threads (created using the `pthreads` library) that run these functions. When running this program, you can tell each thread to run either the `counter` function or the `sleeping` function by passing an argument to the program. You should use the `Makefile` included in the `Kernel_Level_Thread` directory.

User-level threads implementation: You must develop a clear understanding of how the provided user-level threads implementation works. Towards this, you will go over the code and combine this with material covered in class on context switching and user-level threads (Lectures 4-8). The main functions comprising our user-level threads implementation are:

- `int CreateThread(void (*f) (void), int cpuweight)`: this function creates a new thread with `f()` being its entry point. The function returns the created thread id (≥ 0) or -1 to indicate failure. It is assumed that `f()` never returns. A thread can create other threads. The created thread is appended to the end of the ready list (state = `READY`). Thread ids are consecutive and unique, starting with 0. The second argument (`cpuweight`) specifies the weight for a thread when creating it. The weight represents the CPU weight of the thread as employed by the vLOT and mLOT scheduling algorithms.

- `void Go()`: This function is called by your process to start the scheduling of threads. It is assumed that at least one thread is created before `Go()` is called. This function is called exactly once and it never returns.

- `int GetMyId()`: This function is called within a running thread. The function returns the thread id of the calling thread.

- `int DeleteThread(int thread_id)`: Deletes a thread. The function returns 0 if successful and -1 otherwise. A thread can delete itself, in which case the function does not return.

- `void Dispatch(int sig)`: The signal handler for alarm signals. The CPU scheduler will be invoked by this function. In our code, the scheduling algorithm is FCFS. It is assumed that at least one thread exists at all times - therefore there is a stack at any time. It is not assumed that the thread is in ready state.

- `void YieldCPU()`: This function is called by the running thread. The function transfers control to the next thread (in the scheduling order). The next thread will have a complete time quantum to run.
- `int SuspendThread(int thread_id)`: This function suspends a thread until the thread is resumed. The calling thread can suspend itself, in which case the thread will `YieldCPU` as well. The function returns `id` on success and -1 otherwise. Suspending a suspended thread has no effect and is not considered an error.
- `int ResumeThread(int thread_id)`: Resumes a suspended thread. The thread is resumed by appending it to the end of the ready list. Returns thread id on success and -1 on failure. Resuming a ready thread has no effect and is not considered an error.
- `int GetStatus(int thread_id, status_t *stat)`: This call fills the status structure with appropriate thread-related data. Returns thread id on success and -1 on failure.
- `void SleepThread(int sec)`: The calling thread will sleep until current time plus sec. The sleeping time is not considered wait time.
- `void CleanUp()`: Shuts down scheduling, prints relevant statistics, deletes all threads and exits the program.

Make sure you understand how these functions are implemented. Make sure you understand the data structures used by this code (e.g., what are `status_t` and `thread_t` and how they relate to the data structures we discussed as part of the lectures on context switching and CPU scheduling) - these are contained in the file `threads.h`. Make sure you understand the role of various macros that the code defines (e.g., what is `x86_64`?) Many macros are important constants:

- `MAX_NO_OF_THREADS 100` /* in any state */
- `STACK_SIZE 4096`
- `SECOND 1000000`
- `TIME_QUANTUM 1*SECOND`

Our code in the `User_Level_Thread` folder implements FCFS scheduling. You will be implementing RR, vLOT, and mLOT as described below.

3 Description of Tasks

3.1 Task 1: implement and test schedulers

3.1.1 Implement RR, vLOT, and mLOT

You are already familiar with RR and vLOT (from Lecture 6 or 9 depending on your section). mLOT is a modified form of the Lottery scheduling algorithm that uses a CPU

quantum size that is proportional to the weight of the chosen thread. For more details on the schdulers, see Appendix B. Your schedulers should implement the following interface:

- `thread_t *scheduler()`: Returns the thread (from the ready queue) which should run next.
- `void InsertWrapper(thread_t *t, thread_queue_t *q)`: When a thread's state changes from `SLEEPING` to `READY` or from `RUNNING` to `READY`, you have to add it to the appropriate queue accordingly.

Again, your code will be written in the file `my_scheduler.c`. By now, it should be clear to you that the (already implemented) FCFS scheduler always returns the thread at the head of the ready list. Your code for RR, vLOT, or mLOT will choose a thread from the ready list differently based on how these scheduling algorithms work. Note that you may implement other functions based on your needs as well.

3.1.2 Test the correctness of your RR , vLOT, and mLOT implementations

The `Makefile` in User_Level_Thread directory creates a static library named `libutl.a` which contains `threads.c, threads.h` and `my_scheduler.c`. You must link your `Testfile1.c`, ..., `Testfile4.c` (one at a time) against this library to compile and create the `project2` object file. This can be done by running :

```
make t=$X
```

where `$X` is the number of the test file.

After writing the code for RR, vLOT, and mLOT schedulers you should be able to run all test cases for each scheduler type (`$./project2 RR`, `$ ./project2 vLOT`, and `$ ./project2 mLOT`). These test cases are given just for illustrative purposes to test your code. As you will see, each test case consists of creating one or more threads which run specified functions. When vLOT or mLOT is to be used, certain weights are supplied. For example, `CreateThread (clean_up_thread, 1)` creates a thread which runs the `clean_up_thread` function and has a CPU weight of 1. This function prints the thread id within a loop and when the condition j=10 occurs, it shuts down scheduling, deletes all the threads and exits. The following statistics are printed for each thread after clean up:

- **num_runs**: shows the number of runs for each thread for all calls of Dispatch(),
- **total_exec_time**: shows the total execution time for the thread,
- **total_sleep_time**: shows the total sleeping time,
- **total_wait_time**: shows the total waiting time,
- **avg_exec_time**: shows the average execution time,
- **avg_wait_time**: shows the average waiting time.

- **num_continuous_runs**: shows the number of consecutive runs for the thread
- **avg_continuous_exec_time**: shows the average continuous execution time for the thread

NOTE: You are NOT supposed to modify any of the test cases. You should consider creating more extensive test cases for further testing. The TAs will test your code with other test cases during grading and your code should still work correctly.

3.1.3 Understand given parallel programs and re-implement them using user-level threads

As mentioned earlier, you have been given two functions, `counter()` and `sleeping()`, implemented using `pthreads`. Go over these functions, and make your own improvised `counter()` and `sleeping()` functions for use with user-level threads.

You should provide your implementations in the file `User_Level_Thread/my_functions.c`. You may write your own test files to test the validity of your implementation.

3.2 Task 2: evaluate pros and cons of user-level threads

You will conduct the following experiments to compare parallel programs implemented using kernel-level threads provided by us vs. the same programs implemented using user-level threads with the RR scheduler.

3.2.1 Context switch time

Use the `lmbench` benchmark to measure the context switching time for kernel threads on the lab machine you have chosen to work on. (Among the various measurements listed under context switching, you should report the time mentioned under the heading `2p/0K`. Make sure you understand what this heading means and what the other values are for.) Use this link to download the latest version of `lmbench`: http://www.bitmover.com/lmbench/lmbench3.tar.gz. Make sure you take several measurements for statistical significance and report the empirical average and variance across these. (Take the time to read up on these basic statistical concepts if you have forgotten them).

First, you will implement functionality to measure the context switching time within the user-level threads code.

Hint: you could introduce calls to a C library function like `clock_gettime()` (that has a microsecond or nanosecond resolution) to specific functions called in `Dispatch()` for instance.

Launch 2 threads each running this improvised `counter()` function for 10 seconds and measure:

1. Total number of context switches.
2. Time taken for each context switch in microseconds. Report the empirical average and variance across all context switches during your experiment.

3.2.2 Effect of a blocking call

As described earlier, the `sleeping()` function sleeps for a short period of time (less than 10 seconds) before resuming counting. Execute the `sleeping()` function using `kt_test.c` and compare the "work done" with the improvised `sleeping()` function to be used with user-level threads. Call the counter value reported by a thread the work done by it and the summation of the counter values reported by all threads of a program the work done by the program. Report your observation with a brief description. You can use any plotting tool of your choice. To use the plotting script provided in the repository, type,
`$ python plot_script.py <path_to_file>`
Your file should have exactly 6 entries to be compatible with this plotting script.

3.2.3 Using multiple processors

Vary the number of parallel threads within a program where each thread runs the `counter()` function. The number of threads should vary as {1,2,4,8,16,32}. Plot how the work done by a program varies with the number of threads when the program is based on kernel-level threads vs. user-level threads. Relate observed behavior to the number of processors on the machine on which you perform this experiment.

4 Submission and Grading

PA2 will be done in groups of 2. A group may choose either member's repository as their workplace. Do remember to list both members' names in all files you submit (including your report). Just like in PA1, you will submit all your source files, Makefiles, READMEs (to convey something non-trivial we may need to know to use your code), and a report containing the results as described earlier. Check out the updated Git Manual, included in your repository, especially Section 4.

The TAs will evaluate your submission based on (i) the correctness of your implementation as part of Task 1 - this will be based on the number/subset of test cases your code passes (a subset of the test cases will be provided to you) and (ii) the quality of your report on Task 2. The rough distribution of points will be as follows:

- RR implementation and testing: 10%
- vLOT implementation and testing: 20%
- mLOT implementation and testing: 20%
- Context switch time for kernel-level threads using `lmbench`: 20%
- Context switch time for user-level threads: 20%
- Effect of I/O blocking: 5%
- Multiple processors: 5%

We offer miscellaneous tips here.

A SSH connection

- While using `ssh` to log into a department Linux machine, please use the X11 forwarding option by passing the `-X` option like so:

  `$ ssh -X <username>@cse-p204instxx`
  where `xx` is some machine number.

B Scheduler types refresher

B.1 Round Robin

This scheduler allocates time quanta to all threads in a cyclical fashion. When a thread's quanta is finished, this thread is added at the tail of the ready queue, and the thread at the head of the ready queue is started.

B.2 Vanilla Lottery

This scheduler allocates threads at random according to their weights. Each thread is given a weight (that can be interpreted as a priority of some sorts).

When the scheduler has to pick which thread will run, it will pick a thread at random in the ready queue by following the following probability rule:

$$P_i = \frac{w_i}{\sum_k w_k} \tag{1}$$

where P_I is the probability that thread i will be picked, and w_i is the weight of thread i.

B.3 Modified Lottery

This scheduler allocates thread at random, regardless of their weights. However, when a thread is picked, instead of running for one time quanta, it will run for his weight times the time quanta. This means that you have to keep track of how many time quanta this thread has already expended, and compare it to its weight to know if you should schedule it again or pick a new random thread to run.