CS355 Hw 4

Due by the end of day **Sunday, March 10**. Tests posted to github by **Friday March 8**.

User-level Threads You will write a library to support multiple threads within a single Linux process. This is a "user-level thread library" because the kernel doesn't know these threads exist - they are just something the library implements and multiplexes onto a single process. While there are differences between user-level threads and the kernel-level threads we've so far focused on in class and in hw2 (`pthread`), building and scheduling actual kernel-level processes and threads would not be much different, that is, essentially you are now building a part of `pthread` from scratch.

Interface

An interface to the thread library you will create has been defined for you. You will write your own implementation of this interface. Prototypes for these functions are found in `~dxu/handouts/cs355/userthread.h`. You should include this header file in your code to ensure that you are adhering to this specification exactly. You should not change `userthread.h` in any way!

- `int thread_libinit(int policy)`
  `thread_libinit` is called to initialize the thread library. `policy` is one of the following enum ints: FIFO, SJF, or PRIORITY (see Scheduling section below).

  return value: Upon success, `thread_libinit` returns 0. Upon failure, it will return -1.

- `int thread_libterminate(void)`
  `thread_libterminate` is called to shut down the thread library and do all necessary clean ups.

  return value: Upon success, `thread_libterminate` returns 0. Upon failure, it will return -1.

- `int thread_create(void (*func)(void *), void *arg, int priority)`
  `thread_create` is used to create a new thread. The last argument `priority` specifies the priority level if priority scheduling is selected. When a newly created thread starts, it will call the function pointed to by `func` and pass it the single argument `arg`.

  When a thread calls `thread_create`, the caller does not yield the CPU. The newly created thread is put on the ready queue but is not executed right away.

  return value: Upon success, `thread_create` returns a positive integer thread ID that can be used later to call `thread_join` and thus wait for a given thread to terminate. Upon failure, it will return -1.

- `int thread_yield(void)`
  `thread_yield` causes the current/calling thread to yield the CPU to the next runnable

thread. The yielding thread should be put at the tail of the ready queue. If the caller is the only runnable thread, it will be selected to run again, in other words, the yield will have no effect.

return value: Upon success, `thread_yield` should return 0. You may assume that the call always suceeds, which is on par with the current Linux `pthread_yield` implementation.

- `int thread_join(int tid)`
  `thread_join` suspends execution of the calling thread until the target thread terminates.

  return value: Upon success, `thread_join` should return 0 (this includes, of course, the case where the thread is already finished). If given a bad ID or if any other failure occures, `thread_join` should return -1.

Scheduling

As part of this assignment, you will experiment with three scheduling policies: first-come, first-served (FIFO), shortest-job first (SJF) and priority (PRIORITY).

- **First-come, first-served (FIFO)**
  Threads on the ready queue should be scheduled according to the order they arrive, without preemption. Pass FCFS to `thread_init` to select this schedule.

- **Shortest-job first (SJF)**
  Threads on the ready queue should be scheduled according to their estimated run time. Here, you need to keep track of how long a thread runs before calling `thread_yield` or `thread_join`. Pass SJF to `thread_init` to select this schedule.

  Because we cannot know how long a job will take to run, we will use an approximation to implement SJF. You should use the average of the last three times the thread ran to compute its job time. In addition, when a job is created, you should assume it is an "average" job and set its runtime to the average runtime of all threads so far. If there is no runtime history at all (i.e first thread ever), use a reasonable default value (for example, half of the quanta - see below)

- **Priority-based (PRIORITY)**
  Pass PRIORITY to `thread_init` to select this schedule. An integer argument representing a thread's priority is passed to `thread_create` at the time of thread creation. Based on UNIX convention, the smaller the number is, the higher the priority. In this assignment, there are only three levels of priorities -1, 0, 1. For FIFO and SJF, you can disregard this parameter. For priority scheduling, threads scheduled with level -1 should run 1.5 times more often as jobs scheduled with priority level 0, which run 1.5 times more often as jobs scheduled with priority level 1. It is your scheduler's responsibility to ensure this ratio is exact.

  For priority scheduling, you will schedule a SIGALRM signal to be delivered to your scheduler every 100 milliseconds. We refer to this event as a clock tick, and on every clock tick the scheduler will come in and choose which thread to run next. To set an alarm timer at millisecond granularity, refer to `setitimer(2)`.

Threads Context Switching

- **Creating and swapping threads**: You will be implementing your thread library on Linux, as usual. You may not use pthreads to implement your library. Instead, Linux provides some system calls (`getcontext, setcontext, makecontext, swapcontext`) to help implement user-level thread context switching. Please refer to lab 4 for more details about user contexts.
- **Deleting a thread and exiting the program**: A thread finishes when it returns from the function that was specified in `thread_create`. Remember to de-allocate the memory used for the thread's stack space and context (be careful to do this after the thread is really done using it).

  When you create a thread, you may find it useful to invoke a "wrapper" function (also called a stub function) that calls the thread's function and, thus provides a place for the thread to return to upon termination, where the necessary cleanup can be performed.

Atomicity

Threads share data structures. Your signal handler can interrupt any of your threads right in the middle of a `thread_*` library call. It is a very bad idea to have multiple threads accessing your ready queue at the same time. The good news is that you have written a shell with much of the same challenges, so you know what to do.

Scheduler Logging - userthread_log.txt

To help evaluate the functionalities of your scheduler, your scheduler should write to a log file (name it userthread_log.txt) every time it comes in for context switching. This is required for all three scheduling algorithms (FIFO and SJF may leave the PRIORITY column empty). Document clearly the name of the log file and its path. The logger should overwrite logs of the last execution of the scheduler program. That is, it should not accumulate but only record the current session's information. The log will have the following format (tab deliminated):

```
[ticks]  OPERATION    TID           PRIORITY
```

For example, here is a snippet of a logger:

```
...
[100]  SCHEDULED      1                   -1
[200]  STOPPED        1                   -1
[200]  SCHEDULED      2                   -1
...
```

Operations require logging are CREATED, SCHEDULED, STOPPED, and FINISHED. You should log any other information you find helpful as long as you document it clearly. Logging is also a great tool to help you debug your program. It is recommended to implement this feature as soon as possible.

`printf` is not re-entrant and thus there is a risk using it with signal handlers. It's not a big deal when you are just printing a message for debugging, but for the logger, you should use the system call `write` and `STDOUT_FILENO` instead. `fprintf` is thread-safe and thus if you are logging to a file (which is what you should be doing), you are safe.

Valgrind Compatibility

Unfortunately Valgrind by default does not work very well with `ucontext`. To make it work, you will need to include `valgrind.h` header file (Path:`/usr/include/valgrind`) and use `VALGRIND_STACK_REGISTER` and `VALGRIND_STACK_DEREGISTER` to explicit tell valgrind which stack space you are using right now. Please see the example provided by Valgrind at `https://github.com/lu-zero/valgrind/blob/master/memcheck/tests/linux/stack_changes.c`. Please pay special attention to the `init_context` function and see that valgrind requires a integer id to label each stack and to keep track which stack space is associated with which context.

Error Handling and Testing

An integral (and graded) part of writing your thread library will be to write a suite of test cases to validate your thread library. This is common practice in that real world software companies maintain a suite of test cases for their programs and use this suite to check the program's correctness after a change. Writing a comprehensive suite of test cases will deepen your understanding of how to use and implement threads and will help you as you debug your thread library.

Each test case for the thread library will be a short C program that uses functions in the thread library. Each test case should be run without any arguments and should not use any input files. Test cases should `exit(EXIT_SUCCESS)` when run with a correct thread library. Your test suite may contain up to 20 test cases and should contain at least 10. Each test case may generate at most 10 KB of output and must take less than 60 seconds to run. Note that these are upper limits and are MUCH larger than needed for full credit. You will submit your suite of test cases together with your thread library, and we will grade your test suite according to how thoroughly it exercises a thread library. Your README should either contain a testing section, or you should provide an additional README for the test suite.

OS programmers must have a healthy(?) sense of paranoia to make their system robust, so part of this assignment is thinking of and handling lots of errors. Unfortunately, there will be some errors that are not possible to handle, because the thread library shares the address space with the user program and can thus be corrupted by the user program. A real kernel implementation doesn't suffer from this so you may safely ignore this type of errors. In general, keep the test cases short and narrow. It's better to test only one thing so that when a program fails a test, you know exactly why and what to look for. Name your test cases accordingly. For each test case, you should have a header that explains what it tests for, AND (this is important) what the expected CORRECT behavior is.

Since it is hard to imagine all possible scenarios, a public github repository `https://github.com/diannaxu/BMC-CS355.git` is setup for the course where you may post your test cases to. Each of you can test your library on other's tests to help you debug your program. You are not obligated to post all the test cases you come up, but not posting any will cause deductions. Ideally, each of you will look at what's already there and try to add new test cases that will add diversity. I will leave it up to you to decide how many and how extenstive your test contributions are. I only request that you upload to github by Friday 3/8, because it doesn't leave enough time for the other students to test and debug if you contribute any later.

There is already a test suite from previous classes that you are welcome to use as well.

Also be advised that starting with this assignment we will be testing with automatic scripts. The scripts need to have uniform naming/behavior expectations or things will break and you will fail the tests and/or create a lot of unnecessary work for your TA. In particular, make sure that your Makefile works with the default target, i.e. just the command "make" with no additional arguments. In addition, make sure that you create only one library and your libary is named "libuserthread.so", exactly.

Suggestions/Hints

1. Remember that the main thread (the one that's making calls to your library) is also a schedulable context and should not be treated differently.

2. For non-preemptive scheduling policies, when your main thread calls `thread_join`, you should call your function that performs scheduling. This is how the first thread gets to execute.

3. A minimum thread control block is required to do the house keeping of context switching. Your TCB should minimumly include the tid, ucontext and (CPU usage or priority, depending on scheduling algorithm), but you will probably add more.

4. To implement priority scheduling, you should create 3 linked-lists, one for each priority. Then schedule in a Round Robin fashion among all of them.

5. Note that when a process terminates, often times it may not use up all of 100ms. It is your scheduler's responsibility to handle this situation well and ensure the process next to be scheduled will be given a full 100ms quanta.

6. Declare all internal variables and functions (those that are not called by clients of the library) `static` to prevent naming conflicts with programs that link with your thread ilbrary.

7. Assumptions

   (a) You may assume that we only do one join per thread id.

   (b) You may assume that we will not call `thread_yield` from the main thread.

   (c) Your library functions shouldn't print any error messages. Errors are indicated by return values, as is typical with libraries.

   (d) A "run" is defined as the time between a thread being scheduled and the thread being descheduled (via a call to `thread_join`, `thread_yield`, or the thread's function finishing). So, any time a thread is scheduled, you should start timing; when a thread is descheduled, stop; the elapsed time defines the previous "run."

(e) You may assume that we will only invoke `thread_libinit` with a valid scheduling policy.

Extra Credit

Note that it is your responsibility to provide sufficient logger output and/or generate additional tests to prove the functionality of any extra credit components that you implement. If we can not tell that it works, no credit will be granted. You must also clearly document in your README which extra credit you implemented, and how to test them.

1. Change SJF to use aging to estimate execution time, rather than the average of the last three runtimes.

2. Change SJF from non-preemptive to preemptive.

3. Implement priority scheduling to be more UNIX like such that a process gets penalized when it does not finish in n quanta (up to you to define what n is). That is, its priority increases and it will be scheduled less often.

4. Farther down the road of UNIX, implement priority scheduling such that each priority gets a different quanta, meaning process with larger priority number is scheduled less often but its quanta is also longer.

5. Any other significant improvements on the scheduling algorithms. Please document well.

Note that if you implement any extra credit, it is then your responsibility to clearly demonstrate the corresponding functionality. You should create targeted test cases and generate corresponding logger output that shows the extra credit components working. You will not receive credit if we can not tell that it works.

Appendix A: Building a shared library

Your thread library needs to be built into a dynamically-linked shared library. The library should be called *libuserthread.so*. These are the basic steps to build and use a dynamically-linked shared library:

1. Compile each `.c` file

   For each `.c` file that comprises your library (e.g., `userthread.c`), you should compile as follows:

   ```
   gcc -Wall -fpic -c userthread.c
   ```

   Here, the `-Wall` flag prints all warnings, as usual; the `-fpic` flag tells the compiler to use "position-independent" code, which is good to use when building shared libraries; finally, the `-c` flag tells the compiler to create an object file (in this case, `userthread.o`).

2. Link the object files together to create a shared library

   Once you've built all your `.o` files, you need to make the shared library:

   ```
   gcc -shared -o libuserthread.so userthread.o
   ```

3. Linking a program with your library

   Let's say you have a program, `test.c`, that wants to use this thread library. First, `test.c` should include the header file "`userthread.h`"

   To build `test`, you need to link it with your library (assuming all of your code is in the same directory):

   ```
   gcc -o test test.c -L. -luserthread
   ```

   The `-luserthread` flag tells the compiler to look for a library called `libuserthread.so`, and the `-L.` flag tells the compiler to look for the library in the "`.`" directory (in Unix, "`.`" refers to the current directory).

   Before you can run `test`, you'll need to set the environment variable `LD_LIBRARY_PATH` so that the system can find your library at runtime. If you're using bash (echo `$SHELL` to find out), you can use the command:

   ```
   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
   ```

   or, for tcsh:

   ```
   setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:.
   ```

   You can also add this command to your `$HOME/.bashrc` (`$HOME/.cshrc` for tcsh) to avoid having to run it each time.

   If the `export/setenv` command returns an error "LD_LIBRARY_PATH: Undefined variable", do not panic. The error implies that your shell has not defined the environment variable. In this case, you should use instead:

   ```
   export LD_LIBRARY_PATH=.
   ```

   or, for tcsh:

   ```
   setenv LD_LIBRARY_PATH .
   ```

Deliverables:

1. Electronic submissions of the programming assignments.