KFC: Kernel-Free Concurrency

These notes are in the process of being converted into a nicer format. However, they are complete enough at this point to be able to finish the project.

In this project you will be implementing a library called KFC (for Kernel-Free Concurrency). This library provides the userspace portion of a many-to-one or many-to-many threading model, depending on the number of kernel threads requested at initialization. Your interaction with kernel threads will be by calling the `kthread` functions listed in `kthread.h`.

Since you are essentially implementing the context-switching piece of an operating system, you will find useful many of the concepts introduced in the textbook's chapters on processes and threads: process control blocks, ready queues and device queues, short-term scheduler vs. dispatcher, etc. You do *not* need to worry about preventing or detecting deadlock between your threads.

Getting started

Important note: *do not fork this repository through the GitHub UI.* GitHub apparently will not allow a public repo thus forked to be made private. Instead, create your own private repository (don't initialize with a README), then clone mine to wherever you would like to work, and push to yours:

```
$ git clone -o upstream https://github.com/devinpohly/csci455-project2.git
$ cd csci455-project2
$ git remote add origin https://github.com/your-username/your-repo-name.git
$ git push -u origin master
```

(The last two lines are also found on the GitHub quick setup page that appeared when you created your own repository.) At any later time, you can merge in changes from my repository using:

```
$ git pull upstream master
```

Project files

- `kfc.h` and `kfc.c`: this is where all of your work will go. The header file defines the API that your KFC library provides, and the C file initially contains stubs for each of the functions.
- `kthread.h` and `kthread.c`: abstraction of kernel threads. These are mostly wrappers for Pthreads functions, with a few tweaks to make your life easier. You may look through the implementation if you wish, but it will not likely be helpful.
- `queue.h` and `queue.c`: simple implementation of a linked-list queue that you are free to use as needed.
- Tests designed to help you work through and verify each step.
- A Makefile where you can tweak compile options if desired.

Notes/hints

- Some of the function specifications mention "undefined behavior." Undefined behavior means that a conforming implementation is permitted to do *anything* in that case, including misbehaving, crashing, deleting system32, or microwaving your poodle. Practically speaking, it means you may write your code under the assumption that the given scenario will never happen.
- Make sure you always understand why you get a warning, or what you are doing to stop the warning. I would encourage turning on `-Werror` at some point. You can selectively un-error certain warnings with `-Wno-error=`*warning-name*.
- Make checking return values a matter of habit. Write the error check the first time you write the code. Use `perror()` liberally. If it makes sense to do so, propagate the error upward to the program which called your function by setting `errno` and returning nonzero. For internal errors or "should not happen" conditions, which likely indicate a bug in your implementation, feel free to use `assert()` and `abort()`.
- Please use the supplied `DPRINTF` macro for any debug printing. It should behave exactly like `printf()`. It will not only ensure that debugging information is written to the unbuffered stderr stream, but also allow both you and me to turn off debug printing for the purposes of testing/grading.
- Get to know the four ucontext functions (`man ucontext.h`) and what features they have. This will save you from wasting time reinventing the provided functionality.
- If you would like to use Valgrind to help catch problems early, you need to inform it of any memory that will be used as a stack; otherwise, Valgrind will get a bit confused when you switch context. I've added Valgrind's header file to this repository, and the fix is very straightforward:

  ```
  #include "valgrind.h"
  // Whenever you allocate stack memory...
  stackmem = malloc(len);
  // ... add the following line to register it:
  VALGRIND_STACK_REGISTER(stackmem, stackmem + len);
  ```

  (You can see another example in `test-create.c`, in that case for statically-allocated stack memory.)

Steps/tests

Each step comes with a test program that should output "success!" when it is done. Additional tests or updates to the existing ones will likely be added later.

You can run all of the tests in order by using the `test` target from the Makefile

(i.e. `make test`). The `vtest` target will run the same tests in Valgrind. Each of these targets will stop at the first test which fails.

Points for each step are given, with a total of 80.

test-create (10pts)

Implement `kfc_create` simplistically to start, so that it runs the thread to completion before returning. However, instead of calling the thread main function directly, use the ucontext functions to run it in its own context. When the thread context returns, execution should resume from where the calling context left off in `kfc_create`. (This should not require a lot of code once you understand ucontext.) Allocate space for the stack if none is provided, and pass the provided argument on to the thread function. Note: yes, it will be necessary to type-cast a function pointer here, but don't make a habit of it!

When the test succeeds, it will print the text `success!`. If you do not see this message, then the process exited prior to successfully completing the test.

Note: You will only be doing cooperative, many-to-one multithreading to start with (i.e. `kfc_init(1, 0)`), so you do not need to implement support for additional kernel threads until later, and you will only need to consider preemption if you do the bonus challenge.

test-self (10pts)

Update `kfc_create` so that each thread is assigned an integer thread ID upon creation, and make sure this is returned via the `ptid` parameter. Implement `kfc_self` so that it returns the thread ID of the currently executing thread. The main thread gets ID 0. (Hint: keep track of the currently executing thread! Be sure it gets updated when switching context or when a thread exits.)

test-fcfs (10pts)

Update your code so that, instead of returning to the "parent" when a thread exits, the next thread to run is chosen by an FCFS policy. A small queue implementation is provided for you to use if needed. (Note that the queue code is not synchronized internally, so you will need to treat it as "shared data" and synchronize accesses yourself once there are multiple kernel threads.) When `kfc_create` is called, the newly created thread should still execute first.

Hint: it may help to create an extra context which is not specific to any one thread.

Note: implementing this step will change the behavior of test-create and test-self. This is expected. (Reason, for the curious: when the main thread returns, the process terminates and any remaining threads do not have a chance to complete.) They will be fixed in the next step.

test-yield (10pts)

Implement the `kfc_yield` function to enable cooperative multiprogramming. When a thread calls `kfc_yield`, your code should pass control to the ready thread which has been waiting the longest since becoming ready (i.e., FCFS). If no other thread is ready, control will return to the caller. If this does not end up being a particularly simple function, this would be a good time to think about refactoring or asking for a hint on simplifying your approach.

test-yield2 (5pts) Speaking of simplifying code, update the `kfc_create` function so that new threads are not immediately activated but take their rightful place at the back of the line.

test-exit (5pts)

Implement the `kfc_exit` function, which should terminate the calling thread and schedule the next thread in FCFS order. Returning from the thread main function should be equivalent to calling `kfc_exit` and passing it the value returned by the function. Again, this will end up being a simple function if your design is in good shape; if not, ask for some direction.

If `kfc_exit` is called from the process's initial main thread, the other threads should continue to run until the process exits explicitly with a call to `exit`, which the OS will handle for you. (This is the same way that `pthread_exit` would behave for kernel threads, plus it's the easiest approach to implement.)

test-join (5pts)

Implement the `kfc_join` function, which retrieves the value returned by the specified thread when it exited and cleans up the thread's resources, and *blocks* if that thread is still running. Remember that a thread or process which is blocking should not be place in the ready queue until the event it is waiting for has taken place. In other words, calling `kfc_yield` in a `while` loop is not blocking. (Hint: you will need to modify `kfc_exit` to do this correctly.)

test-sem* (15pts)

Implement semaphores. These should behave like the implementation given in the textbook: if the thread needs to wait, it should block as with `kfc_join`. Note that this should block only the *user* thread, not the kernel thread!

test-m2m (5pts)

Time to start using that first parameter to `kfc_init`. Add support for multiple kernel threads (many-to-many threading model) to the KFC library. Up to this point, you didn't need the synchronization primitives provided by the kthread library, since there was only one kthread. Now you will need to synchronize access to any data that could potentially be accessed by more than one kernel

thread. Other data may need to become "per-kthread" so that there is a separate value/copy for each.

For the simplest implementation, you will want to use the following approach:

- Symmetric multiprocessing (SMP): each kthread should run the context switching code for itself.
- No kthread affinity: all kthreads share a single ready queue.
- Something which feels like "pull migration": each kthread schedules a user thread for itself whenever it becomes free and there are threads ready. (This is a corollary of the previous two, and not technically migration since there are not multiple ready queues, but it may be helpful to note the similarity nevertheless.)

test-m2m-pc (5pts) Here's a bounded buffer implementation using KFC. It should hit 100000 different checkpoints. You can check that all the lines are unique with:

```
$ ./test-m2m-pc |& sort -u | wc -l
100001
```

(100001 lines: 100000 checkpoints plus the success message)

Bonus challenge

test-preempt

Time to start using that second parameter to `kfc_init`. Add preemption to turn your library into a round-robin scheduler instead of FCFS. This step will need to be done on Linux due to the need for some newer POSIX.1-2008 functionality. You will need a little help from the OS in order to preempt a thread. The `timer_create` function provides the ability to send a signal to a specific (k)thread after a given time interval elapses, using `SIGEV_THREAD_ID`. I put a lot of effort into ensuring that the return value of `kthread_self` is exactly what you need to use as a thread ID when specifying the target thread.

If you want help digging through the related man pages, feel free to ask.

(Alternatives would be to use the `ualarm` or `setitimer` functions, which allow you to deliver a signal to a *process*. However, as we saw in class, it is not always defined as to which specific *thread* receives a signal that was sent to a multithreaded process.)

Due date and extra credit opportunity

This project can be challenging but is very rewarding! In order to make sure there is ample time to complete it, the project will be due April 7. In order to encourage an early start, I will give 3pts extra credit to anyone who can show me that they are passing the test-create step by this Wednesday, March 10.