Assignment 2: Round-Robin Scheduling

Due Date: Friday, October 24th

Overview

In the last assignment, we developed a mechanism for context-switching between two running threads. In order for our system to be useful, however, we'd like to extend it to deal with an arbitrary number of threads that can be dynamically created (and destroyed) during program execution.

To facilitate this, we will develop a round-robin scheduler: when a thread calls `yield()`, it will put itself at the end of a queue of waiting threads and pull the next thread off that queue.

Crucial to this will be the notion of thread state. We will extend our thread table entry data structure to include a value of an enumeration that the scheduler can use to make decisions about what to do with a thread.

Recall that in the last assignment, we ran into trouble if one thread terminated before the other: if the main thread finished first, the program would end prematurely. Why? Because the main thread had no way of knowing that the other thread was not finished. We can fix this by placing an infinite loop at the end of our main thread that continues until all other running threads are finished.

So we know that we need at least two thread states, "done" and "not done". "Not done" is not very descriptive, however. Threads that are not done may either be running, or they may be suspended and ready to run. Threads may also be suspended and not ready to run, a.k.a. blocked. Thus, we can define an enumeration of thread states:

```
typedef enum {
  RUNNING,
  READY,
  BLOCKED,
  DONE
} state_t;
```

Now also recall that if the spawned thread finished first, the program would crash. Why? Because when a function returns, it pulls the return address of the calling function from the control stack and jumps to it. But our thread *had no calling function*; the initial method began at the bottom (the highest address) of the control stack. Thus, the CPU simply grabbed some garbage from the next address in the heap and tried to jump to it, causing the program to crash.

How should we fix this? One way is to push a finish handler at the bottom of the stack which sets the current thread's state to `DONE`, then calls `yield()`; we can then organize our scheduler in such a way as to never context-switch back to finished threads. Note that this would be impossible without maintaining thread state.

Instructions

1. Make a new directory for your second assignment. Copy your assembly code files from assignment

1 into the new directory. To save you time and let you focus on the important aspects of this assignment, I've written a simple queue ADT for you to use, which has an enqueue and dequeue function, and an `is_empty` predicate. Additionally, the dequeue function returns `NULL` if the queue is empty. Add the files queue.h and queue.c to your folder. It may be convenient to do this with the `wget` command in bash, e.g.:

```
$ wget http://cs.pdx.edu/~kstew2/cs533/project/assign2/queue.h
$ wget http://cs.pdx.edu/~kstew2/cs533/project/assign2/queue.c
```

2. Start a new file called `scheduler.h`, and copy in your definition for `struct thread_t` from the first assignment. Add the `state_t` enum above, and add a `state` field to `struct thread_t`.

   We'd also like to be able to pass parameters to the initial function of a thread. Add a new field, `void * initial_arg`, and change the type signature of the initial function to `void(*initial_func)(void*)`. The polymorphic `void*` argument type will allow our functions to take arbitrary parameters. This is a common pattern in C. For more on this, see this page.

   In this assignment, we'll be writing a more complete scheduling API, so add the following prototypes to `scheduler.h`:

```
void scheduler_begin();
void scheduler_end();
void thread_fork(void(*target)(void*), void * arg);
void thread_finish();
void yield();
```

   The idea here is that we want to be able to write very clear, modular programs, such as this:

```
#include "scheduler.h"


void foo(void * arg) {
  char * str = (char*) arg;
  /* do important stuff */
}


int main(void) {
  scheduler_begin();
  thread_fork(foo, (void*)"bar");
  thread_fork(foo, (void*)"baz");
  scheduler_end();
}
```

   We'll go through ideas on how to implement each of these API functions in turn.

3. Start a new file, `scheduler.c`, which will contain the definitions of our scheduler API functions. Make sure you `#include` both `queue.h` and `scheduler.h`.

   Additionally, start a new file, `main.c`, which will include any test code you write. If everything is set up properly, this should only need to `#include` the file `scheduler.h`.

   When compiling, make sure you link your test code with `scheduler.c`, `queue.c`, and `switch.s`.

4. First, we'll need to make some slight modifications to our `thread_start` assembly routine. Above,

in the overview section, we discussed how we might solve the problem of allowing a thread to terminate gracefully.

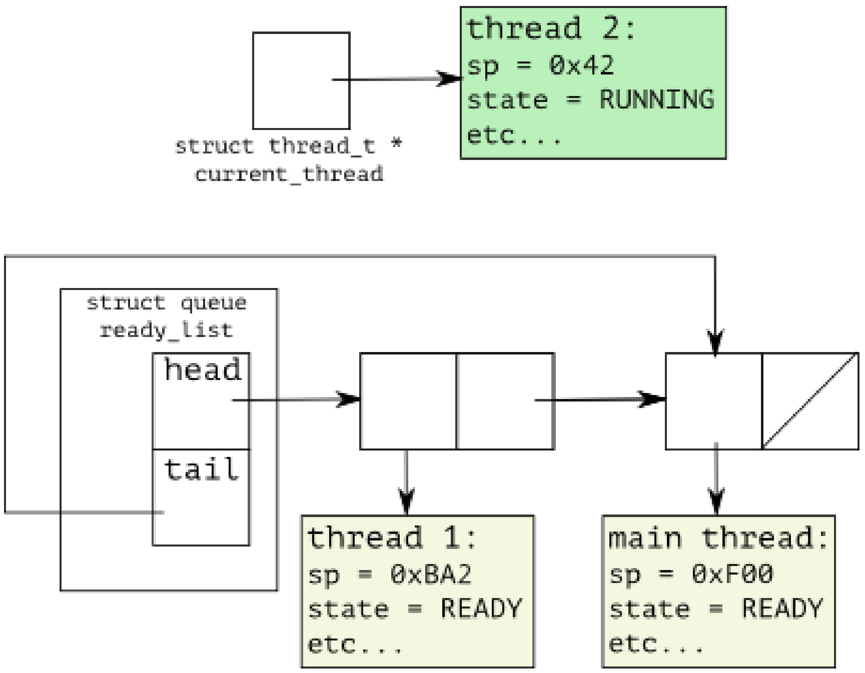
Implement the `thread_finish` function accordingly, then modify your `thread_start` routine to push the address of `thread_finish` on the bottom of the new thread's stack. This can be accomplished with the instruction:

```
pushq $thread_finish
```

We'll also want to pass our initial argument to the target function. Once you're done with `%rdi`, you can overwrite it with the value of the initial argument.

5. What should `scheduler_begin` do? Other than providing a nice dual to `scheduler_end` to make the code look nice, it should initialize and/or allocate any data structures our scheduler will need.

   Here's an idea of how you might lay out your scheduler, using the provided queue ADT:



6. Next, let's implement `thread_fork`. This function encapsulates everything necessary to allocate a new thread and then jump to it. `thread_fork` should:
   1. Allocate a new thread table entry, and allocate its control stack.
   2. Set the new thread's initial argument and initial function.
   3. Set the current thread's state to `READY` and enqueue it on the ready list.
   4. Set the new thread's state to `RUNNING`.
   5. Save a pointer to the current thread in a temporary variable, then set the current thread to the new thread.
   6. Call `thread_start` with the old current thread as `old` and the new current thread as `new`.

7. `yield` is very similar to `thread_fork`, with the main difference being that it is pulling the next thread to run off of the ready list instead of creating it. `yield` should:
   1. If the current thread is not `DONE`, set its state to `READY` and enqueue it on the ready list.
   2. Dequeue the next thread from the ready list and set its state to `RUNNING`.

   3. Save a pointer to the current thread in a temporary variable, then set the current thread to the next thread.
   4. Call `thread_switch` with the old current thread as `old` and the new current thread as `new`.

8. Finally, recall that we need a way to prevent the main thread from terminating prematurely if there are other threads still running. A potential solution is given in the overview above. Implement this solution how you see fit in `scheduler_end`. You might find the `is_empty` queue predicate useful.

9. Test your code! Have some fun with it. You can do whatever you like as long as it is a thorough test. One suggestion is to code some functions that perform some long-running computations, such as finding the nth prime number for large values of n, or sorting large arrays with slow algorithms like selection sort. Fork off several threads (at least four) to run these functions, paramaterized by different values. Verify that small instances finish first, even if their `thread_fork` came later in `main`. Of course, make sure you insert `yield()`s in the core loop of your computation to cause interleaving!

10. Conspicuously absent from this assignment has been any discussion of memory management. Think about when and how to free the memory associated with a thread, including its control stack and thread table entry. How does this change if we want to enable inter-thread control flow, e.g. joins or alerts? Put your thoughts in the write-up.

What To Hand In

If you are working with a partner, please prepare just one submission for the both of you. Make sure your (and your partner's) names are included in each file you submit.

You should submit:

1. All code, including tests. This should include:
   - `switch.s`
   - `queue.h`
   - `queue.c`
   - `scheduler.h`
   - `scheduler.c`
   - `main.c`
2. A brief written report, including:
   1. A description of what you did and how you tested it.
   2. Your thoughts on the question posed in section 10, above.

Please submit your code files *as-is*; do not copy them into a Word document or PDF.
Plain text is also preferred for your write-up.
You may wrap your files in an archive such as a .tar file.

Email your submission to the TA at kstew2 at cs.pdx.edu on or before the due date. The subject line should be "CS533 Assignment 2".

Need Help?

If you have any questions or concerns, or want clarification, feel free to contact the TA by coming to office hours or sending an email.

You may also send an email to the class mailing list, but please pose discussion questions only; **do not** email any code to the class mailing list! (It is okay to send code directly to the TA, however).