Thread

Implement a user-level cooperative thread library, using the POSIX context functions. The necessary functions are specified in this header:

thread.h

```
struct thread;
struct thread *thread_create(void (*f)(void));
void thread_yield(void);
void thread_wait(struct thread *t);
void thread_destroy(struct thread *t);

struct semaphore;
struct semaphore *semaphore_create(unsigned int value);
void semaphore_post(struct semaphore *s);
void semaphore_wait(struct semaphore *s);
void semaphore_destroy(struct semaphore *s);
```

Define the contents of the `struct thread` and `struct semaphore` datatypes, as well as the bodies of the eight required functions, in `thread.c`. An implementation of the Producer–Consumer problem, using the thread library you will implement, will be provided in `main.c` and `queue.c`; build the whole thing into a program called `queue` with this Makefile:

Makefile

```
CFLAGS=-Wall -g

OBJECTS=\
  queue.o \
  main.o \
  thread.o

HEADERS=\
  queue.h \
  thread.h

queue: $(OBJECTS)
queue.o: queue.c $(HEADERS)
main.o: main.c $(HEADERS)
thread.o: thread.c $(HEADERS)

.PHONY: clean
clean:
	rm -f queue $(OBJECTS)
```

There have been some changes to the `queue` program compared to the previous project; specifically, instead of `sleep`ing, the threads call `thread_yield` on each outer iteration, and instead of stopping after a particular amount of time, they stop after a particular number of items have been processed through

(thus avoiding the problem of hanging at the end of the program if the speeds do not match). The options that specify the various parameters are shown in the usage information:

```
$ ./queue -?
Usage: ./queue [OPTIONS]

Options:
  -q SIZE  Queue at most SIZE items
  -p NUM   Produce NUM items at a time
  -c NUM   Consume NUM items at a time
  -n NUM   Stop after NUM items
```

Details

Threads are launched with `thread_create`, which expects a pointer to a function with no arguments or return value, and runs that function in its own thread. Schedule threads in a round-robin fashion (first-come first-served), and whenever new threads are added to the queue (by `thread_create` or on waking up from `semaphore_wait` or `thread_wait`), add them to the tail of the queue (to execute last). Move on to schedule the next thread in the queue on `thread_yield`, and whenever necessary on `semaphore_wait` or `thread_wait`. On `thread_yield`, just move the current thread to the tail of the queue; on `semaphore_wait` on a zero semaphore or `thread_wait` on an active thread, move it to a waiting queue associated with that semaphore or thread.

When the function associated with a thread returns, wake up (reschedule at the tail of the queue) all of the other threads waiting for it (with `thread_wait`). One of them (but not more than one) must call `thread_destroy` to clean up the thread. Note that every thread must be waited for and then destroyed, and it is perfectly valid to wait on a thread either while it is still running or after it has already finished (although in the latter case, there is no need to actually wait). Look at the provided `main` function and bounded queue implementation to see how the thread and semaphore functions you implement should be used.

Because there is no ambiguity about scheduling order or when the running thread changes, execution should always be deterministic—here are some examples of the output you should get with various parameters:

```
$ ./queue
Creating producer thread.
Creating consumer thread.
Waiting for producer thread.
Producing.
Produced 0.
Consuming.
Consumed 0.
Produced 1.
```

```
Consumed 1.
Produced 2.
Consumed 2.
Produced 3.
Consumed 3.
Produced 4.
Consumed 4.
Produced 5.
Consumed 5.
Produced 6.
Consumed 6.
Produced 7.
Consumed 7.
Produced 8.
Consumed 8.
Produced 9.
Consumed 9.
Done producing.
Done consuming.
Waiting for consumer thread.
Done.
```

```
$ ./queue -q 3 -p 2
Creating producer thread.
Creating consumer thread.
Waiting for producer thread.
Producing.
Produced 0.
Produced 1.
Consuming.
Consumed 0.
Produced 2.
Produced 3.
Consumed 1.
Produced 4.
Consumed 2.
Produced 5.
Consumed 3.
Produced 6.
Consumed 4.
Produced 7.
Consumed 5.
Produced 8.
Consumed 6.
Produced 9.
Consumed 7.
Done producing.
Consumed 8.
Waiting for consumer thread.
```

```
Consumed 9.
Done consuming.
Done.
```

Resources

Each thread will need a way to save its registers and stack when it is not running. POSIX specifies getcontext, setcontext, makecontext, and swapcontext for managing such contexts. In addition to the linked specifications, the Linux manual pages (e.g. `man swapcontext`) provide somewhat helpful examples.

When you have finished, submit your `thread.c` on CMS.