Machine Problem 3: Simple Thread Scheduling

This Handout is for Step 2 of MP3

Introduction

In Step 1 of this machine problem you implemented a simple task system that uses the system (or the pthread library) scheduler for scheduling.

In Step 2 we add scheduling support to the task system. For this, you derive a class `Schedulable` from class `Task`, which provides *schedulable threads*. The class is defined as follows:

```
class Schedulable : public Task {
protected:
  friend class Scheduler;
  Scheduler * sched;
  virtual void CarrierForRun();
  /* Get the task ready to run call the "Run()" method. For example,
     put the task into the scheduler ready queue before running, and
     give up the CPU after the "Run()" method returns. */
  /* -- SUSPEND EXECUTION */
  virtual int Block();
  virtual int Unblock();
 /* These two methods simulate the suspension and resumption of execution
     of the task. In a real system, the scheduler would move off the task
     from the CPU. We cannot do this here, so we force the task to suspend
     execution by having it "Block". The task resumes execution
     with "Unblock". */
public:
  /* -- CONSTRUCTOR/DESTRUCTOR */
  Schedulable(const char _name[], Scheduler * _sched);
  /* Create and initialize the new schedulable task. Make sure it knows
     about the system scheduler.
  ~Schedulable();
  /* -- TASK OPERATIONS */
  virtual int Start();
  /* Start the execution of the task. Calls "Scheduler::Start()" to hand
  over execution of the task to the scheduler to start execution of task.
     This method is typically called shortly after the task has been
     created. */
  virtual void Run() = NULL;
  /* The method that is executed when the task object is started.
     When the method returns, the thread can be terminated.
```

```
     The method returns 0 if no error. */
};
```

The execution of schedulables is controlled by an object of class `Scheduler`, which provides the following functionality:

```
class Scheduler  {
protected:
  Schedulable * current_task;
  /* The scheduler maintains a pointer to the currently running task. */
  /* -- READY QUEUE MANAGEMENT */
  list<Schedulable*> ready_queue;
  /* This is a simple example of a ready queue that could be used for
     a FIFO scheduler. */
  virtual int           enqueue(Schedulable * _task);
  virtual Schedulable * dequeue();
public:
  /* -- CONSTRUCTOR/DESTRUCTOR */
  Scheduler(char _name[]);
  ~Scheduler();
  /* -- THE CURRENTLY RUNNING TASK */
  Schedulable * Current_Task();
  /* Return a pointer to the currently running task. */
  /* -- SCHEDULING OPERATIONS */
  virtual int Start(Schedulable * _task);
  /* Start scheduling this task. This method is called in method
     Schedulable::Start() that starts execution of the task, more
     specifically, inside method Schedulable::CarrierForRun().
     If return value is not zero, task could not be successfully
     started. */
  virtual int Kick_Off();
  /* This method starts the scheduled execution of the tasks. It
  is called after all the tasks have been started and are waiting
  in the ready queue. */
  virtual int Yield();
  /* The calling task gives up the CPU. */
  virtual int Resume(Schedulable * _task);
  /* Put the given _task on the ready queue. */
};
```

Tasks are queued for execution on the `ready_queue` of the scheduler. Appropriate synchronization mechanisms must be in place to ensure that a task gets blocked (i.e. "give up the CPU") upon calling `Yield()` and that the correct task gets released. In a real system the yielding task would naturally give up the CPU and therefore suspend its execution. In this assignment we simulate this by having the task block itself (using the method `Block()`) inside the `Yield()` method. The next task in the ready queue then released from suspension by having `Yield()` invoke its method `Unblock()`.

Tasks "wake up" (either from I/O or otherwise) trough method `Resume()`, which puts the task on the ready queue. Note that `Resume()` does not trigger the task to continue executing. This is done inside the `Yield()` method.

The method `Scheduler::Start()` enqueues the task on the ready queue and suspends it (using the method `Block()`). Tasks start actually executing when the main program kicks off the scheduler, using method `Scheduler::Kick_Off()`.

A slight complication arises when you want to start the task in `Schedulable::Start()`: You don't want to enqueue the task in `Schedulable::Start()`, because you would be enqueuing the parent task instead of the newly-generated task. On the other hand, you don't want to add the `Start()` functionality inside the `Run()` method, because that method contains "functional" code only, and the "scheduling" code should be all be kept inside class `Schedulable`. This is where the method `Schedulable::CarrierForRun()` comes in, which contains all the code that interacts with the scheduler when the task starts and when it finishes. (**Note:** Unfortunately, the method `CarrierForRun()` must be available inside of class `Task` as well; see the handed-out source code. The implementation of class `Task` is affected in a minor way only; you need to replace the invocation of `Run()` inside `tfunc` with an invocation of `CarrierForRun()`, which sets up the task at startup time before calling `Run()`.)

A Program then could look as follows:

```
class AvionicsTask : public Schedulable {
public:
  AvionicsTask(char _name[], Scheduler * _sched):Schedulable(_name,
                                    _sched) {}
  void Run() {
    cout << "Avionics System [" << name << "] running\n" << flush;
    for (int i = 0; i < 100; i++) {
      cout << name << " waiting for next refresh interval\n" <<
                                                           flush;
      if (i % 10 == 1) {
          sched->Resume(this); /* preemption */
          sched->Yield();
      }
      cout << name << " refreshing avionics screen\n" << flush;
```

```
      usleep(12000);
    }
  }
};
int main(int argc, char * argv[]) {
  /* -- CREATE SCHEDULER */
  Scheduler * system_scheduler = new Scheduler("scheduler");
  /* -- CREATE TASKS */
  RudderController task1("rudder control", system_scheduler);
  AvionicsTask     task2("avionics task", system_scheduler);
  /* -- LAUNCH TASKS */
  task1.Start();
  task2.Start();
  usleep(100000); /* UGLY!! */
  /* -- START SCHEDULING */
  system_scheduler->Kick_Off();
  /* -- Have the parent thread get out of the way. */
  Task::GracefullyExitMainThread();
}
```

A little Wrench in the Wheel: You will need appropriate synchronization to (a) ensure mutual exclusion in some critical sections (e.g. operations on the ready queue, etc.) and inside the methods `Block()` and `Unblock()`. For this you have to implement a class `Semaphore`, which has to be implemented using POSIX mutex locks and POSIX condition variables. The interface of class `Semaphore` is as follows:

```
class Semaphore {
private:
  int             value;
  pthread_mutex_t m;
  pthread_cond_t  c;
public:
  /* -- CONSTRUCTOR/DESTRUCTOR */
  Semaphore(int _val);
  ~Semaphore();
  /* -- SEMAPHORE OPERATIONS */
  int P();
  int V();
};
```

The Assignment (Step 2)

You are to implement the following classes:

- class `Semaphore`: submit a file named `semaphore.C` that implements the class defined in *semaphore.H.*
- class `Schedulable`: submit a file named `schedulable.C` that implements the class defined in *schedulable.H.*
- class `Scheduler`: submit a file named `scheduler.C` that implements a non-preemptive FIFO scheduler, with the interface defined in file `scheduler.H`. Pay attention that the implementation of the scheduler can be easily modified (for example by deriving a class `FooBarScheduler`) to other scheduling algorithm. (In Step 3 we will investigate *preemptive* schedulers.)

Turn in these three files in addition to a design document that describes how you approached the implementation and how you are addressing re-use of your code to take into consideration other scheduling algorithms. **The files that you provide (including Makefile) should compile. If what you provide will not compile, you will receive 0 for this assignment. Note: if you change any header files that were provided to you, it is best to turn these in as well.**