The Sleeping Barber problem is attributed to Djikstra, and is based on the following scenario:

A barbershop contains a barber, the barber's chair, and N chairs for waiting customers. When there are no customers, the barber sits in his chair and sleeps. When a customer arrives, (a) if the barber is sleeping, he awakens the barber and sits down for a haircut; (b) if the barber is busy and there is a free chair, he sits down and waits; (c) if the barber is busy and there are no free chairs, he leaves immediately.

For this problem set we will model the behavior of this system as concurrent threads of execution – one for the barber, and one for each of the customers.

**Question 1 – synchronization**

Provide pseudo-code for a monitor object which has two methods, barber() and customer(), modeling a barbershop with 4 waiting chairs and one barber's chair:

- barber() does not return, but instead loops forever sleeping and cutting hair
- a thread entering the customer() method corresponds to arrival of a customer at the shop; the thread returns from customer() when the customer would leave. (i.e after getting a haircut, or immediately if the shop is full)

Other than the line “cut hair” in the barber method, your solution should give precise pseudo-code specifying all of the operations which determine correct thread-safe behavior.

The precise definition of a monitor to be used in this exercise is given on the last page.
**Question 2 – POSIX Threads**

There is a straightforward translation from monitor pseudo-code to POSIX thread primitives:

1. Create a per-object mutex, m, of type pthread_mutex_t which is locked on entry to each method and unlocked on exit. (be careful when using multiple exits)
2. Condition variables translate directly to objects of type pthread_cond_t; C.signal() and C.broadcast() become pthread_cond_signal(C) and pthread_cond_broadcast(C);
3. The monitor mutex must be passed to wait calls; thus C.wait() becomes pthread_cond_wait(C,m).

For this exercise we will create a singleton monitor, using global variables instead of object variables, and functions rather than object methods.

Using the attached skeleton file q2.c, which may be compiled with the script make-q2, create a simulation of the barbershop using your answer to 1b, with \( N = 4 \) chairs. Your simulation should contain the following threads:

- 10 customer threads, each of which loops forever alternately trying to get a haircut and then sleeping for a random time \( \text{exponential}(0.1) \) before visiting the shop again. (note that customers who leave immediately must sleep before trying again)
- 1 barber thread, which simulates performing a haircut by sleeping for a random time \( \text{exponential}(0.8) \)

To sleep, use the function \( \text{mon\_sleep(double t, mutex m)} \) provided in q2.c. When called from a monitor method you should pass in the monitor mutex (so it can release it while sleeping), and when called from outside the monitor (i.e. when the customers are sleeping before entering the store) you should pass NULL for the mutex.

The barber and customer methods should print out indications at each key transition – when a customer enters the shop, finishes waiting or exits immediately, and finishes getting a haircut, and when the barber starts and finishes a haircut, and goes to sleep. At each point you should print out the time, returned as a floating point number by the function \( \text{mon\_time()} \).

When debugging your code you may find it useful to run it at a faster speed; this may be done by invoking the program as \( ./q2 -speedup \# \), which will speed up execution by \( \# \) times.
(e.g. \( ./q2 -speedup 5 \))
**Question 3 – Discrete Event Simulation**

Copy q2.c to q3.c, and change `#include <pthread.h>` to `#include “simthread.h”`. You should now be able to compile this file with the script make-q3.sh and run it as a discrete-event simulation.

You will need to simulate the barbershop with the same number of customers and delay distributions as above. In addition to submitting the source code to your simulation, answer the following:

a) What fraction of customer visits result in turning away due to a full shop?

b) What is the average time spent in the shop (including haircut) by a customer who does not find a full shop?

c) What fraction of the time is spent in the state with:
   0 customers in the shop?
   1 customer?

d) 2, 3, 4, and 5 customers, respectively?

Note that there are some sample statistics functions in stats.c which may be of use.

**Question 4 – Markov Model**

The barbershop with 4 seats (plus the barber's chair), a total of 10 potential customers, and exponentially distributed wait and haircut times can be modeled as a Markov process with 6 states corresponding to the number of customers in the shop. Do it.

Note that since we have a finite number of customers with non-infinitesimal arrival rates, the rate at which customers arrive at the door is affected by the number waiting in the shop.

a) What is the steady state probability of each state?

b) What is the probability of a customer turning away due to a full shop?

c) What is the expected time in the shop (including haircut) if the shop is not full?

You may find it useful to compare your answers to this question with those from Question 4 – typically any discrepancies are due to flaws in your code for Questions 2 and 3. This may be a good time to go back and fix them.
Definitions:

Semaphore: A semaphore is an object with an initial value N and two methods: down() and up(). Calls to down() can block, while calls to up() do not block. Note that you don't implement semaphores - you instantiate them:

\[ S = \text{semaphore}(N) \text{ -- create a semaphore with initial value } N \]

Semaphore behavior is defined thusly:
Given a semaphore S with initial value N: at any point in time, if M threads have called S.up(), then no more than M+N threads have returned from S.down().

Monitor: A monitor is a particular kind of user-defined class, which has:
- regular variables
- methods
- conditions --- this is what normal classes don't have

Threads can:
- enter the monitor by calling any of the methods
- leave the monitor by returning from a method
- leave the monitor by calling wait()
- re-enter the monitor by returning from wait()

If there are any threads waiting on condition C:
- C.signal() will release one of them and
- C.broadcast() will release all of them.
- If there are no threads waiting, neither C.signal() nor C.broadcast() have any effect.

Only one thread can be in the monitor (i.e. entered but not left) at any given time. If thread A is in the monitor and thread B tries to (a) call a method or (b) return from wait(), thread B will block until thread A leaves the monitor. (or longer – see below)

When a thread calls signal() or broadcast(), it does not leave the monitor, so that you can be sure that it will continue to run before any thread that it woke up.

However, if you wake thread A via signal() or broadcast(), and thread B tries to enter the monitor by calling a method, you don't know whether A or B will get to run first.