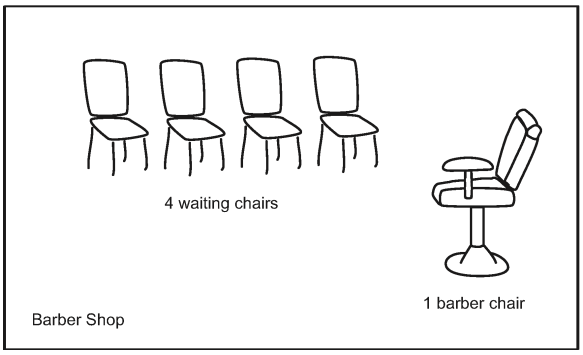
CS 5600 – Intensive Computer Systems
Homework 2 – The Sleeping Barber Problem

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, the customer awakens the barber and sits down for a haircut; (b) if the barber is busy and there is a free chair, the customer sits down and waits; (c) if the barber is busy and there are no free chairs, the customer leaves immediately.



More specifically, there are 10 customers and 1 barber, each represented by a thread, and a monitor representing the barbershop with two methods, `barber()` and `customer()`.

- The barber thread calls `barber()`, which never returns. (i.e. the barber never leaves the shop) The barber sleeps until woken by a customer, and then gives haircuts (average time 1.2 seconds, exponentially distributed) until there are no customers left, after which he goes to sleep again.
- Each customer thread loops, alternately sleeping for a period of time (average 10 seconds, exponentially distributed) and then calling `customer()`, which returns after the customer leaves the shop. (immediately if all chairs are full, or after getting a haircut.)

Question 1 – synchronization

Part 1: Provide a design for a monitor which has two methods, barber() and customer(), modeling a barbershop with 4 waiting chairs and one barber's chair, with 10 customers as described above.

Your design should identify all condition variables and other instance variables, and describe the operation and logic of each method. It is **not** code – please leave out the curly braces, and focus on describing *what your code is going to do*, not what your code *is*. Submit this as either a text file (q1-part1.txt) or as part of a PDF file with Part 2.

Remember the following characteristics of the monitor definition used in this class:

- Only one thread can be in the monitor at a time; threads enter the monitor at the beginning of a method or when returning from wait(), and leave the monitor by returning from a method or entering wait().
- This means there is no preemption – a thread in a method executes without interruption until it returns or waits.
- When thread A calls signal() to release thread B from waiting on a condition variable, you don't know whether B will run before or after some thread C that tries to enter the method at the same time. (that's why they're called *race* conditions)
- You don't need a separate mutex – this is a monitor, not a pthreads translation of a monitor.

Part 2: Illustrate the operation of this monitor using the graphical notation introduced in class.

This should be submitted as a PDF file; it doesn't matter whether it is hand-drawn and scanned or photographed (as long as it is readable) or prepared nicely with a drawing program.

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) Actually, since there's only one object – the barbershop – there should only be one mutex.
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.

You will use a single file, *homework.c*, for both question 2 and 3, using conditional compilation to separate the code for the two; code for question 2 will be compiled with the script *compile-q2.sh* creating an executable named *homework-q2.* The startup code for this question will go in the function *q2()*, and will do the following:

- Initialize the monitor objects. Note that mutexes and condition variables may be initialized either statically or dynamically:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER; /* static init */
pthread_cond_t C = PTHREAD_COND_INITIALIZER;

pthread_mutex_t m;                              /* dynamic */
pthread_mutex_init(&m, NULL);                   /* NULL = default params */

pthread_cond_t C;
pthread_cond_init(&C, NULL);
```

- Create N=10 customer threads; each thread will loop doing the following:
  … sleep for random(mean T seconds)…
  customer()

You are provided with a sleep function, *sleep_exp(T)*, where T is a floating point number giving the mean sleep time in seconds. Each thread will need to know its thread number, from 1 to 10; there is a comment in the code describing how to pass this value when starting a thread.

- Call *wait_until_done(),* which will sleep until a command-line-provided timeout or until the user types ^C.

Running question 2: The *homework-q2* command is used as follows:

```
./homework-q2 [-speedup #f] [#t]
```

where #t is a total number of seconds the homework should run for, and #f is a speedup factor. (e.g. *./homework-q2 100* would run for 100 seconds; *./homework-q2 -speedup 2.5 100* would do the same amount of work, but run 2.5 times faster, completing 100 simulated seconds in 40 real seconds.)

To use the debug script provided for this question, you will need to print the following lines as your code executes:

```
DEBUG: TTT customer # enters shop
DEBUG: TTT customer # starts haircut
DEBUG: TTT customer # leaves shop
DEBUG: TTT barber wakes up
DEBUG: TTT barber goes to sleep
```

where "TTT" is a floating point number returned by the *timestamp()* function.

Note that you must use this exact format for your debug messages or else the verification script (which is also used for grading) will not work.

Debugging: Assertions are very helpful in debugging this sort of code, as they allow you to verify program invariants – e.g. if you put the statement

```
assert(x == 1);
```

in your code, and then run it in the debugger, it will break into the debugger at that point if $x \neq 1$. Note that some program invariants are to check with a single assert – e.g. the barber is either asleep or cutting hair – while others (e.g. all customers are either growing hair, waiting in the shop, or getting a haircut) may require a bit more code, such as a loop.

If you redirect the output of the command into a file, you should be able to use the *verify-q2* script to determine whether or not your implementation obeyed all the rules:

```
./homework-q2 … > q2.out
./verify-q2 q2.out
```

Note that if you run your program for a short period of time, it is less likely to break any rules even if it is incorrect.

I suggest running Q2 for a while with a high speedup, and possibly performing some work in another terminal window on the same machine (to disturb the thread scheduling order) in order to determine whether it operates correctly or deadlocks.

Question 3 – Discrete Event Simulation

For this question you will compile your code using the *compile-q3* script, which will build it with a framework for discrete-event simulation. (this framework is found in *misc.c*, and uses the GNU Pth library) What this means is that your code will run in *simulated time* – basically the thread library "skips" time forward whenever threads are sleeping, and stops the clock when a thread is running. For simulations of small, slow systems (like ours) this results in a simulation running much faster than real time; for fast, complex systems (e.g. simulating operation of an integrated circuit) the simulation might run thousands of times slower than real time.

The simulation library is designed so that it is compatible with Pthreads operations, so you should be able to compile and run the same monitor code you used in Question 2. In addition several functions have been provided for gathering statistics:

```
void *counter = stat_counter();
stat_count_incr(counter);
stat_count_decr(counter);
double val = stat_count_mean(counter);

void *timer = stat_timer();
stat_timer_start(timer);
stat_timer_stop(timer);
double val = stat_timer_mean(timer)
```

A counter tracks an integer variable (e.g. the number of customers waiting in the shop) and provides its average value over time. A timer tracks the interval between a single thread calling *start()* and *stop()*, and provides the average value of these measured intervals.

Run your code for at least 10000 seconds of simulated time and report average values for the following measurements:

1. fraction of customer visits result in turning away due to a full shop. You don't need the stats counters for this – just count total customers (T) and customers turned away (A) and calculate T*1.0/A. (the '*1.0' is there to force C to use floating-point division.)
2. average time spent in the shop (including haircut) by a customer who does not find a full shop. Use a single stat_timer for this one – each thread starts it when it enters the shop and stops it when their haircut is done.
3. average number of customers in the shop (including anyone sitting in the barber's chair). Keep a stat_counter, each customer increments it when they enter and decrements it when they leave.
4. fraction of time someone is sitting in the barber's chair. Note that this is the same as the average number of customers sitting in the chair, so keep a stats counter that each thread increments when they sit in the chair and decrement when they leave.

Your results should be approximately the following:

1. between 0.07 and 0.08
2. between 3.07 and 3.10
3. between 2.33 and 2.40
4. about 0.85

If your numbers are significantly different from this, especially after running for 100,000 simulated seconds, then it's likely that you have a bug in your code and that it is not simulating the barber shop correctly.