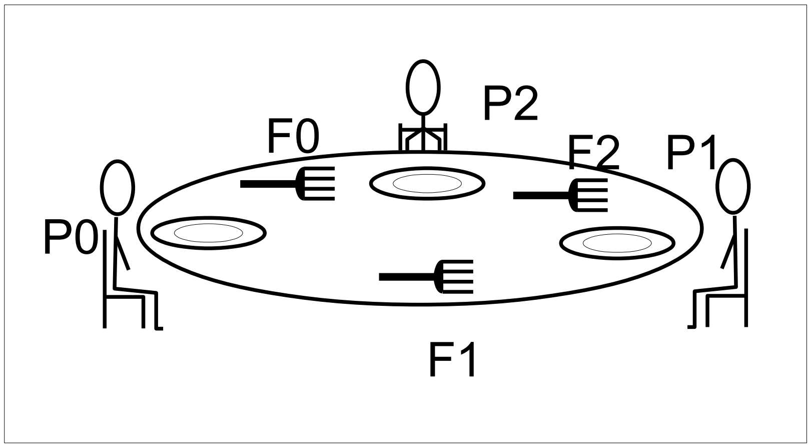
CS 5600 – Intensive Computer Systems
Homework 2 – The Dining Philosophers Problem

Assigned: October 11, 2012
Due: October **26**, 2012

The Dining Philosophers Problem is a classic synchronization problem, and is based on the following scenario:

> A group of philosophers sit at a round table with plates of food in front of each of them, and one fork (or in other versions, chopstick) between each plate, shared between the philosophers on each side, who aren't very fussy about sharing utensils. Each philosopher alternates between thinking and eating, but must pick up both forks to eat; after eating for a while he puts down both forks and goes back to thinking.



More specifically, we have philosophers 0..N-1, and forks 0..N-1. Philosopher number *i* does the following:

- think for some time (random value with mean T_{think} **= 5 seconds**)
- pick up left fork (fork *i*), waiting until it is put down by the philosopher on his/her left if necessary.
- pick up right fork (fork (*i*+1), except for philosopher N-1, who picks up fork 0), waiting if necessary.
- eat for some time (random with mean T_{eat} **= 2.5 seconds**)
- put both forks down, and repeat.

We model this as a monitor with two methods, get_forks() and release_forks(), and N philosopher threads which loop executing the following code:

```
while true:
  sleep(random(T_think))
  get_forks(i)
  sleep(random(T_eat))
  release_forks(i)
```

where *i* is the number of the philosopher corresponding to that thread.

Note – this problem violates the lock ranking guideline for avoiding deadlocks, as there is no total order in which resources (forks) are acquired. However it is free from deadlock as long as (a) each philosopher picks up their left-hand fork first, (b) if both forks are available, they are picked up atomically, and (c) philosophers always pick up forks in the order they tried to get them in.

Question 1 – synchronization

Part 1: Provide pseudo-code for a monitor which models the dining philosophers problem with 4 seats and 4 forks, providing the methods `get_forks(i)` and `release_forks(i)` as described above.

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.

Further submission instructions: Part 1 should provide almost as much detail as code – it should identify all instance variables and condition variables, and all operations affecting them. It may be submitted as a text file (q1-part1.txt) or as part of a PDF file including part 2. Part 2 should be submitted as a PDF file; it doesn't matter whether it is hand-drawn and scanned (as long as I can read it), photographed on your cell phone and converted (again, if readable) or prepared nicely with something like OpenOffice Draw. Drawings on paper, legibly labeled with your name, may be slipped under my door as well.

Question 2 – POSIX Threads

Since this is a singleton object (there's only one table of philosophers in our simulation) we use the following translation monitor pseudo-code to C and POSIX thread primitives:

1. Create a single global 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 global variables of type pthread_cond_t; signal(C) and broadcast(C) become pthread_cond_signal(C) and pthread_cond_broadcast(C).
3. Methods are implemented as functions, and normal instance variables are implemented as global variables so that they are accessible from all methods.
4. The monitor mutex must be passed to wait calls; thus wait(C) becomes pthread_cond_wait(C,m).

(note - If we needed multiple instances of the monitor, the object would be implemented as a C structure or C++ class, with *m*, conditions and other instance variables stored in structure fields, and methods being functions taking a pointer to an instance structure. In C++ it would be a class, with *m* and the conditions being instance variables)

You will use a single file, *homework.c*, for both question 2 and 3; the scripts provided use= 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);
```

- In the main thread, create N=4 philosopher threads calling `philosopher(i)`, where *i* is 0..3; this function will loop doing the following:

```
    sleep_exp(T_think)
    get_forks(i)
    sleep_exp(T_eat)
    release_forks(i)
```

  where sleep_exp(T) is a function provided in the homework which sleeps for an exponentially-distributed length of time with mean T seconds. (T is floating point)

  Each thread will need to know its thread number, from 0 to 3; 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.)

Testing: You are responsible for testing your code and making sure that it implements the required functionality correctly. The system state can be represented by 2N state variables, one for each philosopher and one for each fork. It's likely that your implementation will explicitly track the per-fork state; with another few lines of code you can track the per-philosopher state. You will need to:

- Document in your code (i.e. in comments) what state is being tracked.
- Again, in comments document the correctness properties that you are checking.
- Use the `assert()` function – e.g. `assert(x>0)` – to check these properties at appropriate points in your code.

`Assert()` will cause your program to print an error with the line number and exit, or if you are debugging it will trap into the debugger and let you examine what is wrong.

Note that just because your program *contains* a bug doesn't mean that you will *see* that bug in any individual run. I suggest running with a fairly high speedup (30 or 100) for several minutes to get better test coverage. (it also helps to perform other tasks on the computer while the test is running, in order to disturb the thread scheduling order)

Instrumentation: In addition to your own test code, you will need to print certain output that will be used to check for correctness when grading. In particular, you will need to print the following lines as your code executes:

```
DEBUG: TTT philosopher # tries for left fork
DEBUG: TTT philosopher # gets left fork
DEBUG: TTT philosopher # tries for right fork
DEBUG: TTT philosopher # gets right fork
DEBUG: TTT philosopher # puts down both forks
```

where "TTT" is a floating point timestamp returned by the *timestamp()* function and # is the philosopher number passed to the `get_forks()` and `release_forks()` methods.

You should insert this code early and keep it up to date, as there may be a chance to have your code tested against the grading checker before it is due for submission.

Question 3 – Discrete Event Simulation (3 philosophers)

For this question you will simulate 3 philosophers (not 4), and compile your code using the *compile-q3* script, which will build it with a framework (fromn *misc.c*, using the GNU Pth library) for discrete-event simulation. 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:

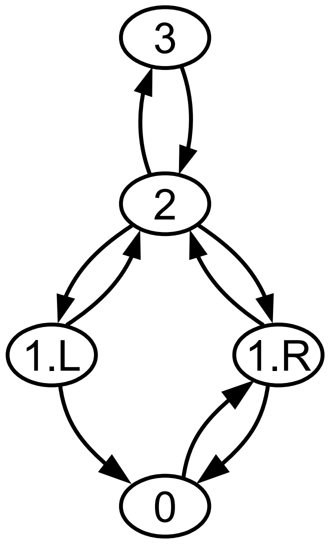
```
void *counter = stat_counter();
stat_count_incr(counter);
stat_count_decr(counter);
double val = stat_count_mean(counter);

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 philosophers eating) 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 measure average values for the following:

- The fraction of time there are 3, 2, 1, and 0 philosophers thinking.
- Time taken by a call to `get_forks()`.
- Number of philosophers waiting in the `get_forks()` method.

Question 4 – Markov Model (revised)



We can model the three-philosopher case as a Markov model, as shown to the left, with states identified by the number of philosophers thinking, as follows:

- 3 – three philosophers thinking
- 2 – two thinking, one eating
- 1.L – one thinking, one eating, one waiting to pick up left-hand fork
- 1.R – one thinking, one eating, one waiting to pick up right-hand fork
- 0 – none thinking, one eating, one waiting for left-hand fork and the other waiting for the right-hand fork.

Note that there are multiple configurations corresponding to some of these states – for instance in state 1.L we might have philosophers 0, 1, and 2 in states (eat, left, think), (think, eat, left), or (left, think, eat) respectively. (just like in the couch example, where the state with one person on the couch and one in the bathroom didn't distinguish which person was which)

Part 1: Label each edge in the graph with its transition rate. You can either draw out the graph or just list the transition rates. (e.g. 2 → 1.L = ***) Note that transitions occur because a philosopher finishes either thinking (with rate 1/5 per philosopher) or eating (with rate 1/2.5 per philosopher).

Part 2: Solve for the equilibrium state probabilities. For this you will need to set up a system of 5 equations in 5 variables, where the variables are the state probabilities, there are 4 (out of 5 possible) flow conservation equations, and a 5th equation saying that the state probabilities add up to 1. These equations are shown in this table:

	3	2	1.L	1.R	0		
3	*	*				=	0
2	*	*	*	*		=	0
1.L		*	*			=	0
1.R		*		*	*	=	0
	1	1	1	1	1	=	1

Each entry (*row, column*) in this table is the flow rate *into* state '*row*' from state '*column*'. (negative numbers for flow *out of* the state) Thus the entry for row 2, column 1.L is the flow rate from state 1.L into state 2, and the entry for 2,2 is the rate from 2 “into” 2 (i.e. a negative number which is the sum of all the rates *out* of 2).

To solve this we use Matlab (or Octave, a free Matlab clone available on the CCIS linux machines) and a bit of linear algebra. We note that the system of equations is equivalent to Ax=y where ***A*** is the matrix corresponding to the table on the left above, ***x*** is the vector of unknowns (P(3), P(2), …), and ***y*** is the rightmost column. We solve this by inverting the A matrix:

$A^{-1} \cdot A \cdot x = A^{-1} \cdot y$ and thus $x = A^{-1} \cdot y$

To solve this in Octave (with the real values replaced by '*'):

```
octave:1> A = [
> *,*,0,0,0;          (state 3)
> *,*,*,*,0;          (state 2)
> 0,*,*,0,0;          (state 1.L)
> 0,*,0,*,*;          (state 1.R)
> 1,1,1,1,1]
A = <big printout of a 6x6 matrix>
octave:2> y = [0;0;0;0;1]
y = <6-element column vector>
octave:3> inv(A) * y
ans = <your answer>
```

Question 4, part 3: Using the answer from part 2, determine the fraction of time that there are 0, 1, 2, and 3 philosophers thinking.

Compare your answers to the simulation results from part 3. They should match (within perhaps 3 decimal places) if you run the simulation for 10,000 or 100,000 simulated seconds.

rev Oct. 15 – added part 4, values for T_think and T_eat.
Rev Oct. 18 – 5-state markov model, revised due date