Homework #4: Whack-a-Mole and Race Conditions

Issued: Thursday, October 12
Due: Tuesday, November 14

Purpose

This assignment asks you to learn about and practice single-process multi-thread (MT) programming, including the design and implementation of MT-safe functions and data structures.

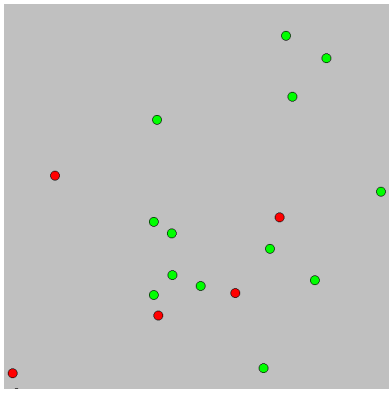
Skeletal Code

We begin with a single-threaded program, which simulates the old arcade game, graphically, but not interactively:

```
pub/hw4
```

It repeats this sad sequence: A green circle ("mole") appears on the grid ("lawn"), turns red (is "whacked"), and soon disappears.

Eventually, your program will display something like this:



The main module is simple:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#include "lawn.h"
#include "mole.h"

static Mole produce(Lawn l) { return mole_new(l,0,0); }
static void consume(Mole m) { mole_whack(m); }

int main() {
  srandom(time(0));
  const int n=10;
  Lawn lawn=lawn_new(0,0);
  for (int i=1; i<=n; i++)
    consume(produce(lawn));
  lawn_free(lawn);
}
```

How to Proceed

1. Begin by getting the provided, single-threaded, program to build and execute. If you are not using an `onyx` node, you might need to install some software (e.g., the FLTK graphics development library).

Although you still need to link against the FLTK library, you can get textual output (to `stdout`), rather than graphical output (to an X11 window), via an environment variable. For example:

```
$ DISPLAY= ./wam
```

2. Modify the program to use your queue, from the earlier homework assignment. You should only need to change `main.c`. Functions `produce` and `consume` should only communicate through a queue.

3. Modify the program to use a separate thread for each `produce` and `consume` call. Design and implement a new module for the thread code. Provide a function to call another function (e.g., `produce` or `consume`) n times, in n new threads (q.v., `pthread_create`). Pass an argument to each thread/function, providing the data it needs to access. The argument can be a pointer to a structure containing multiple values. Provide another function to wait for the n threads, called earlier, to finish (q.v., `pthread_join`).

   MT programming is described in Chapters 29 and 30 of our textbook. My `produce` function, in `main.c`, looks like this:

```
1 static void *produce(void *a) {
2   void **arg=a;
3   Deq q=(Deq)arg[0];
4   Lawn l=(Lawn)arg[1];
5   deq_tail_put(q,mole_new(l,0,0));
6   return 0;
7 }
```

   At this point, your program should crash: Your MT code is accessing your MT-unsafe queue.

4. Design and implement an MT-safe "wrapper" module, around your queue module. For this assignment, one lock per queue is good enough. Your queue module need not change. This is also described in Chapters 29 and 30 of our textbook.

   To increase the entertainment value of your simulation, allow an argument to `mtq_new` to set a maximum capacity, with zero meaning unbounded. I used a capacity of four. This will cause `produce` congestion.

   Use condition variables, to improve the performance of your MT-safe queues.

5. Modify the program to use your MT-safe wrapper, rather than your queue module. You should only need to change `main.c`, referring to types and functions from the MT-safe wrapper, rather than the MT-unsafe original.

Now, my `produce` function looks like this:

```
1  static void *produce(void *a) {
2    void **arg=a;
3    Mtq q=(Mtq)arg[0];
4    Lawn l=(Lawn)arg[1];
5    mtq_tail_put(q,mole_new(l,0,0));
6    return 0;
7  }
```

Requirements

1. Use `valgrind` to show that your module has no memory leaks. A set of "suppressions" prevent the `fltk` library from cluttering your `valgrind` output. See the `vg` script.
2. Provide good documentation and error messages.
3. Your submission will be evaluated on `onyx`.