Comp111 Assignment 2: ants and aardvarks

Overview

One of the most famous problems in concurrency is that of the "Dining Philosophers", who must share knives and forks in a concurrent manner. But this has been studied too much in the literature, so instead, we will study the "Dining Aardvarks".

There are 11 aardvarks and 3 anthills. There are 100 ants in each anthill. Only 3 aardvarks can share one anthill at a time. It takes one second of real time for an aardvark to slurp up an ant, and another second for the aardvark to swallow it, during which time it is not using the anthill and another aardvark can start slurping. However, should an aardvark attempt to slurp an ant from an anthill where 3 aardvarks are already slurping, or make any other kind of mistake, including attempting to slurp from an already empty anthill or an anthill that doesn't exist, it will be *4 seconds* before the aardvark is available to slurp again. The simulator calls that 'sulking'.

Objectives

Your objective is to manage the aardvarks so that all of the ants are consumed in a minimal amount of real time. Each aardvark is a thread in a multi-threaded program. You are given a simulator for the anthills that invokes the aardvarks and measures the resulting behavior.

You must write one thread

```
extern void *aardvark(void *);
```

that is *invoked 11 times*. This thread should simulate the behavior of one aardvark so that 11 instances of this thread solve the problem.

When invoked, the thread has one argument passed to it, which is the name of the aardvark that it is simulating. The bare skeleton of a thread is something like:

```
int initialized=0;
void *aardvark(void *input) {
    char aname = *(char *)input; // name of aardvark, for debugging purposes
    pthread_mutex_lock(&init_lock); // declared in anthills.h
    if (!initialized++) { // this is executed for only one thread.
        // initialize all variables, mutexes, semaphores here
    }
    pthread_mutex_unlock(&init_lock);
    while (chow_time()) {  // there is an ant to eat
        // try to slurp an ant!
        slurp(aname);
    }
    return NULL;
}
```

Each thread may utilize the following interface, documented in `/comp/111/assignments/a2/anthills.h`:

```
#define TRUE 1
#define FALSE 0
#define AARDVARKS 11
#define ANTHILLS 3
```

```
#define ANTS_PER_HILL 100
#define AARDVARKS_PER_HILL 3
extern int slurp(char aname, int anthill); // eat one ant.
extern int chow_time(); // whether there are ants to eat
extern double elapsed(); // how much time has been spent?
extern pthread_mutex_t init_lock; // resolve init race conditions
```

- `slurp` tries to slurp up an ant. It returns in two seconds of real time if it succeeds, and *4 seconds of real time if it does not*. During the second second of a successful `slurp`, the hill can be slurped again. However, the simulator "believes" the thread about what constitutes the identity of his aardvark. One aardvark will not be allowed to slurp from two anthills at the same time, if another thread "impersonates" the same aardvark by giving the wrong name. `slurp` returns 1 if it succeeds (and the aardvark actually consumes an ant) and 0 if it fails.
- `chow_time` returns 1 if there's an ant left anywhere, and 0 if not. Your threads should return when this returns 0.
- `elapsed` returns the seconds of real time (wallclock time) since the start of the simulation.

A starting version of the aardvarks code may be found in `/comp/111/assignments/a2/aardvarks.c`.

The Simulator

You will test your `aardvarks.c` by use of a simulator, currently contained in `/comp/111/assignments/a2/anthills.c` and `/comp/111/assignments/a2/anthills.h`. You may not modify this simulator (except to test your theories), but may use whatever information you can glean from the source code. During grading, *this simulator will be the one that is used*. The simulator creates your threads and waits for them to complete. It also provides mechanisms by which your threads must `slurp` the ants. These mechanisms employ concurrency locks in the manner discussed in class: see the code for details.

Grading

Assignment 4 is worth 10 points, including 2 points style and 8 points based upon the actual performance of your program in elapsed time. Note that if your threads do not complete, your score for that part is 0. *You must not deadlock!*

Getting started

To get started,

```
mkdir a2
cd a2
cp /comp/111/assignments/a2/aardvarks.c .
ln -s /comp/111/assignments/a2/anthills.o .
gcc -g aardvarks.c anthills.o -lpthread -lrt
./a.out
```

This is a pretty miserable version of the aardvarks that makes grevious errors. Edit `aardvarks.c` to create your solution, and use the above compilation command.

Using the Simulator

The simulator takes several arguments that control what it prints.

- `quiet`: suppress normal printing, run quietly.
- `debug`: report errors explicitly.
- `trace`: trace the whole state of the simulator.
- `csv`: create `output.csv` containing a machine-readable trace of the simulation.

The last option can be utilized to visualize the aardvarks.

Visualizing the Aardvarks

There is a simple visualizer for the aardvarks in `/comp/111/assignments/a2/see.R`. This is a program in the R language for statistical analysis. To use it, first run your a.out using the `csv` argument. Then, in the same directory, on any workstation, type:

```
% ln -s /comp/111/assignments/a2/see.R .
% R
>  source('see.R')
>  q()
```

You can also run this on any windows machine, by first transferring output.csv to the machine.

In the visualization, the aardvarks are on Y while wallclock time is on X.

- green indicates that an aardvark is slurping (the shade determines the anthill).
- yellow indicates that an aardvark is swallowing.
- red indicates that an aardvark is sulking (after an error or concurrency violation).
- gray indicates that an aardvark is idle.

Hints

Some initial comments:

- When an aardvark finishes slurping, it rests for a second, during which time another aardvark can -- in principle -- slurp in its stead.
- I am not going to tell you when that second is up! But you have the `elapsed()` function to help.
- Quite obviously, you are going to have to implement your own locking for best performance. I penalize an aardvark rather severely for trying to exceed the "slurp limit" of 3 aardvarks per hill.
- Your aardvarks can collude! You can use whatever mechanism you want to communicate between them, remembering that they are operating in a shared memory environment. But remember to use your own locking!
- You may utilize semaphores; the main consumption of time is when the aardvarks "slurp".
- While you can engineer the locations of my simulator variables and walk on them, it will be easy to see that you're doing this in the source. Don't do that.
- It is rather important to utilize an up-to-date simulator. As the assignment progresses, I will fix bugs and perhaps implement helpful features.

Some of my dirty tricks

Some of my dirty tricks:

- Track the state of the simulator in my own global variables (protected by mutexes!)
- Wrap the `slurp` routine inside my routine `my_slurp` that calls `slurp` and updates state

accordingly. Then if I only call `my_slurp`, my idea of state is always correct.
- Use that knowledge of state to avoid concurrency botches.
- My first thread initializes all of my mutexes; all the others wait for it. A global variable changes when everything is initialized. (Without this, havoc ensued; threads locked mutexes before they were initialized!)

Submitting completed assignments

Your 11 threads should be in a single c program `aardvarks.c`. Programs in other languages are unacceptable. Your program must compile with the following compilation command:

```
gcc -g aardvarks.c /comp/111/assignments/a2/anthills.o -lpthread -lrt
```

To submit this program, first ssh to `comp111-01` or `-02`, then type:

```
provide comp111 a2 aardvarks.c
```

where `aardvarks.c` is a file containing all of your program.

Your submission will be graded offline on one of the comp111 machines. To see grading status or comments, type:

```
progress comp111
```