6.836 Embodied Intelligence, 2001
Research Assignment 5
Issued April 13, due April 27.

On the web site is a system called **Simple**. It is a very lightweight scheduler that could run in an unthreaded operating system on a small embedded processor. It gives the user multiple threads that can share memory space, and have very low overhead for context switching. In order to run it under *Linux* it makes use of a 100Hz interrupt to provide a 100Hz clock, plus regular calls to motor driver routines. The basic clock tick is thus 0.01 seconds. Also to provide a typing interface for the purposes of this research assignment, it includes the same front end lisp system used in the **Sierra** research assignment. That would not be present on a small robot with an embedded processor.

The current version of the code and the make file produce an application called `try`, making use of source code in `test.c`. In this assignment you should replace `test.c` with your own code. But, you might find it easy to start with that and play around with it. Note that there is an interface set up for two lisp procedures of no arguments, `stats`, and `istats`, which respectively print out a display of all the threads, and reinitialize the statistics gathering about the threads.

The file `simple.doc` provides documentation of the **Simple** system. It lets you run hundreds of threads with different scheduling specifications. Each thread runs until it yields, so it is important that you make your threads return! There is a mail box system so that messages can be passed between threads, where a thread has blocked on waiting for a particular message. There is no guarantee on ordering of when threads are run–they should be thought of as asynchronous.

1. Write a simple Braitenberg vehicle simulator using the `motor_interrupt` routine to do the updating of the vehicles' positions. Note that since this routine gets called 100 times a second it should do very little, simply updating the state variables (position and velocity in the appropriate coordinate system), based on the current torques of the motors. You can chose how complex you want to make the physics of your world, but it should be realistic in some sense. The simplest world would be a frictionless two dimensional world, i.e., a slice of free space, with the two motors of each vehicle being rocket engines providing pure forces. I recommend trying this first.

Now write a thread which somehow gives a display of the vehicles' positions and orientations at some regular intervals. The simplest version of this will simply do `printf`s. If you are more aggresive you might run a separate X-window with a graphical display, but this is certainly not expected or required.

Lastly you need to be able to set different motor torques/forces. You can do this by using `register_cprocedure` to connect a Lisp procedure name to an underlying C procedure. E.g.,

```
  register_cprocedure("setm", set_motors,
                      c_void, c_int, c_double, c_double);
```

will let you type to the lisp front end:

```
#? (setm 2 4.5 6)
```

meaning set the two forces on vehicle 2 to be 4.5 and 6.0. The prototype for the C procedure `set_motors` would need to be:

```
void set_motors(int, double *, double *);
```

where the types `void`, `int`, `double *`, and `double *`, are exactly the types refered to above in the call to `register_cprocedure`. It is `double *`, rather than `double`, as the calling convention used between Lisp and C requires every quantity on the stack to be exactly 32 bits.

With `set_motors` properly defined you will be able to type new motor commands to your simulator as the vehicles move about. [Notice that in the supplied file `test.c` there is already a version of this procedure which simply prints out its arguments when called from Lisp.]

2. Now use multiple threads to make a control system for a Braitenberg vehicle. You will need to have a way of placing objects with properties in the world, and have the threads in your control system have access to sensors which can sense something about those simulated obstacles. Note that all the sensor processing should get done in calls from your control threads, rather than in the motor interrupt routine. If you have moving objects they would be controlled by other threads, so be sure to make clear with comments and groupings in your files which of your threads are those which are simulating the environment, and which are those that are part of your controller for your vehicle.

Because you can have arbitrary C code in your threads you are not restricted to the simple two wire sorts of Braitenberg vehicles we had at the start of the term. Instead you can have fairly rich sensors and complex non-linear interactions, along with memory and decision making.

Build some threads to let your vehicle avoid obstacles. Then add threads that let it approach a particularly interesting sort of object (you can make up sensors as you want; e.g., sonar for obstacle avoidance, but a light sensor to attracted to "fireflies").

Now make the "interesting" objects move about in some way, and add a new layer of threads which let your creature do something interesting in the way of anticipation, be it a fixed action pattern, or a learned response.

Turn in the code for all your threads and show some evaluation of how well you creature does. Critique your creature.

3. Independent of any particular application on whatever computer you are running explore how many threads you can run in **Simple** before the the context switching overhead starts to overwhelm your machine. Show the code you used to test this, report on your methodology and your results.