1 Concurrency Models

A computer system which models a natural system must somehow simulate concurrency. A computer is by nature a sequential device. However since computers run very fast, it is possible for a computer to seem as though it is doing many things at once. Computers achieve this by dividing resources into small time-slices and sharing those slices across typically either multiple threads or multiple processes. This short paper will present two programming models of concurrency and discuss how they can be implemented under LabWindows.

Usually when one thinks of a group of cars moving down a highway, one thinks of a collection of independent entities which share a context or space (the road in this case) but must also be aware of each other in order to avoid collisions.

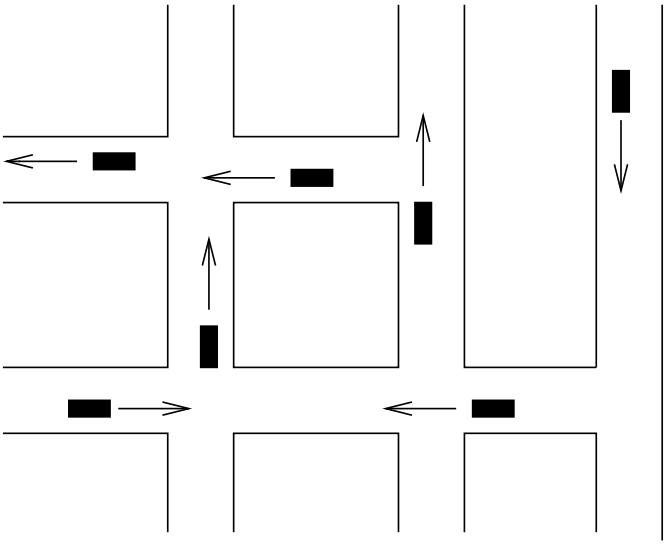

Figure 1: Traffic Congestion Model

Suppose we want to model a system of roads, where drivers choose typical destinations. Some are going to work, some are going shopping, and some are going on holiday. If we are modeling this system in a computer, it would be natural for each car to be represented by an autonomous process. A process here means some sort of isolated program, which while it can be aware of the other programs, it is not necessarily dependent on them. Processes should only communicate explicitly when they want to.

In the case of the traffic model, suppose that each car process had a unique starting point, and a unique destination, its own route, and could follow traffic rules then we could develop a relatively realistic traffic congestion model. The road system might be represented by a governing

rule process which communicated with the cars, and indicated which routes were legal. This style of simulation system has been studied extensively and is usually referred to as a discrete event system. Typical applications include traffic models, telephone network models and factory floor production models.

2 A Continuous Example

In our case we are interested in coupled physical systems which behave in continuous rather than discrete steps. While it is certainly correct to think of physical models as continuous systems, for the sake of modeling them on a computer they must be descretized. Discretization is unavoidable because the machines representation of a floating point number has some limit to its accuracy, and because typical numerical time integration schemes are based on finite differences.

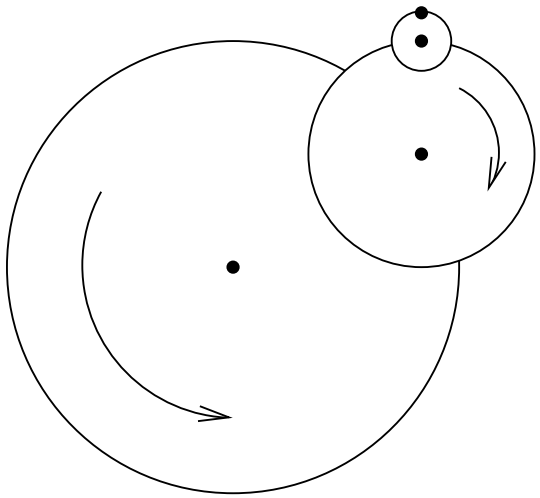

Figure 2: Connected Wheels

Suppose we want to model a continuous physical system which represents a small group of connected and related objects. For this example we will think of three wheels of various diameters which are connected as shown in figure 2.

Each wheel has a pin which goes through its center, the pin of the largest wheel is connected to a stationary post. The second largest wheel is connected to the edge of the first, and the smallest is again connected to the edge of the second largest. The three wheels spin at different rates, and each wheel can be started and stopped, independent of the others. The smallest wheel has a pin connected to its edge, and for this example the problem of interest is to compute the vertical position of this final edge pin.

Suppose we use the following three functions to model motion of pins in the edges of the wheels:

$val1 = 0.6 \times sin(t)$;

$val2 = 0.3 \times sin(t \times 3)$;

$val3 = 0.1 \times sin(t \times 9)$;

Then the position of the final edge pin will simply be the following sum:

$pos = val1 + val2 + val3$

For the sake of exposition, we have implemented each sine function evaluation in a separate thread of control. Although for this case the sine function evaluation is very simple, we could imagine a much more complicated evaluation where this mathematical separation might be very helpful.

The following figure shows the display output from the completed program:

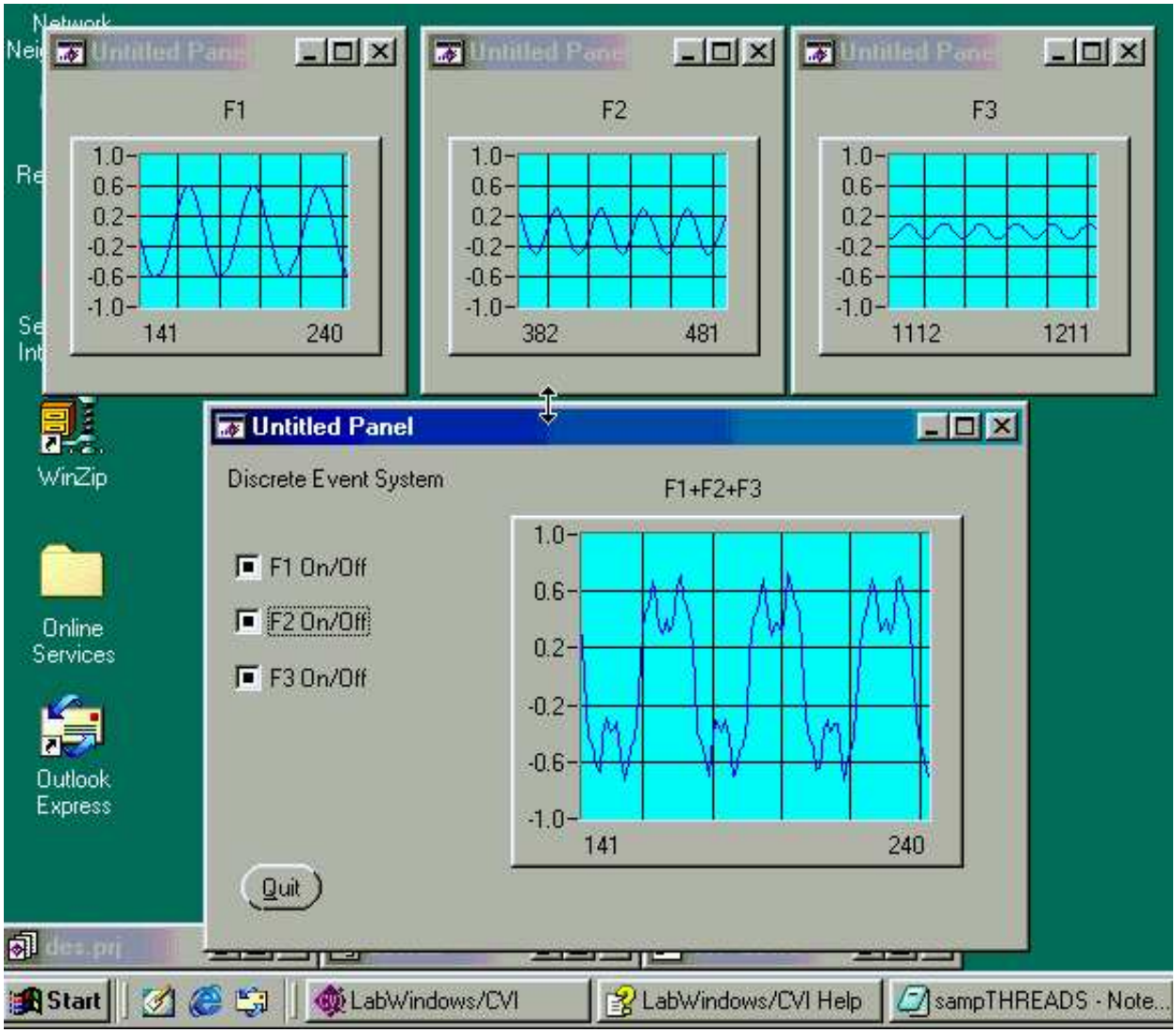

Figure 3: Lab Windows Summation of Sines

The top three windows display output generated by the three sine functions. F3 (the highest frequency function) requires the most frequent updates. F1 (the lowest frequency function) requires the least frequent updates. If a function isn't updated frequently enough then the output becomes inaccurate and this is undesirable. The central window shows the main thread which access the computed values from the other threads through global variables. The main thread computes the sum of the values from the other threads.

Suppose instead of using control threads we computed the final position as:

$pos = 0.6 \times sin(t) + 0.3 \times sin(t \times 3) + 0.1 \times sin(t \times 9)$

then it could be argued that the low frequency elements are being computed too often, and so CPU power is being wasted. Again we should imagine that we are actually computing something very complex, not simply a sine function.

So two principle arguments in favor of threading a continuous simulation are:

- Simplifies the problem through modularization
- Potentially reduces computational overhead

2.1 Discrete Event Programming

One way to implement the system is to use scheduled events. For this example we are only looking at four processes, and we suppose that each thread knows how often it should be called. We also suppose that there is some sort of global clock, that each process can access. In LabWindows this is easily implemented with a timer that updates a global variable. An individual thread which computes one of the sine functions might look like this:

```
/*---------------------------------------------
 * schedule_f1
 *---------------------------------------------
 */
void schedule_f1()  {
        if(G_timer<G_timer1) return;
        G_timer1=G_timer+.2;
        G_val1=.6*sin(G_timer);
        PlotStripChartPoint (panel2, PANEL_2_STRIPCHART, G_val1);
 }
```

The variables G_timer, G_timer1, and G_val1 are all global variables implemented for this procedure. When the function schedule_f1() is called it compares the main timer (G_timer) with its next scheduled event (G_timer1). If the time is right for another calculation then it first schedules a new event .2 seconds in the future, and then computes a new global value for its sine function (G_val1). After computing the partial sine result, it plots the data in its strip chart. If the event isn't ready it returns and does nothing.

We can write as many functions like this as we need. For our example we need 4 functions, one thread of control for each sine calculation and a fourth thread for the main window.

We also need some kind of scheduler function which manages each of the threads. In this case the manager function is implemented as a callback from the timer.

```
/*-------------------------------------------------------
 * Timer_sched
 *
 * A single timer is the life of the system. At each clock tick,
 * each thread of control is executed. Threads decide if they will
 * execute a given action or simply return.
 *-------------------------------------------------------
 */
int CVICALLBACK Timer_sched (int panel, int control, int event,
                void *callbackData, int eventData1, int eventData2)
{
        switch (event)
        {
        case EVENT_TIMER_TICK:
             G_timer+=.02;
             if (G_f1==1) schedule_f1();
             if (G_f2==1) schedule_f2();
             if (G_f3==1) schedule_f3();
             schedule_main();
             break;
        }
        return 0;
}
```

This example includes the possibility of turning the threads on and off, the global variables G_f1, G_f2, G_f3 are connected to the switches on the main panel and allow threads to be effectively blocked.

2.2 Thread Programming

Discrete event programming has several pitfalls. The most obvious is that if the function representing a specific thread simply does not return, then all of the other threads are blocked. It is also the case that if one of the functions is much more complex than the others perhaps it will be given an unfair amount of the CPUs attention. The basic response is to allow the operating system to worry about thread management. LabWindows includes a thread facility and it is worth investigating as an alternative.

LabWindows threads are created in pools. A pool defines a specific number of threads which can be drawn from the pool. For our case we need a pool of four threads, this is created as follows:

```
// Create a pool of 4 Threads for this application
CmtNewThreadPool (4, &G_poolHandle);

// Create the threads
CmtScheduleThreadPoolFunction (G_poolHandle,
      schedule_f1,(void *)ctrlID, NULL);
CmtScheduleThreadPoolFunction (G_poolHandle,
      schedule_f2,(void *)ctrlID, NULL);
CmtScheduleThreadPoolFunction (G_poolHandle,
      schedule_f3,(void *)ctrlID, NULL);
CmtScheduleThreadPoolFunction (G_poolHandle,
      schedule_main,(void *)ctrlID, NULL);
```

Once the threads are created they become active. The thread functions no longer need to exit, they can run in an infinite loop, so long as the application is still running. A thread function now looks like this:

```
/*-------------------------------------------------
 *  schedule_f1
 *-------------------------------------------------
 */
int CVICALLBACK schedule_f1(void * ctrlID)
{
    while(G_running) {
            if(G_timer>=G_timer1) {
                  G_timer1=G_timer+.2;
                  G_val1=.6*sin(G_timer);
                   PlotStripChartPoint (panel2,
                        PANEL_2_STRIPCHART,G_val1);
             }
             Delay(.02);
    }
}
```

The LabWindows thread runs in an infinite loop so long as the global variable G_running=1. The loop checks the global clock to see if it is time for its event. If the event condition is met,

it schedules a new event, computes its sine function, and plots a point on its graph as before. In either case whether the event condition is met or not, a small delay is executed. The delay prevents the thread from needlessly consuming the CPU resources as it checks for its event.

The delay in the thread could be avoided altogether by usage of the threads signaling facilities. It is generally undesirable for threads to poll for events, it would be better if a central scheduler process could signal the thread when it needed to be activated. In this case the scheduler is very simple and only needs to update the global time:

```
int CVICALLBACK Timer_sched (int panel, int control, int event,
                void *callbackData, int eventData1, int eventData2)
{
        switch (event)
        {
        case EVENT_TIMER_TICK:
             G_timer+=.02;
             break;
        }
        return 0;
}
```

3 Comments

This document is intended to generate some discussion on the topic of concurrency. The supplied implementations, for both the discrete event model, and the thread programming model are based on polling, which while probably acceptable, would be better replaced by some sort of global event queue which could signal threads when they needed to be woken. The excessive use of global variables for communication is unfortunate, but I think necessary.

Both problems were implemented under lab windows, and it was observed (although not measured) that the threaded system was significantly slower than the discrete event system. A event queue might solve this problem by eliminating excessive polling in the threaded system. However letting the operating system manage the threads will tend to be much slower than doing the thread management locally, so it is not surprising that the threaded version was slower.

Much can be said about thread programming, concurrency and discrete event simulations. I hesitate to recommend any of these methods, since I really believe that simplicity ought to be the rule. Concurrency and thread programming are usually treated in upper year computer science courses, and I think we need to choose a really simple model as our base. Adding network programming to the mix will complicate this further, so its important that we choose a straightforward easy to understand model of execution near the start.