Threaded C and Freezer OS

Jacky Baltes, Chris Iverach-Brereton, Chi Tai Cheng, and John Anderson

Autonomous Agents Lab,
University of Manitoba,
Winnipeg, Manitoba,
Canada, R3T 2N2
jacky@cs.umanitoba.ca
http://www.cs.umanitoba.ca/~jacky

Abstract. Threaded C is a meta-language that is based on C, but is annotated with thread, monitor thread, and semaphore markup. Threaded C uses the runtime provided by the Freezer OS, a small, memory-efficient embedded kernel. The combination of Freezer OS and Threaded C allows the simple expression of common control problems in robotics. The system is geared especially towards robotics education, as it matches the mental map that children have of how control structures should work.

Keywords: Scheduling, Real-time OS, Embedded Systems.

1 Introduction

This paper introduces Threaded C (TC), a meta-language that supports threads, monitors, and semaphores; and the Freezer OS, an efficient embedded kernel for the AVR AtMega128 series of micro-controllers [3].

The main contribution of TC is support for asynchronous support structures, which greatly simplifies the development of asynchronous control programs for robotics.

In this paper, we motivate the design of TC and Freezer OS by looking at the use of our system in the setting of an introductory robotics course. We demonstrate that asynchronous control structures are understood by children.

2 Motivation

Firstly, we will give an introduction into some of the unique and non-standard features of introductory programming for robots, especially for children. The authors have a lot of experience with introductory workshops for children in robotics and it is this experience that guided our design of the Freezer OS and Threaded C.

Drive Problem: The first problem we ask the students to solve is to drive the robot forward for one meter. We introduce the `DriveMotor(int s, int speed)` routines and tell them that the motors can be stopped by setting their speed to 0. The children quickly realize that this does not allow them to control the

T.-H.S. Li et al. (Eds.): FIRA 2011, CCIS 212, pp. 170–177, 2011.

distance that the robot drives. So we also introduce the `Wait(int timeout)` routine. Children then learn how increasing/decreasing the time out will drive the robot a longer/shorter distance.

The program that the students write looks as follows:

```
=== drive.c ===
void main() {
   DriveMotor( LEFT_MOTOR, 20);
   DriveMotor( RIGHT_MOTOR, 20);
   Wait( 100 );
   DriveMotor( LEFT_MOTOR, 0);
   DriveMotor( RIGHT_MOTOR, 0);
}
```

The children learn that the robot will execute sequences of instructions and learn that a simple robot has to convert timing values into distances.

Robot Dance Problem: In the second program, we ask students to develop a simple dance for the robot, which consists of straight line segments and turns. The children realize that the code is just a long string of repeating commands. We introduce subroutines to the children and tell them that by using a subroutine they can save themselves a lot of typing. We also introduce variables and tell them that they are like place holders for numbers. A typical program created by the children at this point will look as follows:

```
=== robotdance.c ===
void DriveStraight( int distance ) {
   DriveMotor( LEFT_MOTOR, 20);
   DriveMotor( RIGHT_MOTOR, 20);
   Wait( distance );
   DriveMotor( LEFT_MOTOR, 0);
   DriveMotor( RIGHT_MOTOR, 0);
}

void TurnLeft( int angle ) { ...

void TurnRight( int angle ) { ...

void main() {
  DriveStraight(30);
  TurnLeft( 120 );
  ...
}
```

Squares Problem: The next task for the children is to drive larger and larger squares using their robot and to stop when the size of the square is larger than a limit. This introduces variables, simple arithmetic operators, and `while` loops to the children.

Below we show the changes to the previous program to implement the squares subroutine:

```
=== squares.c ===
void Square( int limit ) {
   int distance = 10;
   while( distance < limit ) {
      DriveStraight( distance );
      TurnLeft( 400 );  // Results in a 90 degree turn
      distance = distance + 10
   }
}
```

Search Line Problem: The last task for the children is to make a simple line-tracking robot. The robot begins on a white surface with a path marked by a dark line. The robot must locate the line and follow it to an objective. We use this problem to teach children the three fundamental components of programming in imperative programming languages: sequence, selection, and iteration.

Fig. 1. Line Trackers are a popular entry problem for robotics and can act as navigation and localization system for more complex tasks

The first step is to simply have the robot drive an increasing square and stop when part of the robot touches the line. We introduce the `int GetLightSensor()` function. If the robot has an output device we also introduce a `PrintNumber(int number)` function to the children at this point.

This is the *crucial* part in the exercise. Many children try the following program which unfortunately does not work, since the light sensor will only be read at the end of the straight line and turn.

```
=== search_line_ideal.tc ===
void SearchLine( ) {
   int distance = 10;
   while( LightSensor() < 100 ) /* run_while */ {
      DriveStraight( distance );
      TurnLeft( 400 );  // Results in a 90 degree turn
      distance = distance + 10
   }
   FollowLine();
}
```

Even though children can understand why their program does not work, it is not easy for them to fix it. In fact, fixing this problem requires major changes to the program. Firstly, the `Wait` calls have to be replaced with loops that check the light sensor and terminate the loop if the line is found. `DriveStraight`, which previously did exactly one task, now has to do two and has to know if we are interested in the light sensor. Secondly, a return argument needs to be added which tells the caller if `DriveStraight` returned because it covered the desired distance or because the track was found.

The correct sequential version of the line searcher will look as follows assuming `int DriveStraight(int distance, int sensor)` and corresponding turn routines:

```
=== line_search_sequential.c ===
int DriveStraight( int distance, int light ) {
   int i = 0;
   DriveMotor( LEFT_MOTOR, 20);
   DriveMotor( RIGHT_MOTOR, 20);
   while( i < distance ) {
      if ( light == 1 ) {
         if ( LightSensor() >= LIMIT ) {
            DriveMotor( LEFT_MOTOR, 0); DriveMotor( RIGHT_MOTOR, 0); return 1;
         }
      }
      i = i + 1;
   }
   DriveMotor( LEFT_MOTOR, 0); DriveMotor( RIGHT_MOTOR, 0); return 0;
}
...
int SearchLine( ) {
   int distance = 10; int result = 0;

   while( result == 0 ) {
      if ( ( result = DriveStraight( distance, 1 ) ) != 0 ) {
         if ( ( result = TurnLeft( 400, 1 ) ) != 0 ) {
            distance = distance + 10
         }
      }
   }
}
...
```

The first time that we run this workshop, this complexity turned out to be too difficult for the children. The nested conditionals, scoping rules for brackets, and function values was too hard for the children to grasp in a short weekend workshop.

2.1 Threads

After realizing that children had a lot of trouble with the sequential approach, we decided a different approach in the following year. Even though threads and

parallel processing are often not taught until the 3rd year of a B.Sc. degree, we introduced them to the children right from the start.

To our great surprise, children had little trouble with threads, mainly because they did not know about processors, timer interrupts, and reader-writer problems. For the children, it was easy to grasp that a computer can do more than one thing at a time. After all, children know that the real world has many things moving concurrently and they can multi-task themselves.

Therefore, in the subsequent years, we used NQC [2], a C dialect with primitive support for threads, to solve the `SearchLine` problem by introducing threads to the children and explaining how these can be started and stopped. The program then looked as follows:

```
=== line_search_threads.c ===
...
THREAD void Square( int limit ) {
   int distance = 10;
   while( distance < limit ) {
      DriveStraight( distance );
      TurnLeft( 400 );  // Results in a 90 degree turn
      distance = distance + 10
   }
}

int SearchLine( ) {
   startTask Square( 1000 );
   startTask DetectLine();

}
...
THREAD DetectLine( ) {
   if ( LightSensor() >= 100 ) {
      stopTask Square;
      startTask FollowLine();
   }
}
```

The syntax necessary to declare, start, and stop threads was at first confusing, but the general principle of the program was easy to grasp for the children. Using threads, children as young as ten were able to implement line searching and line following robots.

It is important to note that given the basic building blocks above for driving, turning, and reacting to the sensor, the children enjoyed experimenting and combining them in novel patterns. For example, changing the square pattern to a dance when looking for the line.

3 Freezer OS

After having run workshops for several years, the first author moved to another university and focused on other research areas, in particular humanoid robots[1].

Some of the robot designs were based on the Robotis Bioloid robotics kit, which uses an AVR ATMega128 micro-controller based system to control the motion.

His team developed the Freezer OS, a small, memory-efficient embedded real-time kernel for AVR-based devices. The Freezer OS is a pre-emptive multi-tasking kernel. The scheduler is a priority-based, round-robin scheduller that supports semaphores as synchronization primitives. It also included wait queues for interrupt driven devices such as the AD converter and the serial ports.

One issue in developing a kernel for a low-memory (4 KB RAM) processor such as the AtMega128 is the allocation of the stack space. Since threads may run in any order, we must allocate sufficient stack space for each thread. This fragments the memory and the system will crash if thread 1 overruns its allocated stack space, even though other threads may still have memory available.

However, as seen in the example above, often threads are only needed to check sensor readings or set outputs for an actuator. An example is reading the light sensor in the **Search Line Problem**. We therefore developed the concept of monitor threads. These threads are periodic tasks that run very quickly and do not use a large amount of stack space. Instead of reserving separate stack space for a monitor thread, a monitor thread will use the stack space of the currently running task. This means, that the monitor thread must terminate before the currently running thread can be rescheduled. This constraint is enforced in the Freezer OS scheduler that calls the monitor threads if necessary and then blocks execution of the currently running thread until the monitor thread terminates.

4 Threaded C

Previous versions of the Freezer OS kernel were used extensively to control the motions and deal with the communication of our small humanoid robots. The students at the Autonomous Agents Lab at the University of Manitoba would use standard C to start and stop tasks for example. However, manually dealing with the resources such as threads, stack space, and priorities would lead to problems. For example, students would forget to start a thread or initialize a semaphore correctly.

We therefore considered developing a new programming language to better support the Freezer OS and application development for our humanoid robots in general.

Clearly, developing a new language including compiler, libraries, and runtime is a non-trivial task. Since all of the students were comfortable programming in the C language and avr-gcc — a high quality open-source C compiler for the AVR family of processors — existed, we decided to instead implement a meta-programming language *Threaded C* (TC).

A TC program is a C program with additional markup to declare threads, monitor threads, and semaphores. A python preprocessor reads in TC files and converts them into a legal C files, which can then be compiled by any ANSI C compiler, including avr-gcc. The TC preprocessor also keeps track of the number of threads and sets global variables (e.g., `MAX_NUM_TASKS`, the maximum number

of tasks in the system) and creates a main function which will start the threads. An example of thread markup is shown below.

```
=== search_line.tc ===
%%% THREAD priority=medium stack_size=128 %%%
void search_line( ) {
    run_while( LightSensor() < 100 ) {
        DriveStraight( distance );
        TurnLeft( 400 );  // results in a 90 degree turn
        distance = distance + 10;
    }
    FollowLine( );
    ...
}
```

The main novelty in TC is the support for asynchronous control loops using `run_while()`, which is similar to a while loop, but the test will be evaluated asynchronously and the loop will immediately if the test condition becomes false. Internally, this is implemented as the creation of an implicit monitor thread, allocation of a `jmp_buf` structure and `setjmp` and `longjmp` instructions.

Using TC, and replacing `while` with `run_while`, the ideal **Search Line Problem** program shown above compiles and runs as expected. The TC preprocessor will take the `run_while` and convert it into the following C code. It is important to note that the user rarely if ever has to see the C code, since the C code is created from the TC file shown above.

```
=== search_line.c === /* DO NOT EDIT! NOT MANIPULATED BY USER */
void SearchLine( ) {
    jmp_buf jmpBuf;
    int jmpResult;
    int monitorId;

    jmpResult = setjmp(jmpBuf);
    if (jmpResult == 0) {
        monitorId = startAsyncWhileThread( &asyncMonitorFunction, &jmpBuf);
        while( LightSensor() < 100 ) {
            DriveStraight( distance );
            TurnLeft( 400 );  // Results in a 90 degree turn
            distance = distance + 10
        }
        stopMonitorThread(monitorId);
    }
    FollowLine();
    ...
}
int asyncMonitorFunction(void) {
    return (LightSensor() <= 100);
}
... In schedule(), called by Timer 1 interrupt
```

```
if ( !scheduledMonitor -> execute() ) {
    stopMonitorThread(scheduledMonitor -> tid); /* Disable all nested
monitors as well */
    task[current_task].state = READY;

    sei();
    longjmp(*(scheduledMonitor -> jmp_buf), 1);
} else {
    asm volatile("\t reti\n"::);  /* Continue the thread normally */
}
```

Particular care must be taken when dealing with nested asynchronous control structures, since the conditions run completely asynchronously and thus may terminate in any order. For example, assuming that `DriveStraight` uses another `run_while` to check a touch sensor for collisions, then it is possible that the light sensor monitor thread fires first and execution will continue with `FollowLine`. In this case, all monitor threads associated with the inner `run_while` must be terminated as well, as the context is not valid anymore. So the scheduler `startAsyncWhileThread` keeps saves the level of each new monitor thread. The routine `stopMonitorThread` will disable the current as well as all higher level monitor threads. The system then executes a `longjmp` with an argument of 1, which means execution will continue at line `FollowLine` of the ideal TC program.

5 Conclusion

The Freezer OS and TC combination works well for our applications. However, there are several extensions that we intend to work on in the future. One issue is that since the condition in the `run_while` is converted into its own subroutine it has no access to the local variables of its subroutine. This could be solved by also passing a pointer to the frame of the subroutine, but we have not been able to find a portable way of doing this in C.

References

1. Baltes, J., Anderson, J.: Complex AI on Small Embedded Systems: Humanoid Robotics Using Mobile Phones (2010)
2. Baum, D., Hansen, J.: Not quite c programmers guide, version 3.1r2 (2005), `http://bricxcc.sourceforge.net/nqc/`
3. AVR Corporation. Atmel avr atmega128 data sheet. Configurations (2010)