Embedded Systems Programming
Written Exam

2010-01-15
from 09:00 to 13:00

- **Allowed tools:** An english dictionary.
- **Grading criteria:** You can get at most 20 points.

  To pass you need at least 50% of the points.

  For the highest grade you need more than 90% of the points.
- **Responsible:** Verónica Gaspes, 7380 or 0739957638.

- **Read carefully!** Some exercises might include explanations, hints and/or some code. What you have to do in each exercise is marked with the points that you can get for solving it (as **(X pts.)**).
- **Write clearly!**
- **Motivate your answers!**

Good Luck!

1. **(2 pts.)** Write functions

   ```
   void set(unsigned int *port, unsigned int mask)
   void clear(unsigned int *port, unsigned int mask)
   ```

   to set resp. clear some bits in a port. The bits that have to be set or made clear in the port are the bits in the second argument (`mask`) that are set.

2. Consider the following fragments we discussed in one of the lectures. The first two functions use busy waiting to detect and read input values from a sonar and a radio device. We do not show the code for the functions that implement ordinary algorithms (`control` and `decode`) and for generating output to the servo device.

   ```
   int sonar_read(){
      while(SONAR_STATUS & READY == 0);
      return SONAR_DATA;
   }
   void radio_read(struct Packet *pkt){
     while(RADIO_STATUS & READY == 0);
     pkt->v1 = RADIO_DATA1;
     ...
     pkt->vn = RADIO_DATAn;
   }
   main(){
      struct Params params;
      struct Packet packet;
      int dist, signal;
      while(1){
        dist = sonar_read();
        control(dist, &signal, &params);
        servo_write(signal);

        radio_read(&packet);
        decode(&packet,&params);
      }
   }
   ```

   (a) **(1 pts.)** Discuss some of the problems with this style of programming.

   (b) **(1 pts.)** Show some way of modifying the program so that one of the input sources does not shadow the other one.

   (c) **(1 pts.)** Are there some remaining problems?

3. In laboration 2 we programmed with a kernel (tinythreads) that supports threads. Using the kernel function

   ```
   void spawn(void (* function)(int), int argument)
   ```

   a program can start a new thread to execute a call to a `function` with an integer `argument`. The different threads are interleaved automatically by the kernel that calls `yield()` at regular intervals (we call this timeslicing). With small enough intervals the program seems to be doing several things at the same time (concurrently).

   Imagine now a C program that uses this kernel and a function `f` that might be called from several concurrent threads. The function `f` both reads from and writes to the following types of variables:

   (a) its arguments,

   (b) global variables declared at top level of the program,

   (c) local variables declared within `f`,

   (d) heap-allocated variables, whose addresses are provided as arguments to `f`.

   (2 pts.) Indicate **for each of these cases** whether access to the variables in question may constitute a critical section, that has to be protected by some mutual exclusion mechanism.

4. Using the kernel TinyTimber you can organize programs with *reactive objects* while programming in C. As a programmer you have to follow some conventions and Tinytimber guarantees that the methods of a reactive object are executed strictly sequentialy, thus protecting the local state of the object from critical section problems.

   (3 pts.) Program a class for reactive objects that can be used to *protect* (or encapsulate) an array of integers. The array to be encapsulated can be provided on object initialization. Let the type introduced for the class be `Array`. Then, the methods that have to be provided are

   ```
   // record a position
   int setPosition(Array *self, int x)

   // set the value at the recorded position to x
   int setElement(Array *self, int x)

   // return the value at the recorded position
   int getElement(Array *self, int x)
   ```

5. In laboration 4 you used a reactive object to implement a *blinker* that turns on and off a LED. The blinker can be started, stopped and the period can be set.

   (a) **(3 pts.)** Assuming that you have a class for reactive LEDs, implement a blinker. The blinker should not be hard-coupled to a LED, instead, it should be possible to instantiate blinkers for different LEDs.

   (b) **(1 pts.)** If your blinker is going to be used together with other reactive objects that require processor time (for example computing prime numbers!), what can you do in your implementation to make sure that the blinker keeps up with its frequency?

6. In the lectures we discussed two ways of assigning priorities to task sets where all tasks are periodic with periods equal to the deadlines: Rate Monotonic and Earliest Deadline First.

   (a) **(2 pts.)** Explain how RM assigns priorities and give an example with a task set of 3 tasks (the example should show what priorities are assigned)

   (b) **(2 pts.)** Explain how EDF assigns priorities and give an example with a task set of 3 tasks (the example should show what priorities are assigned)

Continues in the next page!

7. The following Ada fragment can be used to implement mutexes:

```
task type Mutex is
  entry Lock;
  entry Unlock;
end Mutex;

task body Mutex is
   Locked : Boolean := False;
begin
   loop
      select
         when not Locked =>
               accept Lock do
                  Locked := True;
               end Lock;
      or
         accept Unlock do
            Locked := False;
         end Unlock;
      end select;
   end loop;
end Mutex;
```

(2 pts.) Explain how it can be used in a program that needs to protect a critical region.