uOttawa

L'Université canadienne
Canada's university

SEG4145:
REAL-TIME AND EMBEDDED SOFTWARE DESIGN
WINTER 2019

ASSIGNMENT #2 SOLUTIONS

Question 1 (5 points)
Choose the statement, which better describes a Real-Time Operating System

1. A RTOS is an operating system that guarantees a certain capability (execution of a task) within a specified time constraint;
2. An OS is a system program that provides an interface between application programs and the computer system hardware;
3. The applications where dependability that a certain task will finish before a particular deadline is just as obtaining the correct result;
4. All of the above.

Answer: 4. All of the above

Question 2 (15 points

a) (5 points) What is the principal difference between a background task and a foreground task?
b) (10 points) What is a process? What are the attributes of a process? What is a thread?

Answer:

a) A foreground task is triggered by an interrupt event.

When a foreground operation is called, the task in an executing state is interrupted and has to wait until the foreground operation is completed. Only after the completion of the foreground task, the task which was in execution before the interrupt was raised, can go on to execute other instructions.
A foreground operation is useful when data processing can be done after all data is collected and when no other tasks need to be done during the scan. Note also that

during a foreground operation, the computer is effectively inaccessible by the operator or any other program while the process is running.

Many applications require processing the data as it comes in. In this case background operations should be used. For the data to be obtained a scan operation has to take place the system control is given back to the program while the scan is taking place in the background. This way the program can execute other useful instructions and functions. The most common reason for using this feature is the need to process data while the scan is still in process.
It is always recommended to call, the special function that stops to the background operation properly.

Therefore, the main difference between a background and a foreground task is related to the way that they are launched in execution.

A Foreground task is launched in execution by a specific interrupt and has access to the terminal standard i/os while background processes typically run with little or no user (the owner of I/O processes) interaction at all. They interact with the system and they are under Operating System control.

b) **A process**, in a real-time system, and under any operating systems, is an instance of a computer program that is being executed by the CPU, while a computer program can also define a number of processes related to it.

   Some processes require the simultaneous execution of some instructions related to them.
   Some operating systems may allow the division of processes in a set of other processes, called threads.
   Processes therefore define the main units of 'work' in an operating system. A process is a "heavyweight" unit of kernel scheduling, as creating, destroying, and switching processes is relatively computationally expensive.

   A process is also called task, though large Real-Time Processes (STP- straight through processing- very much used in real-time finance operations – like algorithmic stock gambling) can contain many tasks.
   A process is defined by the following **attributes:**
   - the running program
   - the logical unit of work scheduled by operating system to execute the tasks
   - the data structure with the following information State of execution
     - Identity (real-time)
     - Attributes (i.e. execution time) Associated resources

   Processes own resources allocated by the operating system. Resources include memory (for both code and data), file handles, sockets, device handles, windows, and a process control block.

Another important characteristic of computer processes is that they are isolated by process isolation, and do not share address spaces or file resources except through explicit methods such as inheriting file handles or shared memory segments, or mapping the same file in a shared way (interprocess communication).

Creating or destroying a process is relatively expensive, as resources must be acquired or released. Processes are typically preemptively multitasked, and process switching is relatively expensive, beyond basic cost of context switching, due to issues such as cache flushing.

A thread is the smallest sequence of instructions contained in a process that can be managed independently inside that process, by the scheduler. In other words, a thread is a lightweight process which resides in a process, thus shares resources inside a process.
A *thread* is an independent flow of control that operates within the same address space as other independent flows of control within a process.

At least one kernel thread exists within each process. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently.
There are two types of threads:
• kernel threads
• user defined threads
A kernel thread is a kernel entity, like processes and interrupt handlers; it is the entity handled by the system scheduler. A kernel thread runs in user mode environment when executing user functions or library calls; it switches to kernel mode environment when executing system calls.
A **kernel-only thread** is a kernel thread that executes only in kernel mode environment. Kernel-only threads are controlled by the kernel mode environment programmer through kernel services.

User mode programs can access user threads through a library (such as the **libpthreads.a** the threads library of IBM AIX).

User threads are part of a portable programming model.
User threads are mapped to kernel threads by the threads library in an implementation dependent manner.
The threads library uses a proprietary interface to handle kernel threads.

Question 3 (5 points)
A task whose execution begins randomly and within the interval of arrival time of consecutive tasks is known as:

a) periodic task;

b) aperiodic task;
c) sporadic task.

Answer: b) Aperiodic task

Question 4 (5 points)
The time T between two consecutive sensor reading is called:

a) sampling period;
b) response time;
c) turn around time.

Answer: a) Sampling period

Question 5 (10 points)
Describe the sequence of steps when a timer interrupt occurs which eventually results in a context switch to another process.

Answer:

The following is a sequence of steps which occur when a timing interrupt is raised:

a) The timer interrupt is called
b) A trap into kernel space triggers the switching the kernel stack
c) Current application's registers are saved
d) The scbeduler gives next thread which should be run
e) Context switch to other thread loading its stack and flushing the Translation Lookaside Buffer (TLB)
f) Load the register of the interrupted task from the kernel stack
g) Program Counter (PC) register shifts to the befinning of the other task's instruction

Question 6 (60 points)
Consider a robot similar to the ones used in the SEG4145 labs. This robot has to perform the following path: *move forward 40 centimetres; turn 90 degrees clockwise; move backwards for 10 seconds; move in a square spiral assuming each move being linear, (i.e. the robot moves around a tile square of two feet the side); stop movement.*

You will have to build a UML state machine that models the behaviour described above, using **only** the sub-set of UML state machine specifications shown in Background material shown at the Appendix of this Assignment.

- The robot is initially stopped and it will only start moving when a button located on it is pressed.
- An external timer is available, so UML timeout events can be used.
- The following functions are implemented:

- **`move_forward()`** – Robot will start moving forward.
- **`move_backwards()`** – Robot will start moving backwards.
- **`turn_cw()`** – Robot will turn 90 degrees clockwise.
- **`stop()`** – Robot will cease movement.

- These three functions will return as soon as the signal to the motors is sent.
- There is a counter that measures the distance traveled (in centimetres), regardless of direction. Whenever the movement starts, this counter is reset. Whenever the value of this counter reaches 20, the counter is reset and an interrupt is generated.
- Once the robot has executed the path, it stops and pushing the button will **not** restart the path. In order to restart the path, the robot will have to be power-cycled.

In order to build the requested UML state machine, perform the following steps:

a) (20 points) Define and list all necessary UML events that will generate transitions between states of the UML state machine. For each event, specify the UML event type (i.e. signal, call event, timeout event).

b) (20 points) Build a list of states for the UML state machine. List all the states in the following manner: for each state, specify its name; the event that generates the transition out of the state; the guard condition (variable), if applicable, that must be observed for the transition to take place; and the name of the next state (i.e. the state that the system transitions into as a result of the transition). If more than one transition exists out of a given state, list all possible transitions out of that state. Use the following as a guideline when listing the states:

```
State name: ...
       Transition #1
                  Generating event: ...
                  Guard condition: ...
                  Next state: ...
       Transition #2
       ...
```

d) (20 points) Draw the resulting UML state machine. Draw separately the UML definition for any **signal** used in the state machine.

Answer to Question 6

a) Since the path that must be performed contains a 90-degree turn and there is no functionality specified that would allow the robot to turn, we can assume that there is a **turn_cw()** function that will make the robot turn clockwise by a specified number of degrees, and that returns once the turn has completed.

We can now proceed to list all the possible events and their types:

- **move_forward()** – call event

- **move_backwards()** – call event
- **turn_cw()** – call event
- **stop()** – call event
- **distance_int** – signal (this is the interrupt generated when the distance counter reaches its maximum value of 20 centimeters)
- ***tm*()** – generic timeout event

In addition, we assume the existence of a variable named **button_pressed**; this will be used as a guard condition that prevents the system exiting the **START** state before the button located on the robot is pressed.

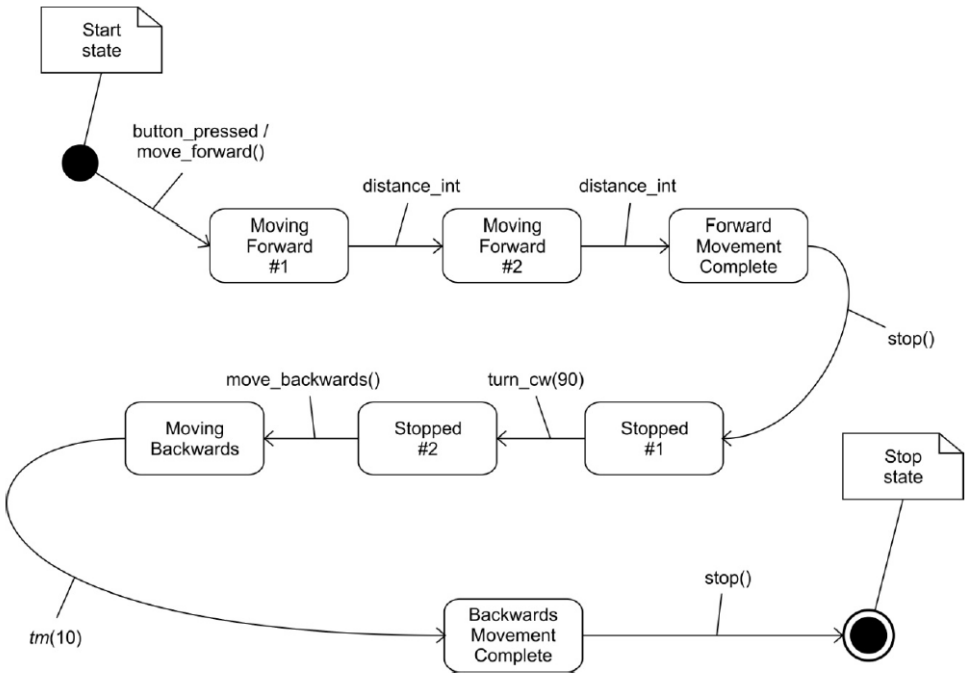
b) The states of this system are listed below.

- Name: **START**
    - Transition #1: •
        - Generating event: **move_forward()**
        - Guard condition: **button_pressed**
        - Next state: **Moving Forward #1**
-
- Name: **Moving Forward #1**
    - Transition #1:
        - Generating event: **distance_int**
        - • Guard condition: N/A
        - • Next state: **Moving Forward #2**

- Name: **Moving Forward #2**
    - Transition #1:
        - Generating event: **distance_int**
        - • Guard condition: N/A
        - • Next state: **Forward Movement Complete**
        -
- Name: **Forward Movement Complete**
    - Transition #1:
        - Generating event: **stop**
        - • Guard condition: N/A
        - • Next state: **Stopped #1**
        -
- Name: **Stopped # 1**
    - Transition #1:
        - Generating event: **stop**
        - • Guard condition: N/A
        - • Next state: **Stopped #1**
        -
- Name: **Stopped # 1**
    - Transition #1:
        - Generating event: **turn_cw(90)**

        - • Guard condition: N/A
        - • Next state: **Stopped #2**
        -
- Name: **Stopped # 2**
    - Transition #1:
        - Generating event: **move_backwards()**
        - • Guard condition: N/A
        - • Next state: **Moving Backwards**
        -
- Name: **Moving Backwards**
    - Transition #1:
        - Generating event: **tm(10)**
        - • Guard condition: N/A
        - • Next state: **Backwards Movement Complete**
        -
- Name: **Backwards Movement Complete**
    - Transition #1:
        - Generating event: **stop()**
        - • Guard condition: N/A
        - • Next state: **Stop**
        -
- Name: **STOP**

c) Based on the states listed above, the following UML state machine results:



There is only one signal event used in the state machine (distance_int).
Its definition is given below.