uOttawa
L'Université canadienne
Canada's university

SEG4145:
REAL-TIME AND EMBEDDED SOFTWARE DESIGN
WINTER 2018

SOLUTION TO ASSIGNMENT #2

Answer to Question 1

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 also called task though large Real-Time Processes (STP) can contain many tasks, 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. 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

Answers to Question 2 (20 points - choose one answer)

1. In real time operating system:
a) all processes have the same priority
b) a task must be serviced by its deadline period
c) process scheduling can be done only once
d) kernel is not required

Answer: b

2. Hard real time operating system has ___ jitter than a soft real time operating system.
a) less
b) more
c) equal
d) none of the mentioned

Answer: a

3. For real time operating systems, interrupt latency should be
a) minimal
b) maximum
c) zero
d) dependent on the scheduling

Answer: a

4. In operating system, each process has its own
a) address space and global variables
b) open files
c) pending alarms, signals and signal handlers
d) all of the mentioned

Answer: d

5. In Unix, Which system call creates the new process?
a) fork
b) create
c) new
d) none of the mentioned
Answer: a

6. What is the ready state of a process?
a) when process is scheduled to run after some execution
b) when process is unable to run until some task has been completed
c) when process is using the CPU
d) none of the mentioned

Answer: a

7. A Process Control Block(PCB) does not contain which of the following :
a) Code
b) Stack
c) Bootstrap program
d) Data

Answer: c

8. The state of a process is defined by:
a) the final activity of the process
b) the activity just executed by the process
c) the activity to next be executed by the process
d) the current activity of the process

Answer: d

9. Which of the following is not the state of a process?
a) New
b) Old
c) Waiting
d) Running

Answer: b

10. The _____ determines the cause of the interrupt, performs the necessary processing and executes a return from the interrupt instruction to return the CPU to the execution state prior to the interrupt.

a) interrupt request line
b) device driver
c) interrupt handler
d) all of the mentioned

Answer: c

Answer to Question 3

A process can contain threads, while threads are parts of the task execution that defines a process. As mentioned above in the Answer to Question 1, some operating systems define **kernel-level threads** and **user-level threads**.

To make threads cheap and fast, they need to be implemented at user level. User-Level threads are managed entirely by the run-time system (user-level library). As they are not kernel-level threads the kernel is unaware of them, so they are managed and scheduled in userspace. The kernel considers user-level threads as if they were single-threaded processes.
User-Level threads are small and fast, each thread is represented by a PC, register, stack, and small thread control block. Creating a new thread, switching between threads, and synchronizing threads is done via procedure call. i.e no kernel involvement. User-Level threads are hundred times faster than Kernel-Level threads, but cannot take advantage of multithreading or multiprocessing, and will get blocked if all of their associated kernel threads get blocked even if there are some user threads that are ready to run.

Some implementations base their user threads on top of several kernel threads, to benefit from multi-processor machines (M:N model).
Threads in the same process share the same address space.
This allows concurrently running code to couple tightly and conveniently exchange data without the overhead or complexity of an IPC.

When shared between threads, however, even simple data structures become prone to race conditions if they require more than one CPU instruction to update: two threads may end up attempting to update the data structure at the same time and find it unexpectedly changing underfoot. Bugs caused by race conditions can be very difficult to reproduce and isolate.

To prevent this, threading application programming interfaces (APIs) offer synchronization primitives such as mutexes to lock data structures against concurrent access. On uniprocessor systems, a thread running into a locked mutex must sleep and hence trigger a context switch. On multi-processor systems, the thread may instead poll the mutex in a spinlock. Both of these may sap performance and force processors in symmetric multiprocessing (SMP) systems to contend for the memory bus, especially if the granularity of the locking is fine.

Advantages:

1. The most obvious advantage of this technique is that a user-level threads package can be implemented on an Operating System that does not support threads.
2. User-level threads does not require modification to operating systems.
3. Simple Representation: Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
4. Simple Management: This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
5. Fast and Efficient: Thread switching is not much more expensive than a procedure call.

Disadvantages:

1. Since, User-Level threads are invisible to the OS they are not well integrated with the OS. As a result, Os can make poor decisions like scheduling a process with idle threads, blocking a process whose thread initiated an I/O even though the process has other threads that can run and unscheduling a process with a thread holding a lock. Solving this requires communication between kernel and user-level thread manager.
2. There is a lack of coordination between threads and operating system kernel. Therefore, process as whole gets one time slice irrespect of whether process has one thread or 1000 threads within. It is up to each thread to relinquish control to other threads.
3. User-level threads requires non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runable threads left in the processes. For example, if one thread causes a page fault, the process blocks.

Answer to Question 4

When a timer interrupt occurs the following sequence of steps are taken by the RTOS:

1. Timer Interrupt is called
2. A trap into kernel the space triggers the switching to kernel stack
3. Current application's registers are saved
4. The Scheduler provides the next thread that should be run
5. Context switch to other thread (load its stack, flush the TLB) is done
6. Load user registers from the kernel stack
7. PC register shifts to the beginning of the other application's instruction

Answer to Question 5

Semaphores work by blocking processes with P, calling thread_block(), waiting for a resource if it is not available, then being put into a queue of processes that want to

access this resource. This is more efficient than other solutions because we avoid busy waiting – other processes can do other things while blocked ones are in the queue! When a resource is released, V is run, which calls thread_wakeup() to signal the next thread to use it. See:

```
typedef struct _semaphore {
        int count;
        struct process *queue;
} semaphore;

semaphore sem;

void P(sem) {
          sem.count--;
          if (sem.count < 0) {            /* add process to sem.queue */
                  thread_block();
          }
}

void  V(sem) {
          sem.count++;
          if (sem.count <= 0) {          /* remove a process from sem.queue */
              thread_wakeup();
          }
}
```

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: **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**

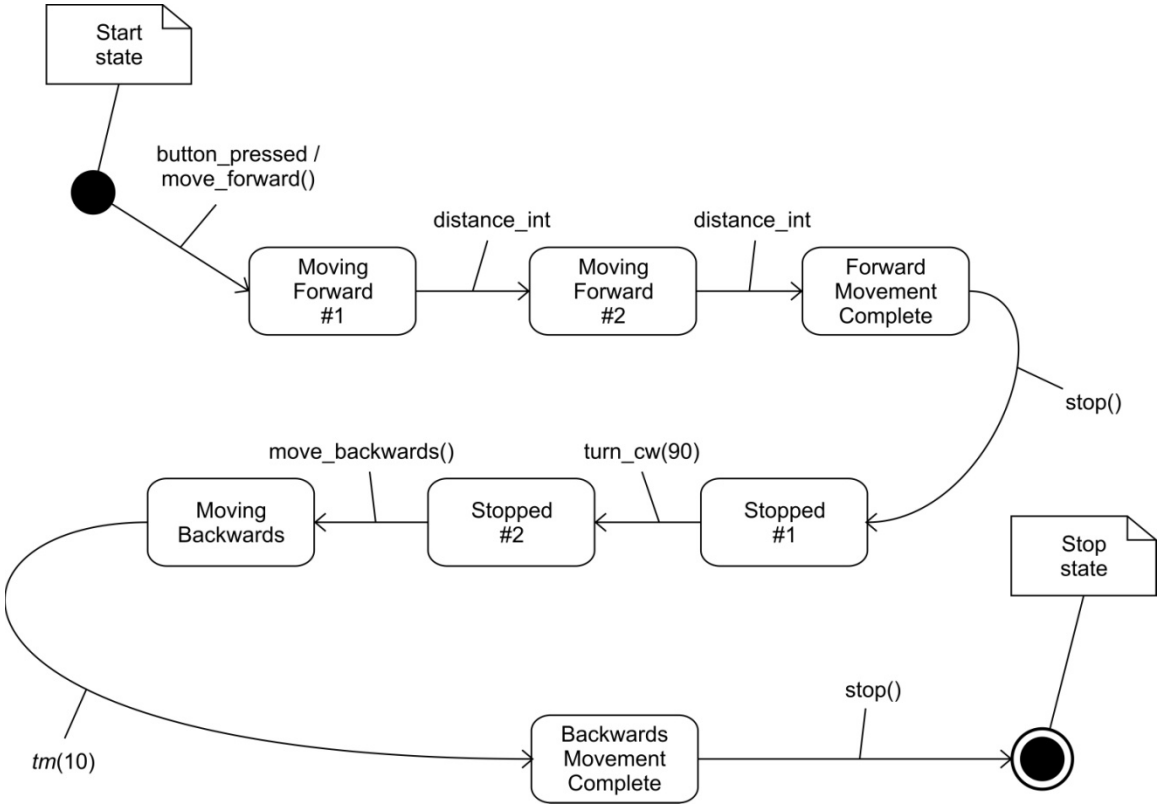
- Name: **Moving Backwards**

- Transition #1:
  - Generating event: ***tm*(10)**
  - Guard condition: N/A
  - Next state: **Backwards Movement Complete**

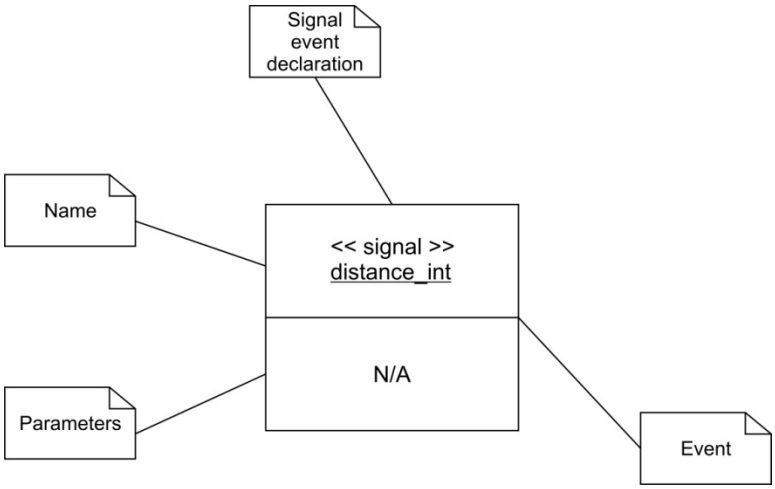
- Name: **Backwards Movement Complete**
  - Transition #1:
    - Generating event: **stop()**
    - Guard condition: N/A
    - Next state: **STOP**

- 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.


Signal event declaration
Name
<< signal >>
distance_int
N/A
Parameters
Event