IIA2017: Industrial IT

**Multitasking and real-time assignment**

Isak Skeie, 245362

Contents

Contents 2

1 Introduction 3

1.1 xxx 3

1.2 xxx etc 3

2 Chapter title 4

2.1 Subchapter title 4

2.1.1 Sub-subchapter title 4

3 Chapter title etc. 4

4 Conclusion 4

References 4

Appendices 4

# Introduction

<Type the introduction here; typical outline: Background – Objective(s) – Methods – Scope – Report structure info>

## Scheduler Setup

Graphical user interface

Description automatically generated

## Theory

1. The difference between Mutex and Semaphore is the usage in MultiTasking in programming. Both Mutex and Semaphore is used to allocate resources to tasks, with the difference being that a tasks uses Mutex to lock the task, while Semaphore is used to signal the use of resources to tasks. [Mutex vs Semaphore - GeeksforGeeks](https://www.geeksforgeeks.org/mutex-vs-semaphore/)
2. The function of mutex is to make sure a resource is only used by one task at time. This is achieved by the task using Mutex to lock the resource.
3. Without a way to control tasks use of resources, several tasks could try accessing one resource at the same time. This will cause issues with tasks writing over memory before its used by another task. As well as Deadlocking. [Mutexes and Semaphores Demystified (barrgroup.com)](https://barrgroup.com/embedded-systems/how-to/rtos-mutex-semaphore)
4. The main difference between a Process, Threads and Task is their relation to memory. A Process uses a Memory Management unit for memory access, so does the Thread, but with the addition that it shares data with its process. A task runs within a Thread, and doesn’t utilize MMU. Lecture notes.

Graphical user interface, text, application, email

Description automatically generated

# Evaluation Of a Multitasking system

<Start each chapter with a short introduction about what the chapter is about.>

## Analysis

The System Sequence Diagram shows a simple process for the threads to execute. With the shared resource being the communication with the console. This is used to display information for the user. This common resources doesn’t affect the running time in any significant way. The delay however, is the component deciding the running time.

To be able to share resources between threads, one could use Semaphores or Mutex. Theses components make sure resources are used correctly by Thread. With the addition of Thread.Sleep(), its possible for threads to share resources in runtime.

### Running Code 0

In Running Code 0, all the threads run in 137 iterations of a loop. Between each loop iteration, there is a delay in the threads, this is given from the parameters in the “MultiTasking and Realtime”-application. The Duration for each Thread is measured from the printed information and verified by multiplying number of loops by delay. Based on the output, its clear that all the threads start concurrently and finish in the order of the delay between the loops. With a noticeably difference in finish time, depending on the delay in the loops. The start and end of Code 0 is shown in figure # and #

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

Equation (2.1) is…

|  |  |
| --- | --- |
|  | (2.1) |

### Running Code 1

For Running code 1, the threads go through 137 iterations. For this run, the delay in the loops of the threads are disregarded. This is because the threads run in an ascending queue. The total run time of all the threads will in this case have a minimal difference. The start of the process is shown in figure #, and the end of the run in figure #

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

### Running Code 2

When running code 2, the process hangs, as seen in figure #. There could be several explanations as to why the process hangs. A higher priority task could be needing a semaphore that’s taken by a lower priority task. From the figure #, thread 1 is the last to be executed, another explanation is that this thread doesn’t release the Semaphore properly.

Graphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generated

## Conclusion on analysis results

When looking at the running time for each of the processes. Its clear that its determined by the delay given for each of the threads. With the run time for Running code 0 and Running code 1 being the same. The synchronization used in Running code 1 makes better use of the resources and the execution better for the user, with all of the threads starting and stopping concurrently. When it comes to Running code 2, the synchronization of the threads fault, making the code go into a Deadlock.

# Development of a multitasking system

The code from the lectures # is used source code. When running this code, it acts the same way as Running Code 0. With the delays in the threads deciding when to print information. By using Semaphores, the goal is to make the code act more like Running Code 1. This means that the threads print out their information concurrently. Disregarding the delay that’s been added. This is done by added a Semaphore that can be occupied by only one Thread at a time. The Semaphore is closed when the thread is initialed, and closed when the initialization is over. It will be closed again when each of the threads enters their loop for writing to the console. And released at the end of the loop. By using Semaphores on initialization, as well as Thread.Sleep(), the threads are queued in the right order for them to execute ascending. The finished code can be viewed in appendix A.

|  |
| --- |
| Initialize static Sephamore;  ThreadClass  {  Initialize integer for loop count;  Initialize integer for delay;  Initialize thread class;  ThreadClass constructor(name, delay)  {  Initialize integer LoopCount = 0;  Sets delay equal to input delay;  Sephamore.WaitOne():  Initializes thread;  Sets name of thread;  }  Run Method()  {  Semaphore release;  Write to console starting thread;  Do While  {  Semaphore WaitOne;  Loop count ++;  Thread sleep with delay;  Semaphore release;  Write Thread name and loop;  }  Write ending of thread;  }  }  Class program  {  Static void main()  {  Write starting program;  Create array of thread classes;  Create string array of thread names;  Create int array of thread loop delay;  Loop that instantiate array of thread classes;  Loop through a count with Thread.Sleep(0);  Write end of program;  }  } |

# Time Requirements for a real-time system

Based on the requirements given in the Assignment Description, a maximum response time is needed for the system. The requirement given is a maximin latency of 250ms from energization of level input, until the pump acts. When the pump receives a signal to change state, it takes 200ms until the process is finished. This leaves the application a window of 50ms to receive a signal from one of the switches, until a control signal is sent to the Pump. Within 50ms the application needs to receive the signal interrupt, run Interrupt Service Routine, run the scheduler, and then run the task for the pump.

Based on the given equation, all of the RTOS fulfill the time requirement, although with different margins. For this use case, RTOS 1 would be a good choice, it doesn’t have the highest margins in milliseconds, nor the highest number of maximum tasks. But it has a low Context switch time, and low instruction time. This is assumed favorable for the speed of the system, over the scalability which is assumed, will stay at a minimum. The timing diagram for RTOS 1 is shown in figure #

Chart

Description automatically generated

# Conclusion

References

1. <Reference 1>
2. <Reference 2> etc

Appendices

Appendix A <Appendix title>

Appendix B <Appendix title>

Appendix C <Appendix title> etc.