CM0605 - Embedded Systems Engineering

Liam Brand

Chapter 1

Real Time Scheduling

1.0.1 Basic Scheduling Analysis

(a)

The utilisation of a task set is given by

$$U = \sum_{i=1}^{N} \frac{C_\mathrm{i}}{T_\mathrm{i}}$$

Where N is the number of tasks, C is the task's execution time and T is the task's period. The given task set's utilization is approximately 0.99.

(b)

A set of given tasks can be proved to be or not be schedulable using rate-monotonic priority assignment by using the utilisation-bound theorem. If the test is positive and the CPU utilisation is below the given bound, the set is schedulable.

$$U = U = \sum_{i=1}^{N} \frac{C_\mathrm{i}}{T_\mathrm{i}} \leq n(2^{\frac{1}{n}} - 1)$$

As previously discussed, the set's CPU utilisation is around 0.99. The given task set has 6 tasks, so the utilisation bound is around 0.73. This test is therefore false, meaning the task set is not schedulable using rate-monotonic priority assignment.

(c)

First the WCRT (worst case response time) for the task set must be calculated. This can be achieved via the given formula.

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Where R is response time, C is CPU utilisation, and hp(i) denotes the set of tasks of priority higher than i.

This calculation is repeated for each task until the calculate task response time converges twice. The following shows each iteration of this formula being applied to the task set.

Iteration 1
E(C=14, T=48, D=48): r = 14
C(C=6, T=240, D=240): r = 20
A(C=42, T=330, D=60): r = 76
D(C=132, T=360, D=300): r = 250
B(C=72, T=720, D=660): r = 342
F(C=18, T=4800, D=120): r = 360

Iteration 2
E(C=14, T=48, D=48): r = 14
C(C=6, T=240, D=240): r = 20
A(C=42, T=330, D=60): r = 76
D(C=132, T=360, D=300): r = 270
B(C=72, T=720, D=660): r = 412
F(C=18, T=4800, D=120): r = 430

Iteration 3
E(C=14, T=48, D=48): r = 14
C(C=6, T=240, D=240): r = 20
A(C=42, T=330, D=60): r = 76
D(C=132, T=360, D=300): r = 270
B(C=72, T=720, D=660): r = 558
F(C=18, T=4800, D=120): r = 576

Iteration 4
E(C=14, T=48, D=48): r = 14
C(C=6, T=240, D=240): r = 20
A(C=42, T=330, D=60): r = 76
D(C=132, T=360, D=300): r = 270
B(C=72, T=720, D=660): r = 606
F(C=18, T=4800, D=120): r = 624

Iteration 5
E(C=14, T=48, D=48): r = 14
C(C=6, T=240, D=240): r = 20
A(C=42, T=330, D=60): r = 76
D(C=132, T=360, D=300): r = 270
B(C=72, T=720, D=660): r = 620
F(C=18, T=4800, D=120): r = 638

Iteration 6
E(C=14, T=48, D=48): r = 14
C(C=6, T=240, D=240): r = 20
A(C=42, T=330, D=60): r = 76
D(C=132, T=360, D=300): r = 270
B(C=72, T=720, D=660): r = 620
F(C=18, T=4800, D=120): r = 652

Iteration 7
E(C=14, T=48, D=48): r = 14
C(C=6, T=240, D=240): r = 20
A(C=42, T=330, D=60): r = 76
D(C=132, T=360, D=300): r = 270
B(C=72, T=720, D=660): r = 620
F(C=18, T=4800, D=120): r = 652

The final iteration shows that the task set is schedulable, as each of the task periods are less than or equal to the task response times. However, tasks A and F have a response time that exceeds their deadline, so these tasks will not meet their deadline.

(d)

The same calculations were performed as the ones in (c), but this time the tasks were organised by deadline (lowest to highest) instead of by their period. As the technique and method for calculation has already been explained and demonstrated, only the final iteration will be shown here.

Iteration 7
E(C=14, T=48, D=48): r = 14
A(C=42, T=330, D=60): r = 70
F(C=18, T=4800, D=120): r = 88
C(C=6, T=240, D=240): r = 94
D(C=132, T=360, D=300): r = 288
B(C=72, T=720, D=660): r = 652

The final iteration shows that each task's response time satisfies the condition of being less than or equal to the task period, so the set is schedulable

using deadline-monotonic fixed-priority assignment. Task A's response time is higher than its deadline however, so it will not meet its deadline.

(e)

Files are an example of a mutual exclusive resource, as only one task can be accessing a file at any given time to perform read or write operations on it. Another example is components like hardware timers, as if multiple tasks want to use the timer there will need to be controlled access to it.

Priority Inversion

An example which exhibits priority inversion can be seen with the Mars Pathfinder. The Pathfinder spacecraft has an information bus which was used to exchange information between the robot's various components. A high priority task ran frequently to manage the data on this bus, and access to the bus was synchronised with a mutex. The Pathfinder had another task for gathering meteorological data, which ran infrequently and had a low priority. This task would acquire the bus' mutex before writing data to the bus. If an interrupt caused the information bus to be scheduled whilst the meteorologic task held it, the bus management task would pend on the mutex and wait for the meteorologic task to be done. There was also a task with medium priority that dealt with communications.

The problem with priority inversion manifested like so. An interrupt would sometimes occur that scheduled the medium priority communications thread to be scheduled during the small moments where the high priority bus thread was blocked waiting on the meteorological data thread. When this happened, the communications task would prevent the meteorological task from running which would in turn prevent the communication task from running, meaning the medium priority task was essentially blocking the high priority task[3].

Priority Inheritance

Priority inheritance can help prevent this. With priority inheritance, tasks that block higher priority tasks will inherit the higher task's priority for the duration of the blocking. Once the task has finished executing and has released the resource it will drop down to its original priority asignment, and allow the high-priority task to use the resource that it has just acquired. For the Pathfinder example, this would mean the meteorologic thread would escalate to the bus management task's priority meaning the communication's task could not acquire the mutex and block the bus management task once the meteorological task had finished executing.

Priority Inheritance Limitations

Transitive blocking can occur. This is when tasks all begin blocking eachother and acquire high importance. For example, there are three tasks with high, medium and low priority, and two resources called R1 and R2. The low task takes R1 and the medium task takes R2. If the medium task wants to take R1, it is blocked and the low tas's priority is set to medium. If the high task then wants to take R2, it is blocked and sets the medium task to high priority. The

task holding R1 has an unchanged priority here even though it is the reason the original high priority task is being blocked.

Priority inheritance also does not prevent deadlocks, where all tasks aer stuck waiting for other tasks to release resources preventing them from releasing resources that other tasks need and so on.

As well, priority inheritance might prevent priority inversion but it doesn't minimize the time that tasks spend being blocked.

An Alternative

1.0.2 Scheduling with Shared Resources

Question a

Question b

Chapter 2

A Distributed Real-time System

2.1 Question 1

For both systems, the CAN controller was initialised with a baud rate of 125000. The Display system would be event driven, responding to interrupts generated by events. The engine monitor would be a time-triggered system, with the interrupts being generated from a timer.

2.1.1 Display

For the Display system, two threads are used. These threads are assigned priorities and tasks for them to run, with one thread running the request task and the other running the display task. Of the two threads one of them had to have a slightly priority. The request thread was chosen for this as it was deemed important that it was regularly able to request information from the EngineMonitor, more so than the ability to display it.

The request thread uses a wait command to run at intervals of 500ms, and transmits a simple CAN message requesting a temperature reading to the EngineMonitor each time it runs. The message is a canMessage_t struct, a simple structure containing variables like the message's ID, its length and some data.

The full request task can be seen below.

```
static void requestTask(void) {
   static canMessage_t requestMsg = {0x23, 8, 0, 0};
   bool txOk;

   while(true) {
      txOk = canWrite(&requestMsg);
      wait_ms(500);
   }
```

```
}
```

One aspect in particular that took some planning was allowing the display task to be event driven. Rather than running at an interval it had to execute and display a value to the terminal whenever it was received a temperature reading.

To achieve this a combination of an interrupt handler and a semaphore is used. The display task pends on a semaphore that is released by a method that runs when a CAN message is received.

```
/* A simple interrupt handler for CAN message reception */
void canHandler(void) {
   canTransferRxFrame(&rxMsg);
   rxDone.release();
}
```

This interrupt handler transfers the most recently received CAN rxFrame to the rxMsg variable, and then releases the semaphore that the display task is pending on. Once the semaphore is released, the rest of the display code runs where it prints the data from the received message to a terminal. After the wait period has elapsed, it begins and pends on the semaphore once again. The full display task code can be seen below.

```
/* Pend on a semaphore released by an RX Interrupt
* When released, display the recieved temperature reading to a
    terminal
*/
static void displayTask(void) {
   canRxInterrupt(canHandler);
   while(true) {
      rxDone.wait();
      pc.printf("Temperature: 0x%08lx\n", rxMsg.dataB);
   }
}
```

2.1.2 EngineMonitor

The EngineMonitor consists of two tasks. A tacho task which simulates monitoring the engine's rotation speed, and a temperature task which sends a response to the Display's request for a temperature reading. A time triggered scheduler is used to run the EngineMonitor tasks.

The scheduler is relatively simple, it uses a task table containing an array of task control blocks. Each block has a pointer to the task, a delay, and a

period. The scheduler initializes by iterating through the task table and setting all values to zero, and all task pointer to NULL. Tasks are added to the scheduler by through the schAddTask method, which takes parameters corresponding to the variables in the task control blocks. The scheduler is then started, which begins a simple timer that starts ticking. The scheduler is then dispatched each tick, running tasks that delays have reached zero and resetting the delay's to the task's period.

The code in the EngineMonitor's main method that intialises the timer, adds the temperature and tacho tasks, then starts and repeatedly dispatches the timer can be seen below.

```
schInit();
schAddTask(tachoT, 0, 20);
schAddTask(tempT, 0, 500);

schStart();

while (true) {
   schDispatch();
}
```

The tacho task simulates measuring the engine's rotation speed by reading some values from the K64F's accelerometer, and assigning these values to a dummy variable which isn't used anywhere else in the program.

When the temperature task runs, it waits on the canReady method returning true. canReady is a simple method that returns true if there is data ready to be read from the CAN port. If the method returns true, the task uses canRead to read the message into a canMessage_t variable.

```
if(canReady()){
   canRead(&canData);
   id = canData.id;
}
```

Following this, the message's id is checked. Rather than simply responding once any message is received, an id of 0x24 is used for all request messages, and should the received message match this id it will be acknowledged as originating from the display. Once this acknowledgment has been done, values are collected from the accelerometer to simulate taking a temperature reading, they are put into a can message, and the message is then written to the CAN controller.

```
if(id == 0x24){
   canMessage_t txMsg = {0x23, 8, acc.getY(raY), acc.getY(raY)};
   bool txOk;
   pc.printf("ID 0x%1x", canData.id);
```

```
    txOk = canWrite(&txMsg);
}
```

2.1.3 System Evaluation

The system's structure and timing requirements have been fulfilled. Both the display and engine monitor systems posses the correct structure, in that they have appropriate tasks that perform the desired functions, and the tasks in both run in their respectively specified event driven and time-triggered ways. The tasks also run at their specified time periods, with the request task running ever 500ms and the tacho task running every 20ms. The temp task only runs in response to a request, and the request it receives is checked before a response is sent rather than simply sending a request as soon as any message is received.

This structure affords them some degree of precision. Both systems could achieve their functionality with a super-loop, with large looping main tasks that perform certain functionalities each loop. The problem with this would be in its lack of precision and efficiency however. By using an event driven system for the display, it can be ensured that the display responds to an event as soon as it is received rather than an event occurring and then the potential for a small wasted interval to occur whilst it loops back around to the relevant code. With the engine monitor, a time triggered system helps ensure that tasks run closer to their correct time, as in a super loop there's no guarantee that all other code in the loop will take the same amount of time to execute each loop. The system being set up the way it is helps minimize the intervals between necessary functionality.

The system has no functionality in place to deal with a message that hasn't successfully been sent however. The canWrite method returns a boolean value depending on whether or not the message has been successfully sent (true if it has, false if it hasn't), and this value is never used for anything. The system could be improved by checking this value once the canRead method is called, and if it returns false it could begin a retry loop where it attempts to send the message again. This loop would continue until the next time the task this is in has to run (e.g. until the next new request message needs to be sent from the display), and should the message never be sent this could flag up as a missed message. Repeated missed messages could be flagged up as a more serious problem. As of right now, the system is massively dependent on all messages successfully sending, with no warning of requests or temperature acknowledgements that haven't been able to be sent.

The display system also performs all display functionality in the interrupt service routine. Whilst this doesn't seem to have any adverse effects yet, should the system expand and begin doing something more elaborate with received temperature data such as printing to an attached LCD, doing everything in the ISR might start to impede performance. To help with this, an EventQueue

could be used. The Mbed RTOS features an EventQueue API[1] that can be used to store events and defer the execution of the event code for a different context. For the display's ISR, the EventQueue could store interrupts containing temperature values and execute them outside of the interrupt context to prevent the interrupt blocking other tasks for too long.

2.2 Question 2

2.3 Question 3

2.4 Question 4

Chapter 3

Reliability

3.1 Section 1

3.1.1 (a)

Fail-Operational

Fail-operational is when a system is still capable of full performance in the presence of faults, with no external signs of the fault manifesting. For the car production line, this would mean a system fault would still result in cars being produced at a good efficiency, meaning the system's objectives to produce cars won't be impeded and as such neither will car production. The encountered fault might cause an issue where damage is done to cars however, so whilst the production might not fail the actual quality of the cars produced would. As well, there would be no guarantee for the system to be entirely safe during its operation here, so there would be a risk to personnel safety.

Fail-Active

Fail-active systems can continue their operation when a fault is encountered, but will do so at a reduced performance. This is 'graceful degradation' where as more parts of the system experience faults the system can still continue its operation (up until a certain point). With regards to the production line, this will likely involve car production continuing at a reduced rate until the fault can be corrected. This won't always be the case however, as the fault may lie in a specific part of the car assembly. If this was the case, reduced system performance isn't tolerable as a key part of the manufacturing process won't be completed, so the cars being produced here won't be of any use.

Fail-Safe

Fail-safe systems will cease operation upon encountering a fault, and enter a safe mode. For example, elevators are often equipped with breaks that will apply

should the cable holding the elevator snap. For the car production line, this will mean the entire system ceases operation in the event of a fault. This will completely minimize risk to human life but production will suffer the most as each encountered fault will result in the system entering its safe mode. Restarting the system and bringing it back to full operational capacity could take a while, and this will result in a potential failure of the system's mission objectives.

High Availability

High availability systems cease operation upon encountering a fault, but must be returned to operation as quick as possible, this can involve hot swapping system units whilst the system is operating. These systems allow for failures, and aim to achieve a high mean time of operation rather than a long continuous time of operation. The goal here is not to avoid faults, but to minimize time spent rectifying them so that the overall system operation time is as high as possible[2]. The availability of these system can be measured by the following, where MTBF is mean time between failures and MTTR is mean time to repair.

$$Availability = MTBF/(MTBF + MTTR)$$

Out of the different types of fault-tolerance, high-availability would be the most appropriate for the car assembly line. Whilst this will result in a lower amount of continuous operation time, as long as the mean operation is as high as possible (which high-availability aims to achieve) production goals should still be satisfied and the production line's mission shouldn't be impeded. High-availability also allows for shutdowns when faults are encountered, which is an imperative feature of the assembly line's fault tolerance given the potential for harm to personnel.

3.1.2 (b)

Potential Risks

The nature of the car assembly line means any single failure will cause the entire system to fail, so in terms of the network only a single connected node in the distributed system needs to experience a problem for the entire system to stop working.

Timing will also be essential within the assembly line to ensure the robotic tools perform their jobs properly, so there is a risk of performance overhead negatively impacting the system too. Networked processors that are in close proximity will communicate incredibly quickly, but processors that are further away from the network will take longer to communicate and this will need to be accounted for otherwise tools may not behave as expected and, should any issues occur, the entire system will fail as previously discussed. In a similar vain, timing will need to be considered to ensure that all operations across the network are in synchronisation with each other. Messages arriving on time is a

good start, but this doesn't guarantee that all of the production line robotics are cooperating as expected.

There is also the possibility for the system's data to encounter issues, such as addressing errors where nodes send or receive messages that are intended for a different node. The data itself could also be corrupted during its transmission.

Architecture Recommendation

3.1.3 (c)

Power-on risks with the production line's hardware will need to be considered. The component's ROM contents could become corrupted by electromagnetic pulses, or the component's flash memory could be reprogrammed and cause issues (e.g. maybe parts of the memory are being incorrectly erased). These issues can be solved during the component's power-up phase by having the component run checks to ensure the validity of its state.

Run-time issues are also a possibility. Like with the start up problem, electrical noise or a power supply fluctuation could cause the program the component is running to become corrupted. Check summing could be used here, where the component's program checks its validity against a checksum to determine if the program is in a proper state. Program copies could be used that have also been check summed, with components running the check summed copies should the program running be found to be corrupt.

One hardware risk with the production line components is the contents of the ROM becoming corrupted by electronic pulses or radiation.

Bibliography

[1] arm MBED. *The EventQueue API*. Available at: `https://os.mbed.com/docs/mbed-os/v5.12/tutorials/the-eventqueue-api.html`.

[2] Jim Gray and Daniel P. Siewiorek. High-availability computer systems. *Computer*, 24(9):39–48, 1991.

[3] Mike Jones. What really happened on mars rover pathfinder. *The Risks Digest*, 19(49):1–2, 1997.