Assignment 2

UBC | EECE 494 | Spring 2010

Released on January 25
Due on February 12

1 Introduction

This assignment has two different aspects, each of which will give you a broad understanding of analysis issues and implementation issues. Remember that all programs should run on `ssh-linux.ece.ubc.ca`.

1.1 Schedulability analysis

To get a first-hand understanding of the different methods for schedulability analysis, you will implement the different analysis algorithms and compare their performance. In particular, you will study the performance of schedulability analyses for static priority scheduling. You will also learn how to generate task sets for unbiased comparisons.

1.2 Real-time communication

Many real-time systems are distributed systems with tasks that flow through multiple stages. The time to transfer the state (or result) of a task from one stage to the next may take non-negligible amount of time. One method for modeling this communication delay is to add a constant (an upper-bound on the communication time) for every communication involved: this will account for the worst-case communication overhead and if the end-to-end response time is less than the deadline of the task, we can feel assured that the task will meet its deadline.

An alternative approach, which is more realistic because communication delays need not be constant, is to model the communication medium itself as a stage in the distributed system. The communication channel (prioritized ethernet, ATM, fiber channel, copper bus, etc.) can be treated as another processor and, depending on the message scheduling policy, we can estimate the worst-case response time for a task. Our schedulability analysis will then include the communication overhead as well. It is, therefore, important to understand how the communication is being arbitrated, and how we can implement efficient communication mechanisms to reduce overhead from this component of a real-time system.

Network switches. A network switch is a real-time system that needs to provide low-latency communication. Consider a network switch that is responsible for routing packets from some number of input ports to some number of output ports. The switch has to read each packet, extract the destination from the header, figure out to what port the packet should be sent (based on the destination address and some sort of lookup-table in the switch), and send the packet to that port, possibly buffering the packet if that port is busy. Switches arrive quickly on multiple ports; if the switch is not fast enough, packets will be dropped, and the performance of the network will degrade. These switches are often designed in hardware (depending on when you took

EECE 353/379, you might have done this), however, increasingly, a processor running software is being used. In this assignment, you will design the software for a simple switch (Figure 1).

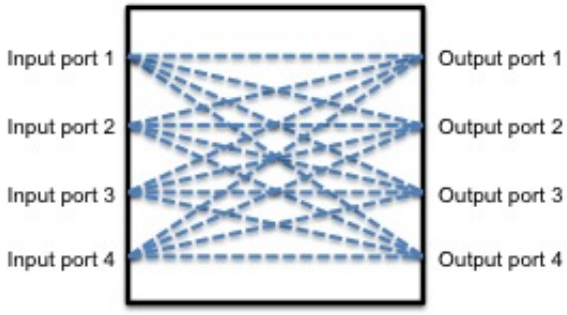

Figure 1: 4-port switch

As shown in the diagram (Figure 1), the switch has four input ports and four output ports. Each packet arriving on any port contains both a destination IP address, and a payload. A destination IP address is of the form: aaa.bbb.ccc.ddd, where each component is an 8-bit number. So for example, a destination IP address might be 137.82.52.165. In our implementation, the payload will be a simple integer.

When the switch receives a packet on one of its input ports, it will look in a routing table to determine the output port the packet should be sent to. The routing table contains a list of all the destination IP addresses that it knows about, and for each destination IP address, the output port that should be used. For example, if a packet with a destination IP address of 137.82.52.165 arrives, the switch would look for an entry in the routing table corresponding to 137.82.52.165, and would read the output port corresponding to that entry. It would then send the packet to that output port. Note that even though there are four input ports, there is only one routing table. This is important, because routing tables are big, and we don't want to store this information four times. In real switches, the routing tables can either be updated on-line (in which case entries are added as the switch operates) or off-line (in which case the routing tables are loaded before the switch starts to operate). We will use an off-line implementation in this assignment.

You will first design a content-addressable memory (CAM) to handle the routing table and then use this CAM to build the software required for a network switch.

2 Programming assignment

1. **Schedulability analysis [25 points]**

   Write a program to analyze the schedulability of a given task set using the deadline monotonic scheduling policy. Write this program in the subdirectory `hw2/task1` within your `eece494` directory. The program should perform the following steps:

   (a) Ask the user to enter the number of tasks.

   (b) Ask the user to enter the parameters for each task, specifically the execution time, the relative deadline and the period (**in this order!**). Inputs will proceed task by task, for example:

```
Enter the parameters for Task 1:  3 10 15
Enter the parameters for Task 2:  7 20 20
...
```

(c) Use three tests: the Liu & Layland bound, the hyperbolic bound and the response time (exact) test. When the bounds cannot be used state `not applicable`. For each of the tests, when applicable, indicate the time taken to verify schedulability. You will have to think about how to determine the timing (see the examples). For a given task set, repeat each test 1000 times and report the *average* schedulability test time. Use the following example as a template:

```
Enter the number of tasks:  2
Enter the parameters for Task 1:  3 10 15
Enter the parameters for Task 2:  7 20 20
Liu and Layland bound:  not applicable (time:  0 ms)
Hyperbolic bound:  not applicable (time:  0 ms)
Response time analysis:  schedulable (time:  0.5 ms)
```

This part of the assignment will be graded by feeding 10 problem instances to the schedulability analyser and counting the number of correct answers. The same 10 problem instances will be used for every submission.

Include a `makefile` that builds an executable file for this analysis; the executable file should have the name `schedulability`.

Grading rubric: 22 points for a correct implementation of the schedulability tests; 3 points for the `makefile`*, etc.*

2. **Generating synthetic workloads [25 points]**

We have examined the worst-case schedulability bounds for rate monotonic scheduling (and some other policies) in class. The worst-case behaviour is important because it helps us provision systems that are safety-critical and cannot afford any deadline misses. In this task you will study, in some more detail, the behaviour of the rate monotonic policy when scheduling tasks that have relative deadlines equal to their periods. The worst case, however, depends on the task periods. In the derivation of the Liu and Layland bound, we assumed that we know nothing about the execution times and periods and obtained the barely schedulable task set with lowest utilization. In some cases, we may know the period values and not the computation times; in these cases we can actually do better than the Liu and Layland test (and the Hyperbolic test). Further, the Liu and Layland bound assures us that no task set with utilization less than $\ln 2$ will miss deadlines. Tasks with utilization greater than $\ln 2$ may miss deadlines but we do not know how often this occurs.

Empirical testing is useful if we want to make better use of resources, especially when workload may vary, and when systems are soft real-time systems. For instance, suppose we choose a bound of 0.8 and only 1% of task sets (the number of tasks in a task set could be arbitrarily high) with utilization less than 0.8 miss deadlines, then we have improved utilization by 11% (from the Liu and Layland bound) and provide a small deadline miss probability. To actually determine what fraction of task sets miss deadlines at a certain utilization level, we would normally fix the utilization level at U. Then we would generate many synthetic task sets (tasks would have random execution times and random periods) that have utilization of U, and determine (using the exact test for schedulability) how many of these task sets are able to meet their deadlines using the rate monotonic scheduling policy. However, the manner in which task parameters are generated may significantly affect the result and bias the judgement about the schedulability tests. Ideally, for a fixed task set utilization U, we would like to generate $< U_i >$, which is the vector of task utilizations, such that $< U_i >$ is drawn uniformly at random from the set of all vectors $< U_j >$ with $\sum_j Uj = U$. We can then generate the task periods P_i using a uniform distribution over $(0, P_{max}]$, and the corresponding execution times would be $e_i = U_i \times P_i$.

For this task: Suggest an optimal scheme for generating task utilizations assuming that the there are n tasks in a task set and that the utilization of the task set is fixed at U. Compare your optimal scheme with at least one other scheme for generating workload. Show that there may be a bias on other schemes. A better scheme would have a more uniform distribution of utilizations. As an example,

consider the case when there are exactly two tasks, T_1 and T_2, and the total utilization is fixed at U. We can consider a two dimensional graph with the x-axis indicating the utilization of T_1 and the y-axis indicating the utilization of T_2. Each (x, y) coordinate represents a pair of utilizations. The straight line joining $(0, U)$ and $(U, 0)$ represents the set of all utilizations with sum U. The better scheme would generate points that are uniformly distributed on this line. A weaker scheme might result in clustering along this line. Explain why one scheme may be better than the other, and illustrate with a graph for the case with two tasks and fixed utilizations of $U = 0.5, U = 0.6$ and $U = 0.7$. (You may plot all these results on one graph.)

Set $P_{max} = 10000$ and start with $U = 0.68$. (At $U = 0.68$ you should see no deadline misses.) Increment U in steps of 0.04 and determine the fraction of task sets (generate 25000 task sets at each utilization level; assume the number of tasks is 20; use the 'better' scheme from the previous part) that are schedulable at each utilization level using the rate monotonic scheduling policy. Use the exact test to determine schedulability. (You can use the Hyperbolic bound as an initial test, and if a task set fails the Hyperbolic bound test then you can resort to the exact test.) Illustrate the change in the fraction of schedulable task sets on a graph. The fraction of schedulable task sets provides some insight into the average-case behaviour of rate monotonic scheduling. What can you infer about the rate monotonic scheduling policy based on this empirical analysis?

For this part of the assignment, you must submit your code for generating synthetic workload, a *clear* writeup describing the schemes you implemented, a graph showing the efficacy of your synthetic workload generation (task sets with two tasks suffice) and another graph illustrating the fraction of schedulable task sets using rate monotonic scheduling. All the relevant files should be in the subdirectory `~/eece494/hw2/task2`. The description of your methods and the graphs should all appear in one file named `report-wgen.pdf` and the code for your preferred workload generation method should be in a file name `wgen.cc` or `wgen.cpp`. Add a `makefile` that would allow you to build the executable binary file for workload generation; the executable file should be named `wgen`.

Grading rubric: 12 points for an optimal workload generation scheme; 6 points for the analysis of the typical case schedulability of rate monotonic scheduling (i.e., fraction of schedulable task sets); 5 points for a well-written report; 2 points for the `makefile`*, etc.*

3. **Optimizations for a network switch [50 points]**

   (a) **Obtaining the skeleton code.** Download the tarball for HW2 from the course webpage and extract it in your `eece494` directory.

   (b) **Building the routing table.** All the work for this task will be stored in the directory `~/eece494/hw2/task3b`. The most important part of the switch is the routing table. As explained earlier, the routing table contains a number of entries. Each entry contains a destination IP Address and an output port number. In this task, you will design and implement an efficient routing table. In addition to the data structure, you will provide two access routines, one to write an entry to the routing table and one to perform a look-up into the routing table. The time to add an entry to the routing table is not important (since it is done off-line), however, the time to perform a lookup into the table is critical, and will directly affect the real-time performance of your system.

   There is ample room for you to be creative in the design of your routing table data structure. The simplest (and infeasible) implementation would be a single array. Since there are 256^4 possible destination IP addresses, the simplest implementation would contain 256^4 entries, as shown in the diagram below. When a packet arrives, it would be a simple matter to lookup the proper output port by accessing the corresponding entry in the table. The problem with this approach is that 256^4 is a *large* number. The amount of memory required to implement such a table is well beyond what you might find in such a system.

   The implementation can be improved significantly by realizing that at any one time, only a very small subset of the 2564 destination addresses are known, and hence need to be stored in the table. Thus, it would be possible to create a table with n entries, where each entry would contain

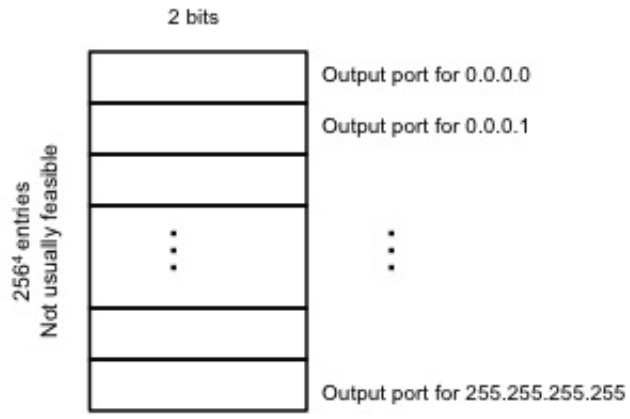

Figure 2: A *huge* routing table

a single destination IP address and an output port, where n is the maximum number of destination IP addresses we expect to know about at any one time. In this example, the table has been loaded with three IP addresses. Whenever a packet arrives with an IP address of 137.82.51.120, it is routed to output port 1. Whenever a packet arrives with an IP address of 126.24.29.108, it is routed to output port 0. If a packet with any other IP address arrives, it is dropped. This implementation (Figure 3) is easy to implement, can be efficient in memory, and it is fast to add an entry to the table. When a new destination IP address is added to the table, it is simply added to the next available entry (assuming fewer than n entries are already used). During a lookup, however, all entries have to be searched to find the proper destination IP address (actually, on average, half of the entries have to be searched).

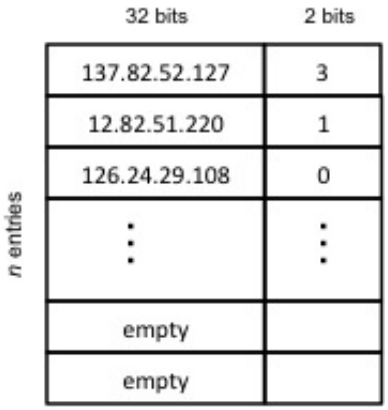

Figure 3: A more compact routing table

The first implementation requires to much storage, and a lookup operation is too slow in the second implementation. In this task you have to be creative and devise a new implementation that allows for fast lookup, yet does not require too much storage (hint: think "hash tables"). You can assume that no more than 400,000 IP addresses are ever stored in the table (so in the above example, $n = 400,000$). Rarely will the table be so full, however. If it matters, you wish to optimize your performance for the case when there are 40,000 entries in the table (so roughly 1/10 of the entries are actually used). For some implementations this will matter, for others it

will not.

The files you copied contain stub-files you can use to implement and test your routing table. Your data structure as well as the implementation of the following four routines should be added to the files `cam.c` and `cam.h`:

`void cam_init()` Initializes the routing table data structure. May be an empty routine or may contain code (depending on your implementation of the table).

`void cam_add_entry(ip_address_t *address, int port)` This routine adds an entry to the routing table. The rst parameter is a pointer to an`ip_address_t` data structure (dened in `misc.h`) which contains the IP address of the new entry. The second parameter is an integer that contains the output port corresponding to this IP address. The time to perform this operation is not critical.

`int cam_lookup_address(ip_address_t *address)` Given a pointer to an IP address, this routine looks for an entry with that IP address in the table. If it is found, the output port corresponding to that IP address is returned. If it is not found, a -1 is returned. This routine needs to be as fast as possible.

`void cam_free( )` This routine frees any memory that might have been allocated for the data structure. Depending on how you implement your data structure, this routine might contain no code.

The file `main.c` contains some code to test your routines. This code simply fills the table with 40,000 randomly generated IP addresses, and then performs a number of random lookup operations. The average time for a lookup operation is recorded. Your code should all be placed in `cam.c` and `cam.h`. You should not have to modify any of the other code.

Testing and write-up. In the same directory (`task3b`), include a file with the name `results.txt` (text version) or `results.pdf` (pdf version) containing answers to the following questions:

i. A short description of how you implemented your data structure. Often a picture is worth a thousand words. [5 points]

ii. The average time to perform a lookup. [2 points]

iii. An estimated average throughput of a switch built using this routing table (in packets per second), given that you have 40,000 entries in your routing table. [2 points]

iv. An estimate of the worst-case time to perform a lookup. In the first implementation described above (the one that required too much storage), the time to perform a lookup is constant. In the second implementation described above, the worst-case lookup will involve searching through all n entries before finding the correct IP address (instead of $n/2$ entries, which is average case). I want you to estimate what is the worst case for your particular implementation, and measure it. You will have to modify `main.c` to measure this (it might involve loading a particular set of IP addresses into the routing table). In your report file, describe

   A. an estimate of what the worst case would be in your implementation (in terms of the number of entries in the table, n) [2 points], and

   B. your measured estimate of this worst case [2 points].

   Note that, although the worst-case time is important, it is the average-case time you should be optimizing for in this network switch case.

Grading rubric: 10 points for correctness of implementation. Contingent on the correctness of the implementation, the remainder of the points as indicated in the list above.

(c) **Building the switch.** You will use your routing table as a component to build the complete switch. Copy the files `cam.c` and `cam.h` to the directory `task3c`.

You will leave all the code (files) for this task in the directory `task3c`. The code you write here will likely be simple and fast, thus, you do not need to measure it for speed (the speed of your switch is determined primarily by the performance of your routing table). The files you copied

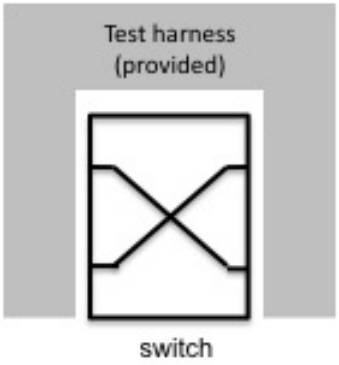

Figure 4: The test harness

contain stubs you should use to write your code. Most of this code is actually part of a test harness (Figure 4) that we have built to allow you to test your code.

The test harness runs as a single thread. Your code can run as either a single thread or a collection of threads (I would suggest a single thread). The threads communicate through the global variables `in_port[0..3]` and `out_port[0..3]`, each of which is an array of type `port_t` (`port_t` is defined in `misc.h`). In addition, there is a single global variable called die. When the test harness is done, it will set this global variable to `TRUE`; any threads that you create should die when this variable goes to true. The global variables are all defined in `main2.c`.

The port variables deserve special attention. Each port variable is a structure containing three fields. One is a mutex (since you have [at least] two threads accessing the port, you want to provide for mutual exclusion). The second parameter is a packet (represented in a `packet_t` data structure that is dened in `misc.h`). The third parameter is a flag that is true whenever the port contains a valid packet. When the test harness sends a message through an input port, it locks the mutex, sets the flag to `TRUE`, and stores the packet in the port data structure. It then frees the mutex. Some time later, your code will lock the mutex, check the flag variable, and if it is `TRUE` (meaning a packet is waiting at this port), it will read the packet, set the flag to `FALSE`, and free the mutex. It can then process the packet. Note that (and this is important) if your code does not read a packet before the test harness wishes to send the next packet on this port, the packet will be overwritten by the test harness (this is considered a "dropped" packet). You would like to minimize the number of dropped packets, so you should make sure you read a packet as quickly as possible after it appears on the input port. On the output side, the test harness will monitor the flag variable in each output port, and when one flag goes `TRUE`, it will read the packet and set the flag to `FALSE` (locking and unlocking the mutex as appropriate).

There is an interesting question of what to do when you wish to send a packet to an output port, but the test harness has not yet read the previous packet from that output port. The simplest implementation would just drop or overwrite the packet. A better implementation might try to somehow buffer the packets until they can be sent. You should try for this better implementation. How many of these buffers you need, what they look like, and how they are used is left to your creativity. Note that, no matter how big you make your packet buffers, there is still a chance they will eventually fill up and then you will have to drop some packets.

The heart of the switch is the routing table. The code you add for controlling the switch will go in `switch.h` and `switch.c`. In these files, you must provide the following functions:

`void switch_init( )` Initializes the switch data structures. May be an empty routine or may contain code, depending on how you implement your switch.

`void switch_add_entry(ip_address_t *address, int port)` This routine adds an entry to the

routing table. The first parameter is a pointer to an `ip_address_t` data structure (defined in `misc.h`) which contains the IP address of the new entry. The second parameter is an integer which contains the output port corresponding to this IP address. This routine is probably implemented by simply calling the `cam_add_entry` routine.

`void *switch_thread_routine(void *arg)` This is the switch thread. This thread will be started in `main2.c`. You should ignore the arg input parameter. When this thread starts, it should immediately start checking the input ports, and routing any packets it nds. It will then pause for a while (using `sleep` or `nanosleep`) and then check again. If you wish to use multiple threads within your switch, you can start the new threads within this routine. It appears simpler to use a single thread. The thread runs until the global variable `die` goes `TRUE`. Then the thread exits.

`void switch_free( )` This routine frees any memory that might have been allocated for the data structures. Depending on how you implement your data structures, this routine might contain no code.

The test harness will first initialize the switch, and then send messages through the switch, monitoring how many of the messages sent into the switch come out the other end. At the end, the test harness will tell you how many packets were not received (an ideal switch would drop zero packets, but this may not always be possible, as described above). You should aim to build an implementation that drops less than 2% of the packets sent.

Note that the test harness actually encodes the output port as part of the packet payload. You should not use the payload to determine what output port a packet should be routed to. The test harness only does this to allow it to easily make sure the packets that arrive from your switch arrive in the right place. Your code should treat the payload as a black box that you can not look inside of. Instead, you will use the destination IP address of the packet to obtain the destination port number using the routing table.

Testing and write-up. In the `task3c` directory, include a file with the name `results.txt` or `results.pdf` (pdf version) containing answers to the following questions:

i. A short description of how you implemented your switch [7 points].
ii. The packet drop rate as indicated by our test harness [3 points].
iii. Anything else you want the markers to know.

Grading rubric: 15 points for correctness of implementation. Contingent on the correctness of the implementation, the remainder of the points as indicated in the list above.

4. **Handing in your assignment**

Ensure that your code runs on `ssh-linux` and then use `handin` (from `ssh-solaris.ece`) to submit the assignment as usual.

```
handin eece494 hw2
```

The submission must include the names (and student ids) of the students who worked together on the assignment.

This group assignment contributes to 10% of your total grade. *Your grade for this assignment will be affected by peer review. The process for peer review will be announced soon.*

3 What to take away

Schedulability analysis. This assignment will make you familiar with the implementation of schedulability tests and the relative time complexity of the different tests. The relative complexity of implementation affects

the throughput (number of schedulability decisions per second) that can be made, and this may be a crucial factor in some systems where a fast and approximate decision is more important than the best decision. For example, a network switch that admits new flows only if it can meet the deadlines associated with the flows may need to make many thousand schedulability decisions each second.

You will also learn to generate synthetic workload and you will understand the gap between worst-case analysis and typical (average-case) behaviour of scheduling policies. This shows the connection between sound statistical methods and rigorous testing.

Network switches. Building a real-time system requires careful engineering of many components. While, in lectures, we have focused on higher-level policies related to scheduling and resource allocation, it is just as important to optimize the performance of components (e.g., execution times). This example with a network switch illustrates that principle. Recall that the previous assignment demonstrated an optimization that avoided the use of floating-point computations. Unfortunately we do not have the time to explore many of these performance enhancements but these are factors that you should always consider.

Further, notice that in the switch that you built for this assignment, we assume that packets can be transferred from an input port to an output port almost instantaneously. That requires extremely fast memory transfers. As the number of ports and the expected bandwidth increase, conventional memory speeds cannot cope with the requirements of the switch. This has led to the design of alternative switching mechanisms; the next assignment will touch upon one such design.