**Exercise 2**

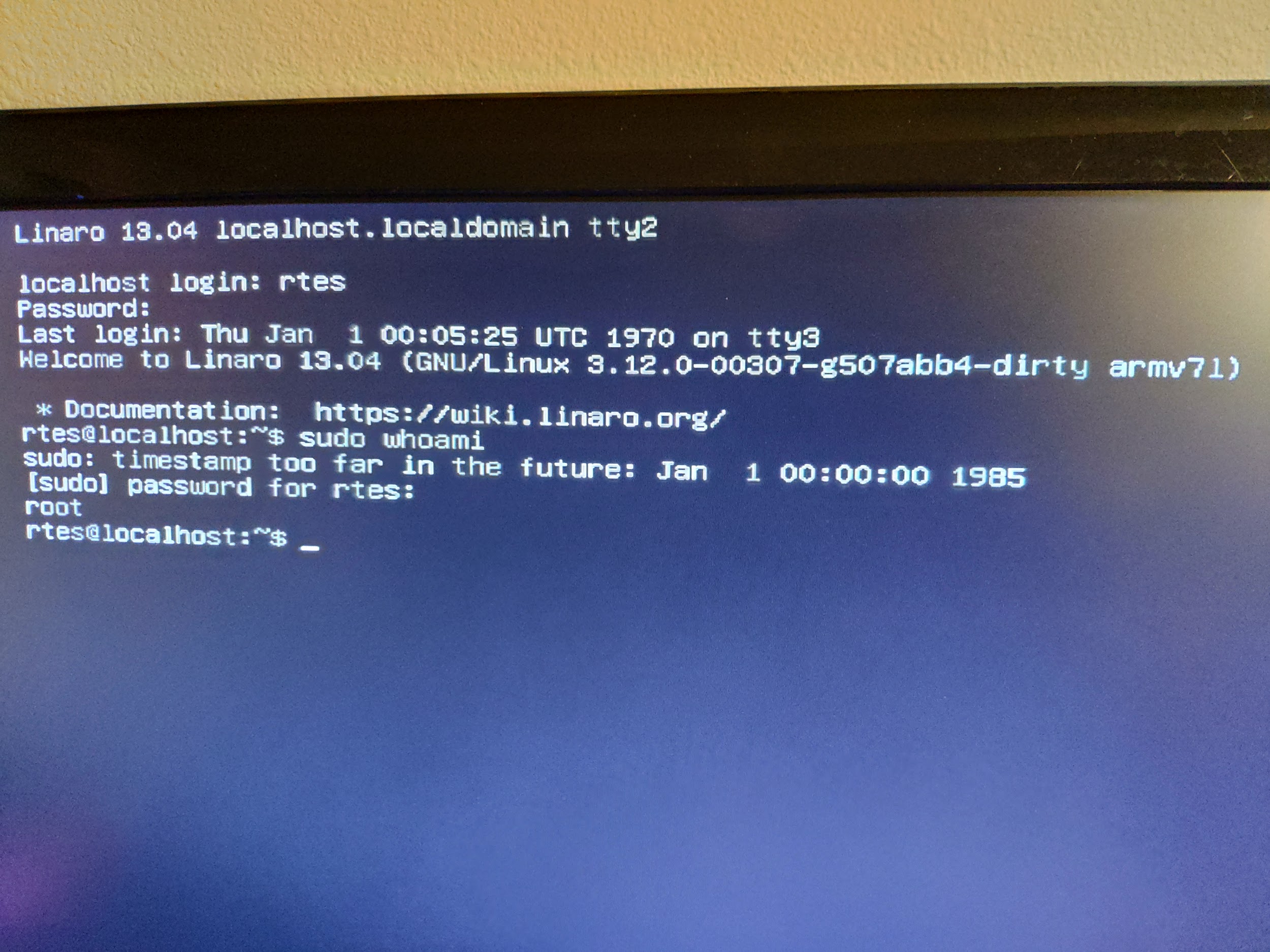
**ECEN 5623 Real Time Embedded Systems**

**Dominic Doty and Ayush Dhoot**

**Question 1:**

*Create a new account on the board and grant it superuser permissions.*

We were able to make an account on the Altera DE1 SoC board using the LXDE image (Linaro 13.04). We have attached a screenshot of the login to verify the same.



**Question 2:**

*Explanation and Critique of STS PASS System:*

Shuttle system was complex, had very limited memory, required lots of redundancy. This drove a somewhat complex memory management system, as well as redundant networked computers.

The system needed to support all phases of flight but working memory was too limited to store all the required code. The result was to segment the application into operational sequences (OPS) which could be loaded from a larger, slower memory as required for different phases of flight. The OPS were further segmented into resident, major functions, and overlay chunks. Resident chunks can remain in memory for all OPS, major functions remain in memory for several OPS (and are shared across them), while overlays are functionality that is unique to that OPS segment. This reduced the overhead required to transfer the data from mass memory into working memory.

This implementation also lead to very modular design, which the author notes was helpful in development and testing of the software, allowing parts to be tested in the system as they were completed. Within each OPS segment, there were generally three main applications. These were Guidance, Navigation & Control (GNC), Systems Management (SM), and Vehicle Checkout (VCO). The applications were run in a multi frequency executive, which would dispatch various tasks at different frequencies from 25 Hz for high priority functions, down to 6.25 to .25 Hz for medium and low priority functions. The overall system was much more complex than a simple cyclic executive however, due to the redundancy and secondary computers used to offload the main general purpose computers. The article notes that offline secondary processors were used (in an asymmetric multiprocessing like setup) to parse user input and output display information from the main computers. There were also special provisions made to synchronize operation of the five redundant general purpose computers, allowing them to vote on outputs and remove computers that returned erroneous results.

Overall the PASS system is very impressive in that it achieved high reliability and extensive redundancy in very compact data size both in code size and working memory usage (overall code base was ~1900 KB, working memory of the computer was ~414 KB according to the author).

*Advantages:*

1. NASA says they liked the fixed priority interrupt driven scheduling over time slicing because it “degrades gracefully when overloaded” (<https://history.nasa.gov/computers/Ch4-5.html>, near source 108).

2. Highly reliable and deterministic because the schedule was determined at compile time.

3. Potentially simpler than full preemptive scheduling.

4. Limited unexpected concurrency since all functions within one frequency cycle are guaranteed to never interrupt each other, providing clear boundaries for testing. (Simplified thread safety).

*Disadvantages:*

1. Difficult to modify, adding more functions requires reverifying the whole schedule.

2. Redundancy requirements and system complexity drove a very complicated system in spite of the reduced complexity from using a cyclic executive.

3. Basically 80% of the work of implementing a preemptive, networked, multiprocessor system, but doesn’t achieve the benefits of a true networked multiprocessor system (sharing more resources, easier scheduling).

**Question 3:**

*Describe the concept of the Cyclic Executive and how this compares to the Linux POSIX RT threading and RTOS approaches we have discussed*

The cyclic executive is a simpler implementation of a real time system. The main components of the cyclic executive are the major cycle, minor cycles, and frames. The major cycle is made up of several minor cycles, and each minor cycle is made up of several frames. The cyclic executive uses two timers to control real-time operation. One timer for the minor cycle, and one for frames. At the beginning of each minor cycle and each frame, their respective timers are started. If the timer expires while its respective block is still executing (minor cycle or frame) an overrun has occurred and an exception is raised. In most systems, the exception handler will cancel the currently running iteration of the minor cycle or frame, and reset the component to a safe state for its next execution. The handler may also perform some degraded functionality to cover for the component that missed its deadline.

This differs significantly from the Linux RT and RTOS approaches that we have discussed, in that it is not preemptive. The timers ensure that a tasks will always be stopped and restarted if they overrun their deadlines, but one task will never preempt another in execution. This has several benefits including no context switches (low overhead), no unexpected concurrency, and a fully predictable schedule. On the other hand, there are downsides to using a cyclic executive over a preemptive scheduler. The downsides are mainly driven by the low level of abstraction in a cyclic executive. The programmer effectively writes the schedule at compile time and thus must make sure that every task will complete in its allotted time. This makes changing code more difficult as the whole schedule must be reevaluated for every change to a single task.

For a simple real time system with a small number of periodic tasks, a cyclic executive can work well, but as system complexity increases and the number of tasks increases a cyclic executive quickly becomes difficult to maintain and more work than a full RTOS.

**Question 4:**

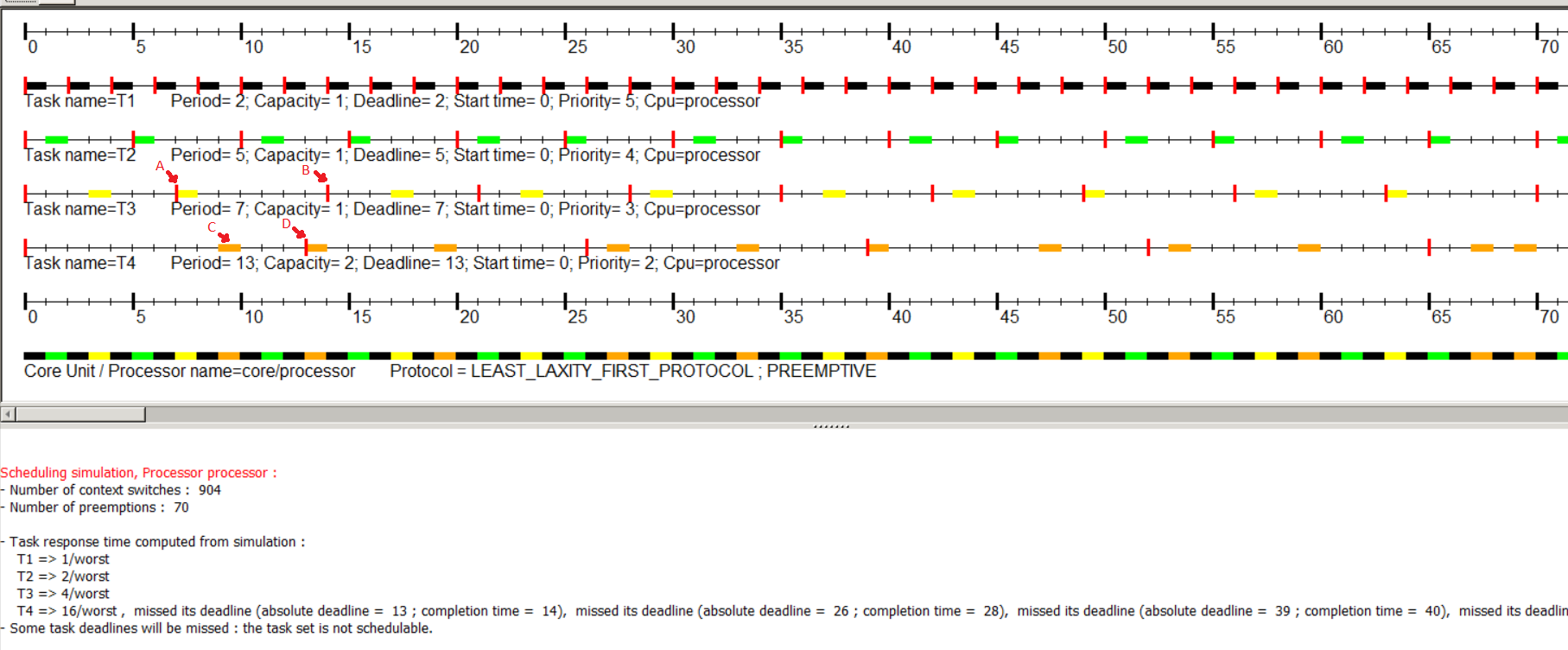
*Simulate the provided example schedules in Cheddar, and in custom simulation code.*

All noted examples were run in Cheddar 3.1 on Windows 10, as well as custom scheduler simulation code written in C (feasibility\_tests.c). The custom scheduling tests were compiled and run on a machine running Windows 10 with Mingw64 GCC 8.2.0 targeting x86\_64, as well as a virtual machine running Ubuntu 18.04 with GCC version 7.3.0 targeting Linux x86\_64. The results of these tests are noted in the below table. Note that examples 2, 6, and 8 are the same set of services, and thus have the same result.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Ex.** | **Custom Test Code** | | | | | **Cheddar Analysis** | | |
| **Scheduling Point** | **Completion Time** | **RM Over LCM** | **LLF over LCM** | **EDF over LCM** | **Rate Monotonic** | **Least Laxity** | **Earliest Deadline** |
| 0 | Feasible | Feasible | Feasible | Feasible | Feasible | Feasible | Feasible | Feasible |
| 1 | Infeasible | Infeasible | Infeasible | Feasible | Feasible | Infeasible | Feasible | Feasible |
| 2 | Infeasible | Infeasible | Infeasible | Feasible | Feasible | Infeasible | Infeasible | Feasible |
| 3 | Feasible | Feasible | Feasible | Feasible | Feasible | Feasible | Feasible | Feasible |
| 4 | Feasible | Feasible | Feasible | Feasible | Feasible | Feasible | Feasible | Feasible |
| 5 | Feasible | Feasible | Feasible | Feasible | Feasible | Feasible | Feasible | Feasible |
| 6 | Infeasible | Infeasible | Infeasible | Feasible | Feasible | Infeasible | Infeasible | Feasible |
| 7 | Feasible | Feasible | Feasible | Feasible | Feasible | Feasible | Feasible | Feasible |
| 8 | Infeasible | Infeasible | Infeasible | Feasible | Feasible | Infeasible | Infeasible | Feasible |
| 9 | Feasible | Feasible | Feasible | Feasible | Feasible | Feasible | Feasible | Feasible |

In example 1 and 2, we see that RM fails but LLF and EDF passes. As we know, RM is a fixed priority policy which will define the priority at the start of the test according to the highest frequency. Meanwhile, EDF and LLF are dynamic priority policy based where the priority is recalculated each time a new task is to be serviced. This has a significant effect in these cases because the task periods are very inharmonic (2, 5, 7, 13). Rate Monotonic scheduling can only exceed the Least Upper Bound in special cases with harmonic services. Because this set is inharmonic, it is not schedulable with RM. LLF and EDF do not have harmonic requirements however, and so are able to be scheduled.

One interesting case is that of example 2 (which is the same as 6 and 8), where Cheddar determined that the schedule was infeasible with the Least Laxity Scheduler but our analysis shows that it is feasible. In this case, it appears that Cheddar is incorrectly implementing the Least Laxity calculation, shown in the below screenshot.



The above image shows that the scheduler runs task 3 (yellow) at point A, when it should run task 4 (orange). At point A in the above image, the LLF of Task 3 is 6, while for Task 4 it is 4. This can be seen easily in the above image since task 4 has a closer deadline (point D) than task 3 (point B) and it also has a longer execution time remaining (2 executions to task 3’s 1 execution).

**Question 5:**

*Derivation of RM LUB*

*Assumptions:*

1] The requests of all services are periodic in nature and that period is always constant.

2] The completion time for a task must always be within the time period.

3] The runtime is known and deterministic for all cases.

*Constraints:*

1] The deadline of any service should always be the same as that of its period.

2] There is a Fixed Priority, Preemptive, Run-to-Completion Scheduling used.

3] Service requests are independent.

*Key Derivation Steps:*

1]

In Theorem 3, Case 1, it is written as the processor utilization factor U is monotonically decreasing in C1 which is difficult to imagine without any graphical or arithmetic validation. The same is for Case 2 where U is monotonically increasing in C1. The text helps in the understanding of the statements as it graphically represents the processor utilization factor along with the arithmetic it used to arrive at the representation.

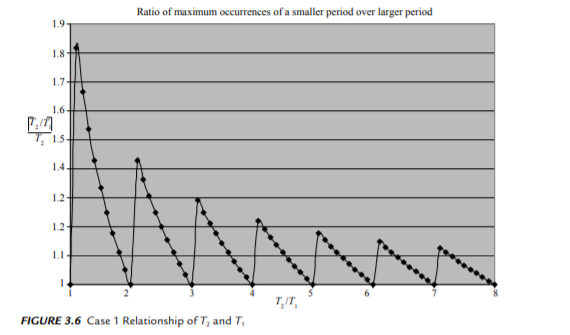


Figure : Case 1 where U is monotonically decreasing (note y axis shows ceiling function when it should show floor function)

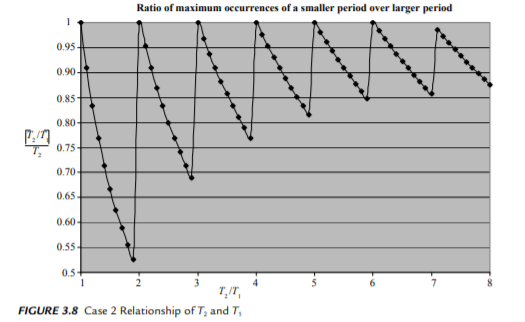


Figure : Case 2 where U is monotonically increasing

2]

Theorem 4 in Liu and Layland is searching for a set of “tasks that fully utilize the processor and minimize the processor utilization factor” I don’t understand how it is possible for tasks to fully utilize the processor and minimize the utilization factor.

3]

Theorem 4 in Liu and Layland also defines a sequence of task sets that “clearly … fully utilize the processor” (53, Liu/Layland). The generalization from C1’ to Cm’ is confusing. I don’t understand why they wish to prove C1 = T2 - T1, nor how the sequence shown proves that is true.