Electrical, Computer & Energy Engineering
UNIVERSITY OF COLORADO **BOULDER**

ECEN 5623 Exercise #5 Real-Time Continuous Media

Course Name: **Real-Time Embedded Systems**
Student Name: **Rushi James Macwan**
Board Used: **Raspberry Pi 3B+**

Note: Correct answers are in Blue Font

Please Note: The images included in this report may not be very clear and as desired. Please, refer to the attached zipped file which contains all the images provided in this report.

1) [15 points] Research, identify and briefly describe the 3 worst software and hardware computer engineering design flaws and/or software defects of all time [product lack of success can be considered a failure such as Blackberry Storm, Windows Genuine Advantage, Windows 8, Windows ME, Apple Lisa, Pentium FPU bug, as well as mission failures such NORAD false alarms, Mars Express Beagle 2, Challenger and Columbia Shuttle Loss]. State why the 3 you have selected are the worst in terms of negative impact, negligence, and bad decisions made, but why the failure was not really a real-time or interactive systems design flaw. Rank them from worst to least worst.

Sol.

3 Worst Software Failures:

#1 Failure:

The 1979 NORAD false alarms and the 1983 Soviet Nuclear false alarms:

On both the occasions of the 1979 NORAD false alarms and the 1983 Soviet Nuclear false alarms, there were one of the worst software failures because of which I have ranked both of them in the first place. In the case of the 1979 NORAD case, there was a technical glitch in the system that allowed the technician to send the warnings across the country when the system was fed a test tape. There were not enough system features that could account for the differences between the test tapes and the real-scenarios. Because the technician was testing the NORAD alarm system for a nuclear strike, the system simply sent out the warnings to the entire North American continent that nuclear strikes were being performed. However, the system should have other features to prevent this from happening. This was a very simple yet major software defect in the NORAD alarm systems. On other occasions, at least two more – as one of the references claims, the NORAD alarm system failed because of the failure of a computer communications device and because of faulty computer chips as the other references speak of.

On a different occasion, the 1983 Soviet nuclear false alarm was of a similar nature but with some other technical glitches in the system. The system was allegedly not tested sufficient enough and took the software in the system assumed the reflections of clouds in the atmosphere as an incoming missile. This led to a serious emergency situation as other references speak of. The software had this bug which could have been eliminated with thorough research and testing but it was not known until after the incident actually happened.

Both the above software failures on several occasions have been the worst in my opinion. This is because they had the potential to lead the world into a serious warfare that has been unprecedented ever before. Because of some careful decisions, immensely catastrophic destruction was saved and mankind was spared.

References: [1] [2] [3] [4] [5] [6]

#2 Failure:

The 1991 Patriot Mission Failure:

One of the major software failures is the 1991 Patriot Mission Failure where the missile defense system failed to detect an attack on an army barracks and led to the death of 28 soldiers while 100 more were injured. This was the result of a simple bug in the software of the missile defense system. The system software had an inaccurate tracking calculation problem that became worse the longer the system was operated. The original problem was related to an inaccurate calculation of time since the system was boot due to computer arithmetic errors. Because of the inaccuracies with its 24-bit fixed point register and the arithmetic operations associated with it, the incoming scud was outside the tracking range of the missile defense system. This led to serious casualties as described above. Ironically, as the reference document claims, the program was fixed in some parts for the timing inaccuracies but not for all points and so the inaccuracies were unevenly cancelled out which led to the tracking problem.

References: [1]

#3 Failure:

Mariner 1 Failure:

Mariner 1 had a serious failure on July 22, 1962. This was a result of equipment failure and due to faulty instructions programmed into its onboard computer. As the referenced document claims, faulty application and bugs in the software design of the programmed computer on board Mariner 1 led to false application of guidance commands and were directing the spacecraft towards a crash in the North Atlantic area, possibly in inhabited areas. It was later found that the computer program that was used to fly the Mariner 1 had some serious errors. Upon failure to receive airborne beacon signals, the error code fed to the computer caused it to accept the sweep frequency of the ground receiver which was mixed up with the incoming tracking data sent to its guidance computer. Eventually, it ended up receiving faulty navigation guidance and veered of its course. Also, it was later found that a small mistake in the software of the code (i.e. a missing hyphen in the application software) was responsible for the spacecraft in acquiring faulty transmission signals and therefore, it explains the faulty maneuvering of the spacecraft.

References: [1] [2]

Reasons for ranking:

Negative Impact:

- The NORAD and Soviet nuclear false alarms both had a chance of very high negative impact. However, due to careful consideration of the fact that the systems could be potentially filled with errors, the most catastrophic warfare was prevented in the human history.
- The 1991 Patriot Mission Failure had its own casualties in terms of loss of human lives on the side of the US military. The negative impact was very catastrophic as human lives were lost because of the failures to achieve reliable and deterministic results.
- The Mariner 1 failure although did not lead to loss of human lives, the Mariners 1-10 cost approximately $554 million. Therefore, it definitely had major negative impact on the country and its ambitions to explore space. Great research could have been performed had this failure had not happened.

Negligence & Bad decisions made:

- The negligence of the NORAD and Soviet nuclear false alarms were as a result of technical design issues in the software of the defense systems that were used. The chances of loss were astronomically high and a lot of loss could was saved in careful decisions that were made on both the occasions.
- For the 1991 Patriot Mission Failure, the result of a buggy software accounted for the lives of many US soldiers. It could have been prevented with careful design considerations.
- Mariner 1 failure was again as a result of bugs in its software design like the presence of a hyphen in its program that was loaded in its computer system. It could have been prevented for sure with the help of reliable and careful testing.

Why the failure was not really a RT / interactive systems design flaw:

- Both the NORAD and Soviet nuclear false alarms were as a result of non-RT design flaws. The primary flaws were the buggy codes and had very limited dealing with human interaction. Although, the technician fed test tape for NORAD testing, it was less of an interactive system design flaw than a system software failure that did allow the technician to send global warnings.
- The patriot mission failure was again not a RT failure as the system had a deterministic failure every time that was based on a bug in the software code. The bug did not have to deal with any RT external constraints except for time and the system was not programmed flawlessly and was not optimized enough for the real tested environment. It was not an error of interactive systems design flaw either but completely a buggy software.
- Mariner 1 again was not a RT failure as the system was tested incorrectly for the real-space application as the hyphen in its program made it accept wrong values for its position and navigation and eventually ended up in a complete failure. It was not an error of interactive systems design flaw either but completely a buggy software.

Courtesy: Reference Links: [1] [2] [3] [4]

2) [15 points] Research, identify and briefly describe the 3 worst real-time mission critical designs errors (and/or implementation errors) of all time [some candidates are Three Mile Island, Mars Observer, Ariane 5-501, Cluster spacecraft, Mars Climate Orbiter, ATT 4ESS Upgrade, Therac-25, Toyota ABS Software]. Note that Apollo 11 and Mars Pathfinder had anomalies while on mission, but quick thinking and good design helped save those missions]. State why the systems failed in terms of real-time requirements (deterministic or predictable response requirements) and if a real-time design error can be considered the root cause.

Sol.

3 Worst Real-Time System Failures (design and implementation flaws):

#1 Failure:

Mars Observer:

The Mars Observer incidence revolves around the problem that the communication with the spacecraft was lost. It was assumed that this happened as a result of the rupturing of the fuel pressurization tank in the spacecraft's propulsion system. Later, this would leaked the fuel during the spacecraft's cruise to Mars. The leaking gas probably resulted in a high spin rate which the system was not capable of. The high spin rate allegedly led the system into its contingency mode which interrupted the stored command sequence on its flight computer and eventually the transmitter was not turned on to communicate with the ground station. The propulsion failure was as a result of inadvertent mixing of chemicals inside the propulsion system which led to the rupturing of the fuel pressurization tank and hence the high-spin rate as a result of fuel leakage. However, if the software had been considered to handle this exception in RT, the spacecraft would have still managed to communicate to the ground station and the destruction could have been prevented. In my opinion, I assume that this happened due to an overloading of information on the spacecraft computer system when the spacecraft experienced high spin-rate. Due to very high spin-rate, the spacecraft serviced those threads of execution that were related with the acquisition of the space position and navigation. This resulted in a higher frequency of execution of these services and because of potential Rate Monotonic scheduling policy that would have been implemented for the spacecraft, the thread pertaining to the position and navigation system on the spacecraft computer preempted other important services and acquired higher priority than all other services due to its high frequency of execution. This was potentially the reason why the spacecraft was unable to run the stored command sequence on its computer as it was being interrupted and thus the transmitter was not turned on.

References: [1]

#2 Failure:

Northeast blackout of 2003:

The Northeast blackout of 2003 was one of the worst real-time mission critical failures. The blackout was a result of a computer bug that was all about race conditions that existed in the General Electric's energy management computer system. Due to the race condition in a multi-threaded environment that dealt with 256 power plants, a race condition occurred due to a software bug (which was a RT issue). The bug stalled the control room alarm system and because of that, system operators were unaware of the malfunction. The failure prevented the operators from receiving any acknowledgement of the underlying malfunction. This was essentially because of the race condition that the system was facing. Unprocessed events queued up as the referenced document claims, and the server failed. Later, the same unprocessed events were queued up on the backup server and it failed as well. This led to a widespread system failure and the server failures even slowed the screen refresh rate of the operator computers. In reality, this was a truly a RT mission critical failure, where a race condition (a bug) led to a massive power outage across the Northeast of the US.

References: [1]

#3 Failure:

ATT 4ESS Upgrade:

The ATT 4ESS upgrade was a truly a RT system failure that led to a crash of AT&T network in 1990. The primary reason behind the ATT 4ESS upgrade was a bug in the new software that was launched to service the network connections. The new software allowed faster network setup and servicing but had a RT-system fault that was not detected until after the network crashed in 1990. In the previous switching software, whenever a network device failed, the associated switch (e.g. A) would notify all other 4ESS switches that it was resetting and it wasn't communicating during the resetting time. Once it was reset by its system, it would inform again to all other switches that it was ready for successful communication. However, with the new software, the switch upon getting reset, would not inform other switches that it was restored to operation but would directly begin processing calls and sends out routing signals. Therefore, the reappearance would inform other switches that the switch A was restored. There was a problem with the new software that when switch A was restored, it would start requesting other switches (e.g. switch B) for information and then switch B would reset itself before servicing switch A to change its internal logic to show that switch A was restored from its faulty stage. However, the problem arose when switch B got a second call from switch A while it was resetting its logic to showcase that switch A was back. This led switch B to crash and this very process occurred for the communication of switch A with every other switch that it communicated to and so all the switches in the AT&T network that switch A connected crashed. This led to a crash of the ATT 4ESS upgrade network.

References: [1]

Reasons for ranking:

Why the systems failed in terms of RT requirements (deterministic or predictable response requirements):

The Mars Observer was a RT system failure as it was not tested for its deterministic response to conditions that actually put it into its contingency mode as it were. The mission was not reliable for a high spin rate as mentioned in the above explanation resulted in its critical software being interrupted and preempted by the service that was to responding to the extreme spin rate as it were.

The Northeast blackout of 2003 was again a completely RT failure as it was explained because of its race conditions in its computer software. The access to the system software services was not protected through synchronization primitives and it led the system to fail in responding to the requests in RT. The operator computers failed to refresh the information at a desired rate and so did the power plants were lost and the servers failed because of the continuous race conditions that were not handled and accounted for in RT.

As with the Northeast blackout of 2003, the AT&T crash of 1990 is a similar event where necessary RT system debugging was not performed and as a result the network switches went into a failure mode as there were both bugs in the software and necessary RT-resetting interface was not designed. It was both a failure as a result of RT-response errors and also as a result of buggy code.

Whether a RT design error be considered a root cause:

For all the above three cases, a RT design error can be considered to be a reason of the root cause and the reason behind this is the fact that these systems were loaded with buggy codes that were not efficient to provide RT system response in environments that demanded highly efficient and reliable RT responses. In the case of the Mars observer, it happened that the system was not designed in its RT aspect to address all those issues of overloading of navigation and position data as it happened when it experienced the high spin-rate. For the Northeast blackout, the buggy codes that created a racing condition did not have a proper mechanism to correct errors while synchronize the access to these systems. For the AT&T 4ESS network, again there was a network failure which was as result of RT-design errors. If the design had accounted for the associated errors and if the system was reset and handled properly, the domino effect where all the switches, one after other that they failed and shut down themselves, could have been prevented. Therefore, in my opinion, all of the above three cases are as a result of RT design errors that were not taken care of.

Courtesy: Reference Links: [1] [2] [3] [4]

3) [20 points] The USS Yorktown is reported to have had a real-time interactive system failure after an upgrade to use a distributed Windows NT mission operations support system. Papers on the incident that left the USS Yorktown disabled at sea have been uploaded to Canvas for your review, but please also do your own research on the topic.

a) [5 pts] Provide a summary of key findings by Gregory Slabodkin as reported in GCN.

b) [5 pts] Can you find any papers written by other authors that disagree with the key findings of Gregory Slabodkin as reported in GCN? If so, what are the alternate key findings?

c) [5 pts] Based on your understanding, describe what you believe to be the root cause of the fatal and near fatal accidents involving this machine and whether the root cause at all involves the operator interface.

d) [5 pts] Do you believe that upgrade of the Aegis systems to RedHawk Linux or another variety of real-time Linux would help reduce operational anomalies and defects compared to adaptation of Windows or use of a traditional RTOS or Cyclic Executive? Please give at least 2 reasons why or why you would or would not recommend Linux.

Sol.

a. <u>Key findings:</u>

The paper discusses the key reasons behind the system failures with the case of Yorktown. Gregory Slabodkin discusses the topic in detail involving the opinions of several resource persons who were directly associated with Yorktown and its failure.

As Gregory Slabodkin mentions in his paper, through several sources and resource people, the following key findings were presented in my opinion:

- Yorktown suffered a systems failure and its propulsion system crashed as a result of it. The primary reason according to the paper behind this issue was that bad data was fed into the computers on the system.
- Yorktown was equipped with the Windows NT Operating System to service the LANs (Local Area Networks) and Computers on board the ship. The Windows NT system was responsible for running all the software applications on the ship including the Remote Database Manager (RDM) program.
- At a certain point, an administrator on the ship was required to enter a zero into one of the data fields of the RDM program. However, the officials were not aware that it was a bad data for the computer system and the associated applications on Yorktown.
- The RDM program attempted to perform a division operation using the entered value in the data field in RDM which was a zero. This led the program to crash as there were buffer overflows as the computer were not able to perform the division.
- However, according to the paper, what is intriguing is the fact that this failure of the RDM program led to the crashing of the entire Windows NT OS that serviced all the software applications on the ship which included responsibilities to provide damage control, run the ship's control centre on the bridge, monitor the engines and navigate the ship when under way.
- Therefore, when the bad data crashed the RDM program, the Windows NT OS loaded in both the networks and computers on board the Yorktown, failed to address this problem and crashed. Because the Windows NT OS crashed, the network switches associated with the LANs on the ship and the computers crashed leading to a domino effect where there was a chain reaction of computer failures, as the paper describes.
- According to a source in the paper, Unix OS are better for control of equipment and machinery whereas the Windows NT 4.0 (which was loaded into both the networks (LANs) and computers on board the Yorktown) is more suitable for the transformation of information and data.
- Thus, **<u>to summarize the key findings of the paper</u>**: Yorktown faced a critical systems failure when the Standard Monitoring Control Systems Administrator (SMCS) entered a bad data (zero) into a data field in the Remote Database Manager (RDM) program. This led the RDM program to request the computer to perform division operation by the bad data (zero) which in turn led to buffer overflows and the program crashed. Windows NT was charged with the responsibility to service the RDM applications and all other software applications on Yorktown which included managing the network switches (for LANs) and applications like controlling machinery (i.e. propulsion system). Since, due to resources in the paper, Windows NT was not apt for handling machinery applications, the crashing of the RDM program led to a domino effect where the OS crashed due to a system failure and all critical software applications serviced by Windows NT crashed. As an end result, the propulsion system on board the Yorktown which was serviced by Windows NT as well, crashed as the OS crashed because of the RDM program failure. At the bottom line, the applications (i.e. RDM) were not developed to address such issues of bad data and had programming inadequacies that resulted in its crash

and because Windows NT was not suitable for such machinery applications crashed as well when the RDM program crashed and as a result – all the networks and applications were blacked out (i.e. crashed) leading to a systems failure involving the failure of the propulsion system.

- According to sources in the second referenced document, ***Windows NT was never the cause of the problem*** on the ship but essentially there were problems with the programs, database and code created within the individual pieces of software that were used and developed by the software programmers of Yorktown. It was added that the software programmers had quickly come up with the new propulsion system that was serviced by the Windows NT OS but there were inherent risks with the new applications as they were built without much testing and with bad programs that could crash the program and thus the entire OS leading to complete systems failure.

References – [1] and [2]

b. Alternate key findings:

The key findings by Gregory Slabodkin have mixed reviews on the premises that Windows NT was responsible behind the systems failures. While some resources in his papers, point to the fact that Windows NT was entitled to serve both the networks and the computers aboard the Yorktown, it was not actually the primary culprit behind the systems failure. It was mostly established that the critical failures were as a result of the error prone software applications that were brought up quickly for the automated ship system which included the new propulsion system. These new application software were not thoroughly tested as Gregory claims in his work through several resources, and that these programs had errors pertaining to bad data which would crash upon encountering a bad data. Therefore, the sole premises of Gregory's work revolves around the fact that "***bad data and faulty programs designed for the Yorktown ship were ultimately responsible for the crashing of the systems applications on board and that the Windows NT 4.0 Operating System was not inherently the reason behind the failure.***"

According to alternate resources [1] and [2], the following facts were established which have an alternate theory about the failure:

- Although, the inefficient programming of the software applications could have resulted into a bad data leading the programs to crash, the Windows NT 4.0 Operating System that was loaded into the networks and computers on board the Yorktown was not efficient enough to prevent system failures across other applications when one or more of these applications failed.
- The above fact was insinuated with the prime argument that the RDM was fed with a bad data entry (zero) and the application attempted to perform division operation with a zero. That resulted into buffer overflows and the RDM application crashed. While Gregory claims that the RDM application failure resulted into a systems failure as it was also operating other software applications, the alternate resources focus on the fact that the "***Windows NT OS was not capable of segregating other application software executions inside the RDM which were not associated with the bad data entry application (i.e. SMCS).***"
- Because the Windows NT OS could not segregate the crashing of the program that was dealing with bad data, one program failure led to another inside the RDM and the entire RDM system crashed because of the inefficiency of the underlying Windows NT OS.
- The alternate sources claim that if the Unix OS system was used, which are suitable for such applications and provide high reliability, the failure of one software application should not have affected other one (i.e. the propulsion system should not have failed) and the human-computer interaction could have been improved much better with that.
- However, the alternate sources claim that because Unix OS were considered not to have as good GUIs as Windows NT, the officials chose Windows NT over Unix OS for a more user-friendly experience and reduced workload on the ship, eventually leading to critical failures.
- The second alternate referenced document also claim that there was a problem with the system applications onboard Yorktown when a valve that was physically closed appeared to be open digitally by the system. The technician tried entering the zero value into the data entry leading

to the failure of the division operation by the RDM and since RDM also serviced all other critical applications like the propulsion system, crashing of RDM led to a disruption of all other associated critical software applications including the propulsion system.

- **To summarize, the alternate resources claim that the use of the Windows NT OS was responsible behind the critical failures as the programs may have been faulty and would have crashed but eventually, the OS was not able to segregate the errors with one application from all other applications – which is observable by the domino effect of the chained failures across the critical software applications on the ship associated with RDM.**

References – [1] and [2]

c. Root cause and culpability:

The root cause of the fatal and near fatal accidents with the USS Yorktown are associated with the bad programs that were quickly brought up without sufficient testing. In light of that, my opinion aligns more with Gregory Slabodkin that although the Windows NT operating system may have been less suitable for applications that involved dealing with applications-specific machinery but the prime culprit behind the failures were the poor programs that were written for the software applications. These programs were simply not capable of resetting the system in RT whenever an error like a division by zero was observed.

Moreover, the application software did not have necessary interface to segregate errors from the software applications that were unrelated but interconnected because of the OS. Sufficient care was not taken when the programs were created which resulted into buggy programs that had improper synchronization of shared global data across the services (software applications) ran by the OS. Issues like a division by zero were not handled properly and these conditions may not have been addressed which resulted the program to crash straightforward. Because, the synchronization of shared data and execution across the services may not have been designed efficiently, the crashing or disruption led to a complete disruption of all other software applications tied to the same OS. Primarily, this could have been prevented with necessary modifications to the buggy code that did not address situations where the thread could be attempting to perform division by a zero. Secondly, even if such situations had occurred, the system should have had provisions to segregate the threads such that failure of one thread should not have affected other thread. Last but not the least, the system did not have a fast resetting mechanism that could quickly reboot the system in RT without losing any data.

Therefore, in my opinion, the root cause behind the critical and fatal accidents of Yorktown was **primarily** the buggy code that was developed for the software applications which did not address the issues in RT and did not have sufficient synchronization primitives that prevented disruption of other services in case if one failed, and importantly, the system did not have a mechanism to reset in RT (i.e. WDT that could quickly reset the system) leading to all the critical fatal and near fatal accidents mentioned in the previous sections. However, **secondly**, Windows NT was not suitable for the application environment on Yorktown and the reasons are mentioned in the below section. The use of a RT-Linux OS could have reduced the operational anomalies and the defects that led to critical failures.

d. RT Linux ok, why or why not?

I believe that RT-Linux would have been suitable for the application than the use of Windows NT. It would have helped to reduce the operational anomalies and defects found in the above sections. This would have been true for the following reasons:

- Primarily, RT-Linux kernel has a CONFIG_PREEMPT_RT patch for HRT systems and a CONFIG_PREEMPT patch for the SRT systems. With the use of the RT patch, the failures pertaining to other software applications could have been prevented as the thread which had a failure could have been preempted and/or terminated. This could have provided safety from critical fatal and near fatal accidents that occurred on Yorktown like the failure of the propulsion system as the RDM crashed because of some other software application related to a bad data (i.e. division operation by a zero). With the use of the Linux-RT services, thus the subsequent failures of software applications and domino effect could have been prevented.

- Secondly, RT-Linux kernel is a monolithic architecture and the Windows kernel has a hybrid architecture. The prime difference here is that a monolithic kernel application is segregated and independent of other kernel computing applications. The same does not hold true for Windows wherein the hybrid kernel architecture deals with the combined benefits of both microkernel and monolithic kernel. The problem is that the computing applications were not segregated from each other in the Windows NT OS that was used for Yorktown software applications. Therefore, although the software codes that were developed were buggy, the RT-Linux which isolates the computer software applications from each other (because of its monolithic kernel architecture) could have made it possible to prevent complete systems failure as failure of one computing application (i.e. one thread) would not have affected and disrupted another computing application (i.e. the thread dealing with the propulsion system). Therefore, in my honest opinion, the use of RT-Linux could have prevented saved Yorktown from falling a victim to the domino effect which involved chained failures with the computer applications serviced by the loaded Windows NT OS. Although, monolithic kernels have their own disadvantage against hybrid kernels (that Windows NT OS implemented) like one kernel component failure can lead to the entire system failure, proper and robust programming with lesser complexities can prove to be a better solution that using the Windows kernel.

- Thirdly, the RT-Linux kernel, being a monolithic kernel, is fast and can provide better response than the Windows kernel which has a hybrid architecture. This aligns with the first point where RT responses are much needed in applications that go into a system like the one on Yorktown. With the use of RT-Linux kernel threads, preemptive scheduling and fast handling of multi-threaded environment can prove to be resourceful in software applications that are applications specific and are focused on handling machinery than focusing on the transformation of information and data. This is again aligned with the views presented by the alternate resources.

- The RT-Linux kernel supports 100% multi-user environment which is the prime necessity in applications like the Yorktown. The same does not go true with the Windows kernel which can lead to total systems failure as was discussed earlier.

- RT-Linux GUI stack is in the user-space and not in the kernel space and therefore any issues with the human-computer interaction and user interoperability could not directly affect the response of the system to critical software applications like the failure of the propulsion system. Therefore, although buggy codes had led to a situation where the system attempted to perform division operation by zero, the RT-Linux kernel could have prevented the same with necessary software protection in the GUI stack that could have segregated the errors from the kernel and would have eliminated the errors in the user-space itself.

- Last but not the least, Windows NT does have a slower kernel because of its hybrid architecture and does not provide a SRT support as it has a multi-media class scheduler service and has no patch for HRT response. This leads to the primary discussion that it is a best-effort OS and does not provide RT system support for such critical applications as in the case of Yorktown. Plus, the absence of a proper scheduler (i.e. the multi-media class scheduler) for software applications that revolve around machinery instead of transformation of information and data, **do not provide a predictable and deterministic RT response for software applications as in the case of Yorktown and directly contribute to the complete systems failure that was reported.**

Courtesy: For the above explanation, please refer to the references: [1] [2] [3]

4) [50 points total] ***Form a team of 1, 2, or at most 3 students and propose a final Real-Time prototype, experiment***, or design exercise to do as a team. For the summer RT-Systems class, all groups are asked to complete at time-lapse monitoring design as outlined in requirements with more details provided in this RFP. Based upon requirements and your understanding of the UVC driver and camera systems with embedded Linux (or alternatives such as FreeRTOS or Zephyr), evaluate the services you expect to run in real-time with Cheddar, with your own analysis, and then test with tracing and profiling to determine how well your design should work for predictable response. You will complete this effort in Lab Exercise #6, making this an extended lab exercise, and ultimately you will be graded overall according to this rubric and you must submit a final report with Lab Exercise #6 as outlined here.

a) [30 pts] Your Group should submit a proposal that outlines your response to the requirements and RFP provided. This should include some research with citations (at least 3) or papers read, key methods to be used (from book references), and what you read and consulted.

b) [20 pts] Each individual should turn in a paragraph on their role in the project and an outline of what they intend to contribute (design, documentation, testing, analysis, coding, drivers, debugging, etc.). ***For groups of 2 or 3, it is paramount that you specify individual roles and contributions***.

Sol.

GROUP PROPOSAL:

As part of my efforts for the Final Project for ECEN 5623, I plan to work **individually** on the **Time-Lapse Project** required for the summer session. In this project, I will be strictly following the guidelines and requirements as mentioned in the references. To briefly summarize the key goals and objectives of my project, please refer to the below sections:

Project Goals & Objectives:

The prime goal behind the **Time-Lapse Project** is to design a SRT multimedia system that can process frames of images in SRT (using the Linux-RT environment) and can accurately measure a time length of 33 minutes and 20 seconds. To meet this objective, the Logitech C270 camera will be used. For an ideal frame rate of 1Hz, 2000 frames will be collected during the said time period and the system will implicitly measure time without any observable error based on the changes to the frames of images. The collected frames would be then converted into an MPEG video file. For a reliable testing environment, certain time-lapse methods will be used to obtain robust evidences that the system is responsive to the changes in the external environment. For efficient testing process, the use of the following elements may be used:

1. Melting of ice
2. Candlelight and its movement with respect to the environment it is present in
3. An analog clock providing time with a resolution of 1 second.

Target & Stretch Goals:

In addition to the goals outlined above, as part of an extended work, efforts will be made to modify the system so that it can run at up to 10Hz (i.e. 10 frames of images per second – 1 frame per 100 milli-seconds). As a result of this extended effort, errors ranging under 1 second arising out of the disconnect between the start and end time of the acquisition system will be addressed. Also, efforts will be made so that the system only stores unique image frames so that redundancy in the storage of image frames is eliminated. As a final note, accurate SRT time-stamping will be performed with the device name for each entry made into the logger.

The target and stretch goals can be briefly described as below:

- Use of compression techniques in SRT to reduce the storage space required by the system for frames storage in NAND flash. Efforts will be made to ensure that if implemented, this section will be handled by the thread that stores all the required unique frames to the NAND flash memory. However, once this goal has been implemented, the effective thread execution timings will change as per the requirement of CPU resources for meeting the above mentioned goal. Sufficient scheduling changes will follow the design changes based on the jitter responses acquired through the histogram studies for a set of tests.
- Effective implementation of Ethernet so that frames can be acquired remotely from the camera. This goal has not been discussed in the below but will be implemented as part of the thread acquiring frames from the camera.
- Running the test environment at higher frequencies (i.e. 10 Hz). This is a stretch goal and may require additional designs and may potentially add more to the system complexity. But, based on the availability of time, this goal will be addressed.
- The thread storing the frames of images will only accept a thread for storage into the NAND flash memory if there is a change detected which is more than 1% of the total pixel value summed. More has been discussed how this may be implemented in the below sections.

Project Architecture and Description:

As the first step towards meeting the above project goals, the following **architectural efforts** will be made:

1. Creating a SRT framework with separate threads for each execution mechanism and a sequencer to enforce a reliable scheduling policy. An RM scheduling policy will be designed based on the efforts made in the previous exercise (i.e. Exercise-4) where calculations were made for average and worst case execution times for an image frame processing thread.
2. A sequencer (CORE 0) will be designed that can run the system in SRT without any observable error.
3. Thread-1 (CORE 1) will be designed to capture frames in SRT.
4. Thread-2 (CORE 1) will be designed to process images and calculate spent time in SRT.
5. Thread-3 (CORE 2) will be designed to store images (after valid compression – as part of stretch goals) as per requirement in SRT.
6. Thread-4 (CORE 2) will be designed to provide acknowledgement using the logger in SRT.
7. Thread-5 (CORE 2) will be designed to provide an MPEG video in SRT.

To meet the above mentioned architectural design principles, the following **project description** will briefly address the high-level execution of the program in SRT:

Sequencer (CORE 0):

- The sequencer (a thread of execution) will be designed to sequence the four threads of execution as mentioned above, each allocated its own unique task with a set of commands.
- The sequencer will have the highest priority as it must sequence the threads of execution according to the RM pre-emptive scheduling policy. The Linux-RT SCHED_FIFO scheduling routine will be used for implemented the execution of POSIX threads in SRT.
- The sequencer will sequence the threads as per their reasonable SRT average and worst-case execution times (i.e. WCET).
- The execution timing and deadlines will be established through an intensive study of the SRT time response of the system for execution of each thread. After careful establishment of the fact that the system can reliably measure SRT response from the multimedia system (i.e. the Logitech C270 camera), further extended efforts will be made as defined in the goals & objectives section of this response.

Thread-1 (CORE 1):

- Thread-1 will be designed to capture frames in SRT. The frame rate will be ideally set to 1 Hz for the minimum testing requirements. If the system satisfactorily meets the testing requirement of a time length of 33 minutes and 20 seconds, efforts will be made so that the thread can acquire frames at a greater frame rate to also satisfactorily support MPEG video support provided by Thread-5.

Thread-2 (CORE 1):

- Thread-2 will be designed to process images as per continuous media transformations like transforming acquired images to RGB, grayscale or sharpened RGB/grayscale images. Appropriate aspect ratio will be shared across the executions of Threads 1 and 2 so that there is a reliable image transformation.

Thread-3 (CORE 2):

- Thread-3 will be designed to store images as a .ppm file into the NAND flash memory whenever two consecutive frames of images are unique.
- To store unique frames of images, thread-3 will monitor the frames of images fed to it by thread-2. After one unique image frame is fed to thread-3 by thread-2, it will store the frame into a buffer. Meanwhile, it will keep looking for a unique frame of image as thread-3 gets fed by thread-2 as per its periodic nature of execution. Whenever a unique frame is acquired, thread-3 will flush out the previous thread from the buffer, and it will store it into the NAND flash memory. After doing that, it will also ensure that the newly acquired unique frame of image is stored into the temporary buffer for comparison with another newly acquired frames from thread-2. Whenever there is a difference between the frames, the old one stored in the buffer will be flushed into the flash memory and the new one will be stored into the buffer as a new comparison specimen.
- As part of the stretch goals, before storage of the frames as a .ppm file into the NAND flash, the frames will be compressed in SRT. This will ensure that more frames can be stored and at least 3x the minimum number of frames can be stored on the NAND flash memory.

Thread-4 (CORE 2):

- Thread-4 will be designed to provide updates in SRT to the user using the logger system. This will be accomplished by the use of "syslog" statements. The logging process is considered as a separate thread as it will keep on an eye on thread-3. Whenever thread-3 stores a unique frame to the flash memory, thread-4 will update the logger and timestamp the event so as to make the user aware that the system is acting as expected.

Thread-5 (CORE 2):

- Thread-5 is just an extended version of Thread-2 that will process all the stored frames in the flash memory into an MPEG video file. Thread-5 will acquire a signal from Thread-4 whenever the system has counted a continuous time duration of 33 minutes and 20 seconds. When thread-5 acquires the timing signal from thread-4, it will suspend all other threads and will create an MPEG video file using the stored frames of images.

Other complimentary goals of the project:

To summarize, I have listed below the primary challenges that I may face in this project development:

1. Unbounded priority inversion
2. Blocking
3. Deadlocks
4. Unexpected Jitter responses

To fight some of the above challenges, I will use the following design approaches to develop the **Time-Lapse Project** gradually in fractions with gradual testing for SRT system response.

- Use of semaphores and mutexes to synchronize access of critical shared global resources to avoid prevent data corruption and infidelity. Plus, semaphores would signal other threads (whichever is required) to take on the CPU resources for its critical sections.

- Use of priority inheritance protocol as per the requirement to reduce chained blocking and unbounded priority inversion.
- Use of thread-indexed global variables (wherever appropriate) for multi-threaded environment to prevent race conditions and further issues of blocking.
- Use of timed locking of critical shared global resources to avoid issues of deadlocks and/or other anomalies as observed in the complicated designs of Apollo-11 and Mars Pathfinder.
- Use of "syslog" statements only as opposed to presence of DEBUG "printf" statements that can add to the Linux-RT latencies.

PERSONAL CONTRIBUTION:

I will be working INDIVIDUALLY on the above Time-Lapse Project in its entirety. Based on the goals mentioned above, I have briefly discussed the milestones that I am looking forward to acquire in the coming days to save sufficient time for debugging while ensuring a coherent and gradual development of the project.

DATES	Milestones	Goals
27th Jul – 30th Jul	**DEBUG-1** ***Minimum Requirements***	**First Development:** Primary design for limited testing – only two threads and a sequencer to test for image capturing, and processing & storage in SRT using appropriate RM scheduling and Linux-RT SCHED_FIFO policies along with safe synchronization primitives like semaphores & mutexes. Debugging for primary stage bugs in the source files.
30th Jul – 3rd Aug	**DEBUG-2** ***Minimum Requirements***	**Second Development:** Secondary development phase for testing of resolved issues and bugs found in the **DEBUG-1** session. Design for thread expansions into multiple cores (i.e. CORE 1 and 2) for increased implementation complexity and reduced system overloading.
3rd Aug – 8th Aug	**DEBUG-3** ***Target Requirements***	**Third Development:** Third development phase for resolving issues and bugs found in **DEBUG-2** session. Plus, efforts will be made to test the **minimum** project requirements for debugging it for errors. Efforts will be made to meet **TARGET** goals.
8th Aug – 11th Aug	**DEBUG-4** ***Final testing for Minimum & Target Requirements*** + ***Efforts to meet some of the stretch goals***	**Fourth Development:** Fourth development will involve final testing for minimum target requirements and will entail efforts to meet some of the stretch goals. Focus will be laid on running the minimum and target goals without any observable errors/issues.
11th Aug – 13th Aug	**STRETCH GOALS** + **BUFFER / DEBUG**	**Fifth Development:** Fifth development will ensure that the project minimum and target requirements are fully met while efforts will be made to meet as many stretch goals as possible. Essentially, debugging will be performed for the stretch goals for any errors/issues that are unresolved.
13th Aug – 15th Aug	**STRETCH GOALS** + **BUFFER / DEBUG**	**Sixth Development:** Sixth development will see some final efforts for meeting all the stretch goals if possible in the available time. The system will be thoroughly debugged and retested to see that it successfully meets the minimum and target requirements without any glitches and anomalies.
16th Aug	**DEMO**	***Final Project Demo***

Courtesy:

For the above project proposal and contribution, I have referred to the below resources on several occasions to come up with the project design that I have briefly discussed:

[1] [2] [3] [4] [5]

Other auxiliary references:

[6] [7]