ECE 315 Assignment 1

Arun Woosaree
XXXXXXX

February 5, 2020

1

Hard real-time vs. Soft real-time embedded systems

Similarities

- With both, we want to avoid violating real-time constraints

Differences

- Violations of real-time constraints are undesirable, but tolerated in soft real-time systems while in hard real-time systems, violations of real-time constraints are unacceptable (a violation for a hard real-time system would cause catastrophic failure, or even death.)

When selecting an implementation strategy, it is useful to distinguish between hard and soft real-time systems because you want to select a strategy that is appropriate. You want to consider what might happen in the case of failure to meet real-time constraints. If a violation of these constraints would cause catastrophic damage, such as loss of life, you have a hard real-time system, so your implementation strategy should take that into account. If you have a soft real time system, and failure to meet the real-time constraints is not catastrophic, your implementation strategy does not need to avoid failure to the same extent as it would need to for a hard real-time system.

For example, you probably would have a different implementation strategy for making a pacemaker versus a vending machine. In one scenario, a person's life is dependent, while in the other, a person may or may not get a snack.

2

2.1 Cloud Computing

As per ISO/IEC 17788:2014 :

> “ Cloud computing is a paradigm for enabling network access to a scalable and elastic pool of shareable physical or virtual resources with self-service provisioning and administration on-demand. The cloud computing paradigm is composed of key characteristics, cloud computing roles and activities, cloud capabilities types and cloud service categories, cloud deployment models and cloud computing cross cutting aspects that are briefly described in [clause 6 of ISO/IEC 17788:2014]. ”

Advantages:

- The cloud computing provider takes care of purchasing and maintaining hardware, not the user
- Allows for flexibility. The user can scale services as they need on-demand. That way, a user only pays for the resources they use. For example, the user can request more computing resources when their service has more traffic, but when there is less traffic, the user can scale down their operations to save cost
- Reduces barrier of entry for deploying an application. One can deploy a toy project they've been working on for a few dollars, without investing in the up-front cost of purchasing a mainframe, maintaining it, and selling the computer when they're done.

Overall, I think this term is extremely well-defined, considering there is an ISO specification that is 10 pages long, and is very thorough in defining everything. It is, after all the international body for defining standards...

2.2 Edge Computing

According to Cloudflare:

> “ Edge computing is a networking philosophy focused on bringing computing as close to the source of data as possible in order to reduce latency and bandwidth use. In simpler terms, edge computing means running fewer processes in the cloud and moving those processes to local places, such as on a user's computer, an IoT device, or an edge server. Bringing computation to the network's edge minimizes the amount of long-distance communication that has to happen between a client and server. ”

Advantages:

- Lower latency (better user experience, streaming and gaming applications for example. Users would experience higher quality streams and faster response times for games)
- Less bandwidth and server resource usage, and therefore lower costs associated with these resources

Overall, it is not very specifically defined. However, most sources seem to agree that the general idea is moving cloud computing resources geographically closer to users.

2.3 Fog Computing

According to the paper "Finding your way in the fog: Towards a comprehensive definition of fog computing" by L. M. Vaquero and L. Rodero-Merino:

> "Fog computing is a scenario where a huge number of heterogeneous (wireless and sometimes autonomous) ubiquitous and decentralised devices communicate and potentially cooperate among them and with the network to perform storage and processing tasks without the intervention of thirdparties. These tasks can be for supporting basic network functions or new services and applications that run in a sandboxed environment. Users leasing part of their devices to host these services get incentives for doing so."

Advantages:

- even lower latency and utilizing resources like bandwidth more efficiently compared to edge computing. It brings the edge closer to the user, kind of like how fog is closer to the ground than clouds
- allows for data to be sent to a local node for faster proccessing and response times. An example application would be sensors in a factory which need to automatically shut down equipment if something goes wrong

Overall, this term is fairly well-defined. There are multiple papers, IEEE articles and the like which give in-depth definitions of fog computing which seem to agree with each other.

2.4 Mist Computing

According to the National Institute of Standards and Technology,

> "Mist computing is a lightweight and rudimentary form of fog computing that resides directly within the network fabric at the edge of the network fabric, bringing the fog computing layer closer to the smart end-devices. Mist computing uses microcomputers and microcontrollers to feed into fog computing nodes and potentially onward towards the centralized (cloud) computing services."

Advantages:

- similar to how fog computing takes edge computing a step further, mist computing takes it a step further to reduce latency, and better use resources like processing power

- no need to connect to the 'cloud' to perform decisions, but overall telemetry data can still be aggregated by connecting to the broader internet Allows for applications like self-driving cars to make decisions locally where decisions have to be made quickly.

Overall, this term is not well-defined. Finding a definition was difficult to begin with. Additionally, the sources that mention mist computing seem to talk more about fog computing than mist computing. However, generally, they agree on the local processing aspect of mist computing.

3

1. Military applications. Say for example, a missile was being launched, and is guided by a computer system. When a missile is launched there were (hopefully) multiple humans who made the decision to do so, and they were sure of their actions. If the system was able to be overridden by humans, that gives the enemy a chance to disarm or take control of the missile, which is not ideal. In this case, the missile should lock out human and override human control to deliver its payload to the intended recipient.

2. Antilock braking systems. Humans are not great at pressing on their car's brakes just enough so that there is static friction, as opposed to kinetic friction. If static friction is maintained, the car can come to a stop quicker and also prevents the car from skidding so that the driver can steer the car more easily and potentially avoid a collision. In this case, the computer forces the brake pedal back up, overriding the human's action of pushing the pedal down. This is a feature that has saved many lives.

3. Safety systems. Imagine a computer at a manufacturing plant which has shut down operations due to one of its sensors reporting a value which results in an unsafe condition. An uninformed worker, or maybe a malpracticing human prioritizing production over safety might try to resume operations without fixing the cause of production being shut down. In this case, it makes sense for the computer to lock out human control for resuming operations until the safety issue is fixed.

4

A pointer can be created in the following ways:

1. You can ask the OS to allocate a region of memory using `malloc()`, `calloc()`, or `realloc()`, and give you a pointer to that region of memory. It is up to the programmer to check if NULL was returned and deal with error handling.

2. The `&` operator is used to obtain a pointer to an existing piece of data

3. math is done on an existing pointer, and the result is stored as a new pointer. (e.g. `char* new_ptr = other_ptr + 8;`)

4. A pointer can be declared but uninitialized. A so-called wild pointer can point to anywhere in memory and therefore should not be used. A pointer should be initialized with a value before use.

5. A pointer can be declared and initialized to a specific address (e.g. `void* ptr = 0x93784928;`).

Throughout a pointer's life, it can be modified and overwritten. An example of a pointer being modified would be a typed pointer being incremented or decremented before or after they are used. (e.g. `*(ptr++)`. Void pointers, which can be used to point to anything can be casted to a different type before being modified (e.g. `ptr = (char* ptr)`) as well. Also, a pointer can be overwritten by simply assigning another pointer value to the existing pointer. Pointers are also usually dereferenced in their lifetime using the * operator. This allows for the data that the pointer points to, to be accessed.

When the object a pointer points to no longer needs to be used, a responsible C programmer will call `free()` to recycle the memory. Failure to call `free()` after the object is no longer required results in what is known as a memory leak.

The programmer should ensure that other potential pointers pointing to the object are no longer being used! Once the block of memory that a pointer points to is freed using the `free()` function, the pointer can be safely discarded. Actually, the pointer should no longer be used, because the pointer now points to a region of memory that has been recycled. This is called a dangling pointer, dereferencing a dangling pointer and interacting with the memory it points to results in undefined behaviour.

5

A memory leak happens when a dynamic object is not freed after it is no longer required. Even though the program no longer needs the object, it holds on to the precious space in memory and does not allow for that space to be recycled to allow new objects to take up that space. Left unchecked, the system will run out of memory, degrade in performance, and eventually cause system failure.

Memory leaks and initialization errors can be avoided using static analysis tools in your code editor.

Tools like dmalloc and mpatrol can be used as drop-in replacements for `malloc()` and they can help keep track of allocations to help the programmer fix memory leaks.

Another strategy for embedded systems to avoid memory leaks is to avoid dynamic allocation altogether, and pre-allocate all the memory you need for the duration of the program. This is known as static allocation. This is what NASA does.

Smart pointers can be used in place of raw pointers to help with avoiding initialization errors.

When the object no longer needs to be used, the destructor belonging to the software object can be is called, which is supposed to clean up the object, thereby avoiding a memory leak.

The RAII, or Resource Aquisision is Initialization technique can also be used to help with avoiding memory leaks and uninitialized pointers.

6

Buffer overflow happens then elements are added to a buffer (implemented as an array) and they go beyond the buffer's capacity. This can be dangerous, since it allows for arbitrary data to potentially overwrite other data which might be in use, or code. This can be a security issue which allows an attacker to change data that they should not have access to, or to modify code that will be run. This can also result in system instability and crashes, since unexpectedly changing data results in undefined behaviour.

Embedded systems are generally more vulnerable to buffer overflow, since they usually do not offer the same protections as a fully fledged operating system.

In C, these problems can be avoided by using `strncpy()` as opposed to `strcpy()`, and `fgets()` instead of `gets()`. Bounds checking can also help with preventing buffer overflows. If the programmer makes sure that they never attempt to go out of the bounds of the buffer, overflow won't happen.

7

- Most of MicroC/OS is written in standard ANSI-C. ANSI-C is a highly portable language.
- The features of MicroC/OS-II can be accessed using subroutine calls from applications written in C.
- The software architecture of MicroC/OS was partitioned to hide machine dependencies within a relatively small and well-defined processor port. This means that a small amount of code needs to be changed to port the operating system to a new platform, therefore making it easier to port
- MicroC/OS does not make many assumptions about the hardware. There must be a CPU to execute compiled ANSI software, and a hardware timer resource to provide software-independent timing for use in functions with programmable real-time time delays.

8

The system would fall behind on its real-time guarantees if the amount of idle time was reduced below a safe percentage of the CPU's execution time.

For externally initiated events, too little idle time would mean not enough excess CPU capacity so that the worst-case bursts of event-handling workloads would not be able to be handled within the maximum response time variability specifications.

For internally initiated events, not enough idle time would mean that the effects of internal sources of timing variability (e.g., variability in the execution time of compiled software, cache memory and virtual memory, and DMA activity) would not be absorbed and therefore not hidden.

You would start to see problems like the system being less responsive, and longer wait times for tasks to complete due to lower priority tasks not getting a chance to run as often. (With the idle task being the lowest priority task, if it does not run often, that means that other lower priority tasks likely also did not get many chances to run.)

9

9.1 Major elements of this architecture

The super short version:

1. it's single threaded
2. it's non-preemptive
3. there is a main loop, where most of the processing is done
4. Interrupt Service Routines (ISR) respond to critical events

A little more in-depth:

1. foreground-background systems:
   Background processing: The main loop can meet less challenging timing constraints through normal instruction execution.
   Foreground processing: ISRs provide faster response for critical events (assuming interrupt hardware is present)
2. sleep mode with interrupts: This allows the microcomputer to go in a low-power sleep state that can be woken up with external interrupt signals that exceed a specified interrupt priority level.
3. one non-preemptive loop with no interrupts: A number of input sources or devices are polled, it iterates at a frequency that is at least the fastest required polling frequency. The longest possible execution time in the loop must be less than the loop period, and the frequency is determined by a hardware timer. A waiting sub-loop at the end of the main loop safely uses up excess time at the end of each iteration of the main loop.

4. one non-preemptive loop with interrupts: We enhance a single non-preemptive loop with a few interrupts to ensure that hardware-triggered events are serviced promptly and deterministically. It can enable data processing by signalling a flag or semaphore that allows software outside of the ISR (which should be short) to proceed later after a small delay, and it also keeps the processor lightly loaded so that the worst-case execution of ISRs does not compromise the real-time performance of the main processing loop.

5. multiple non-preemptive loops with interrupts: Two or more non-preemptive loops which should be implemented using one software thread are used that iterate at different harmonic periods. Interrupts are used to provide fast and deterministic response to external events. Executions of the multiple loops are enforced using a single hardware timer to ensure timing accuracy. Most data processing is done outside the ISRs in the one main software thread, so the ISR should be kept short. This keeps the processor lightly loaded so that even the worst-case execution of ISRs does not compromise the real-time performance of the multiple processing loops. Also, each polled input or device is assigned to the one loop that iterates just fast enough to meet its real-time constraints

9.2 How it simplifies meeting these real-time specifications

a) maximum response times: Using one non-preemptive loop gives us a guaranteed maximum response time. Using one non-preemptive loop with interrupts, we get predictable maximum response time for both polled devices and external events. Having a predictable maximum response time makes it easy to determine if we will meet the maximum response time criteria

b) maximum variability in the response times: A long main loop might actually result in overly variable response times to input events, but in general, this should be lower in a non-preemptive loop architecture compared to a preemptive one since exceptions and internal signals would contribute to the variability, but for a preemptive architectures, exceptions, internal signals, and preemptive tasks contribute to the variability in response times.

c) accuracy of repetition frequencies: A non-preemptive system is likely to have a much more predictive repetition frequency compared to a preemptive one, since in the non-preemptive system, this repetition frequency is based on the loop, while in a preemptive system there is overhead with running the OS among other factors which result in less predictable repetition frequencies compared to a non-preemptive system.

d) maximum variability in the repetition frequencies: For the same reason as c) above, the maximum variability in the repetition frequencies will

overall be less variable and more consistent in a non-preemptive system compared to a preemptive system due to the looping architecture in the non-preemptive system being much more predictable than running an operating system that has extra overhead.

e) control of degradation behaviour: In preemptive systems, the control of degradation behaviour is actually implicitly handled through prioritization, but for non-preemptive systems this behaviour has to be explicitly implemented. Luckily, ISRs can be used to provide a faster response for critical events such as if the battery is nearly exhausted. ISRs are supposed to not compromise the real-time performance of the system, and the small overhead they add does not compare the overhead of a fully-fledged operating system that uses preemptive scheduling.