

# Exercise 4

C++ Sequencer and Services

Students

Bella Wolf

Nalin Saxena

Instructor

Steve Rizor

<b>Problem 1.....</b>	<b>3</b>
Describe the 3 different kinds of pipeline hazards. Which is the most impactful? What are some ways to avoid the impact?.....	3
What is branchless programming, where might it be most useful? Should you proactively pursue this optimization?.....	6
What do you notice about integer division in the instruction table? How might this inform your coding behavior?.....	6
<b>Problem 2.....</b>	<b>7</b>
What are two rules that you found the most surprising? Why do you find it surprising, and do you agree or disagree with the rules?.....	7
What are two rules that make the most sense to you? Why do they stand out to you?.....	8
Implement a pathological “hello world” program and makefile that breaks every single rule..	9
<b>Problem 3.....</b>	<b>11</b>
Compare the canonical service example from the lecture slides. What do you notice is the same or different?.....	11
Creating a Makefile and building the sample code with modifications to implement all of the todo items. Documentation of what was learned while implementing and any difficulties.....	12
Implement your prior fib10 and fib20 synthetic workload services in this C++ model.....	14
Enhance the Service class to record execution time statistics for the service (from release to completion of the release), including at least min/max/average/execution time jitter/start time jitter for the service. Log the statistics for each service during shutdown. Document your results and comment on how useful you do or don’t see this enhancement.....	17
<b>Appendix:.....</b>	<b>21</b>
File 3b.cpp.....	21
File 3b.hpp.....	23
File Fibo_Sequencer.cpp.....	29
File Fibo_Sequencer.hpp.....	32

# Problem 1

*Read chapter 3 of Instruction-Level Parallelism*

Describe the 3 different kinds of pipeline hazards. Which is the most impactful? What are some ways to avoid the impact?

Pipelining is the instruction of subsequent instruction without waiting for a previous instruction to complete. It increases overall throughput of the system.

Pipelining is the action of passing instructions one after another such that the prior instructions don't fully process before another starts to process thereby increasing the overall throughput of the system.

Pipeline is architecture dependent and the number of steps in pipeline can vary for example:-

1. Arm Cortex M3 follows a 3 step pipeline **Fetch,Decode,Execute** ([Source](#))
2. Arm Cortex M9 follows a 5 step pipeline **Fetch,Decode,Execute,Memory,Write**.

There are potential downsides that come with a pipelined system. They might be somewhat complex to implement. These hazards are commonly grouped under a term called pipeline hazards. There are 3 main types of pipeline hazards -

- **Structural Hazard** - A structural hazard is caused by two or more instructions needing the same part of the cpu at the same time , for example a file of register or ARF (Architectural Register File) may only support a read or write operation at a given time (Von neumann Architecture). Hence this issue might arise if a **Fetch** and **Memory** step have a clash. Similar issues can also arise if for example ALU is required by two instructions.
- **Data Hazard**- Data Hazard are caused due to a dependency of a current instruction to wait for a previous result or instruction to finish.

These can be subcategorized into 3 main categories

**Read After Write Hazard (RAW)**  
**Write After Read Hazard (WAR)**  
**Write After Write Hazard (WAW)**

Lets us take an example for RAW-

```
ADD R0,R1,R2
ADD R4,R0,R3
```

In case the above two instructions are pipelined and try executing at the same time this will lead to erroneous results in case the second instruction tries to read the value of R0 before the first instruction updates it.

- **Control Hazards-** Control hazards can also be seen as a type of data hazard where the next instruction to be executed is uncertain (Program counter value is not known for certain). This happens in case of branching instructions such as BEQ, JMP.

```
MOV R1, #2      ; Load the value 2 into register R1
MOV R2, #3      ; Load the value 3 into register R2
ADD R3, R1, R2  ; R3 = R1 + R2 (R3 = 2 + 3)

CMP R3, #4      ; Compare R3 with 4
BEQ branch_label ; If R3 == 4, branch to branch_label

MOV R3, #0      ; If R3 != 4, set R3 to 0
B end_label     ; Jump to end_label to avoid executing branch_label

branch_label:
ADD R3, R3, #1  ; If R3 == 4, increment R3 by 1 (R3 = 4 + 1)

end_label:
; Continue with the rest of your program here
```

The above code can introduce 2 pipeline hazards

1. CMP instructions relies on the value of R3
2. It is not known which instruction should be fetched till BEQ instructions reach its execution state.

	A	B	C	D	E	F	G	H	I	J	K	L	M
IF		ID	EX	MEM	WR	MOV R1, #2							
		IF	ID	EX	MEM	WR	MOV R2, #3						
			IF	ID	EX	MEM	WR	ADD R3, R1, R2					
				IF	ID	EX	MEM	WR	CMP R3, #4	(data race ?)			
					??	??	??	??	??	BEQ branch_label		(control hazard)	

Control hazards are the worst and most detrimental to performance. A large portion of the pipeline has to be completely thrown out and flushed.

Below are a few ways to mitigate the effects of pipeline hazards.

### Structural Hazard mitigation

- Wait for the particular resource to become free.
- If possible it might be a good idea to add more hardware to cpu in case the stalls are more often due to a single resource.
- To prevent bottlenecks of having the same memory for code and data (as in Von Neumann) it is better to use Harvard architecture.

### Data Hazard mitigation

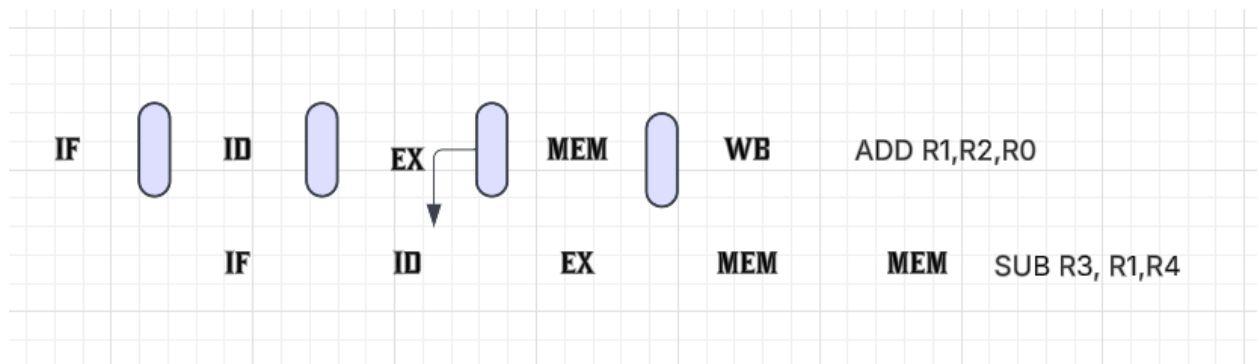
- Operand Forwarding- A buffer register can hold the value of a necessary computation required for next instruction and **“forward”** it.

For example if we have an instruction sequence as

ADD R1,R2,R0

SUB R3, R1,R4

A buffer register ( shown as blue) can pass the value at the execute stage to the decode of next instruction so that the pipeline is not stalled



### Control Hazard mitigation

- To prevent flushing the entire pipeline when a branch instruction uncertainty is encountered, **NOP** instructions can be added to the pipeline before fetching new instructions.
- Another method is to optimize and write branchless code.
- Adding compiler hints can help in case of speculative execution (in case it's known in advance that a branch is more likely to happen).

## What is branchless programming, where might it be most useful? Should you proactively pursue this optimization?

The goal of a branchless program is to eliminate branches caused by a “decision” made in the program through elements like conditionals. To achieve this goal the use of arithmetic, bitwise, or predicated instructions are used as a way to compute both potential outcomes without branching. As a result there is a reduction of branch mispredictions, preventing pipeline stalls and improving execution efficiency.

There are a number of different cases where branchless programming would be useful but the most useful cases are when the potential for mispredictions is high and/or very costly to the system.

While branchless programming can be a great optimization technique it's not always needed and can occasionally cause more harm than good in certain cases. If a branch is highly predictable then it makes much more sense to utilize the prediction since correct predictions execute faster than computing both outcomes branchlessly.

## What do you notice about integer division in the instruction table? How might this inform your coding behavior?

The value of throughput and latency is very high for the DIV instruction as compared to other instructions. Meaning it would take a long time to compute and the instruction is relatively slow. This is due to the necessity of multiple ALU stages and needing a lot of silicon area. Additionally, DIV cannot be efficiently pipelined which further reduces performance.

How would this impact our code-

1. Precompute division values - If possible we can precompute the values and store it in a macro rather than recomputing them unnecessarily inside a loop
2. Avoid using Division operation altogether.
3. Use the Right shift operator(>>) in case division is performed by 2.

## Problem 2

*Read The Power of 10: Rules for Developing Safety-Critical Code*

What are two rules that you found the most surprising? Why do you find it surprising, and do you agree or disagree with the rules?

The first rule that we find surprising is “**Do not use dynamic memory allocation after initialization.**” (RULE-3)

The reason this is surprising is that it is usually a bad practice to allocate large data structures such as arrays and large structs on the stack since a lot of operations rely on stack use such as function calls, local variable allocation. Using dynamic memory allocation also has the advantage of resizing these data structures with the help of **realloc** and other api(s).

However I agree with this rule since memory leaks can be really hard to debug and solve. Generally, the heap grows against the stack and a runaway memory leak can potentially crash the entire system. There are other potential issues such as the problem of **double free** which can happen if we are not careful with dynamic memory.

The second rule that we find surprising is “**No function should be longer than what can be printed on a single sheet of paper in a standard format with one line per statement and one line per declaration. Typically, this means no more than about 60 lines of code per function.**”

This rule is similar to the “S” in **SOLID** principles of modern day programming which stands for **Single Responsibility Principle**. This principle improves the ease of testing as each function would have one unique functionality.

While there are benefits of using this rule we believe that overuse of this can cause over fragmentation of code as it would lead to functions calling other functions which might be difficult to track and analyze. It could also result in increased overhead of function calls and more complex error handling.

## What are two rules that make the most sense to you? Why do they stand out to you?

The first rule that makes most sense is to use recursion and jump statements. In embedded systems with limited amounts of stack space recursion can introduce a massive overhead of the function making multiple calls to itself. For example a simple code to calculate factorial of a number is done with and without recursion and the stack usage is analyzed with **-fstack-usage**. More stack memory is used for the Recursion case. Recursion also creates very complex and hard to debug call trees and usually an iterative solution is possible.

The same goes with jump statements such as goto also add non linear code flow which is difficult to analyze and can introduce logical errors.

Recursion	Non Recursion
<pre>#include &lt;stdio.h&gt;  long unsigned calc_fact(int num) {     if (num == 0    num == 1) {         return 1;     }     return num * calc_fact(num - 1); }  int main() {     long unsigned int ans = calc_fact(20);     printf("factorial %lu\n", ans);     return 0; }</pre>	<pre>#include &lt;stdio.h&gt;  long unsigned calc_fact(int num) {     long unsigned ans = 1;     for (long unsigned i = 1; i &lt;= num; i++)     {         ans *= i;     }     return ans; }  int main() {     long unsigned int ans = calc_fact(20);     printf("factorial %lu", ans); }</pre>
<pre>./recursion.c:3:15:calc_fact 48 static ./recursion.c:10:5:main      32 static</pre>	<pre>no_recursion.c:3:15:calc_fact 16      static no_recursion.c:13:5:main      32 static</pre>



The second rule that we most agree with is “**Declare all data objects at the smallest possible level of scope.**” If variables are declared with the smallest possible scope there is a lesser chance of data corruption and inadvertently modifying the value. Ideally an interface such as getters and setters should be exposed to modify and access the value of the variables. Following this rule also enhances the code readability and maintainability. Any change related to fetching and setting has to be made only at the getter and setter methods.

Implement a pathological “hello world” program and makefile that breaks every single rule.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
// RULE 8 BROKEN USE OF CONDITIONAL MACROS
#ifdef DEBUG
#define LOG(fmt, ...) printf(fmt, __VA_ARGS__)
#else
#define LOG(fmt, ...) // Empty statement, no logging
#endif

int str_len = 0; // RULE 6 BROKEN : DECLARED str_len as global only used
in main

// RULE 1 BROKEN USE OF RECURSION
int recursive_str_len(char *ch)
{
    if (*ch == '\\0')
    {
        return 0;
    }
    return 1 + recursive_str_len(ch + 1);
}

//RULE 4 broken function having multiple Responsibilities
void function()
{
```

```
char chr[] = "Evil Hello World!";

int (*func_ptr)(char *ch) = recursive_str_len; // RULE 9 BROKEN NUSE OF
FUNCTION POINTERS
str_len = func_ptr(chr);
LOG("string length of %s is %d \n", chr, str_len); // RULE 7 BROKEN NOT
CHECKING RETURN VALUE OF PRINTF
assert(str_len > 0); // RULE 5 BROKEN NOT
ENOUGH ASSERTIONS

char *dyna_str = malloc(sizeof(char) * str_len); // RULE 3 BROKEN USE
OF MALLOC
int itr = 0;
while (itr < str_len)
{ // RULE 2 BROKEN NO MAX LIMIT ON LOOP
    dyna_str[itr] = chr[itr];
    itr++;
}
dyna_str[itr] = '\0';
LOG("dynamic memory %s", dyna_str);
}

int main()
{
    function();
}
```

## Makefile

```
CC = gcc
TARGET = evil_hello_world
SRC = evil_hello_world.c
#borken rule 10 no compiler flags for -Wall -Werror and pedantic
$(TARGET): $(SRC)
    $(CC) -o $(TARGET) $(SRC)

clean:
    rm -f $(TARGET)
```

## Problem 3

*Look at the provided sample C++ implementation of the Sequencer and canonical Service using Linux threads. Compare it to the canonical Service from the lecture slides.*

Compare the canonical service example from the lecture slides.  
What do you notice is the same or different?

Some key Similarities between the C and C++ implementation are

- Both approaches are using similar thread based implementations
- The Service release mechanism is common in both which is using a semaphore
- The method of provide\_service is similar in both , to wait and acquire a semaphore and perform the necessary action.
- The general structure and idea of the initialization shutdown framework is similar.

### **Key Differences**

<b>Canonical C implementation</b>	<b>C++ implementation</b>
The <pthread> implementation requires the threads to be manually joined once the service is completed.	The <thread> library uses a more advanced <code>jthread</code> to create threads which supports autojoin of threads on destruction
This approach is completely based on functions.	This approach makes use of classes, The class Sequencer has data members of the class Services and can control and manage services through its own member functions.
Initialization functions which set the priorities and affinity have to be called manually	The Constructor will be automatically invoked

Creating a Makefile and building the sample code with modifications to implement all of the todo items. Documentation of what was learned while implementing and any difficulties.

The modified code is linked in the Appendix section as **Sequencer.hpp** and **Sequencer.cpp**.

Some of the key learnings and new concepts are-

- template classes and constructors
- Initialization list in constructors
- Argument forwarding
- The construct or idea of having an array of objects of one class in another class.
- Vector `emplace_back` syntax.

One of the key challenges I faced is while adding a `binary_semaphore` and atomic variable to the Service class I was facing the below error.

```
nalin@Nalin:~/Desktop/RTEX_EX_4$ g++ --std=c++23 -Wall -Werror -pedantic Sequencer.cpp -o Sequencer

In file included from /usr/include/c++/11/vector:66,
                 from /usr/include/c++/11/functional:62,
                 from Sequencer.hpp:13,
                 from Sequencer.cpp:11:
/usr/include/c++/11/bits/stl_uninitialized.h: In instantiation of 'ForwardIterator std::uninitialized_copy(InputIterator, InputIterator, ForwardIterator) [with InputIterator = std::move_iterator<Service*>; ForwardIterator = Service*]':
/usr/include/c++/11/bits/stl_uninitialized.h:333:37:   required from 'ForwardIterator std::uninitialized_copy_a(InputIterator, InputIterator, ForwardIterator, std::allocator<Tp>&) [with InputIterator = std::move_iterator<Service*>; ForwardIterator = Service*; _Tp = Service]'
/usr/include/c++/11/bits/stl_uninitialized.h:355:2:   required from 'ForwardIterator std::uninitialized_move_if_noexcept_a(InputIterator, InputIterator, ForwardIterator, Allocator&) [with InputIterator = Service*; ForwardIterator = Service*; Allocator = std::allocator<Service>]'
/usr/include/c++/11/bits/vector.tcc:474:3:   required from 'void std::vector<Tp, _Alloc>::_M_realloc_insert(std::vector<Tp, _Alloc>::iterator, Args&& ...) [with Args = {main()::lambda()>, int, int, int}; _Tp = Service; _Alloc = std::allocator<Service>; std::vector<Tp, _Alloc>::iterator = std::vector<Service>::iterator]'
/usr/include/c++/11/bits/vector.tcc:121:21:   required from 'std::vector<Tp, _Alloc>::reference std::vector<Tp, _Alloc>::emplace_back(Args&& ...) [with Args = {main()::lambda()>, int, int, int}; _Tp = Service; _Alloc = std::allocator<Service>; std::vector<Tp, _Alloc>::reference = Service&]'
Sequencer.hpp:86:31:   required from 'void Sequencer::addService(Args&& ...) [with Args = {main()::lambda()>, int, int, int}]'
Sequencer.cpp:22:25:   required from here
/usr/include/c++/11/bits/stl_uninitialized.h:138:72: error: static assertion failed: result type must be constructible from value type of input range
   138 |         static_assert(is_constructible<ValueType2, decltype(*_first)>::value,
       |                        ^~~~~~
/usr/include/c++/11/bits/stl_uninitialized.h:138:72: note: 'std::integral_constant<bool, false>::value' evaluates to false
```

After consulting with the course instructor and reading some threads online I was able to understand that vectors use dynamic memory allocation and might move and copy data in memory when new elements are added or the vector resizes. This is a problem since by default the atomic and semaphore types are non movable and copyable.

The solution implemented is to modify the way `_service` vector stores the objects of the Service class. The idea is to use **smart pointers** so that only a pointer of the object is stored in the vector.

```
std::vector<std::unique_ptr<Service>> _services;
```

The Code is linked in the appendix as **3b.hpp** and **3b.cpp**. The code was run for a duration of 50ms. By visualizing inspecting we can see that the service1 got executed 10 times and service 2 was run for 5 times which is expected

```
nalin@raspberrypi: ~  
File Edit Tabs Help  
nalin@raspberrypi:~/Desktop/excercise3b $ make all  
g++ -std=c++23 -Wall -pedantic -pthread -c 3b.cpp -o 3b.o  
g++ -std=c++23 -Wall -pedantic -pthread -o 3b 3b.o  
nalin@raspberrypi:~/Desktop/excercise3b $ sudo ./3b  
LOG_MSG[33564]: SYSLOG TEST MESSAGE!  
LOG_MSG[33564]: constructor called  
LOG_MSG[33564]: affinity 1  
LOG_MSG[33564]: priority 99  
LOG_MSG[33564]: period 5  
LOG_MSG[33564]: constructor called  
LOG_MSG[33564]: affinity 1  
LOG_MSG[33564]: priority 98  
LOG_MSG[33564]: period 10  
LOG_MSG[33564]: Initializing service...  
LOG_MSG[33564]: Initializing service...  
LOG_MSG[33564]: Service initialized with affinity 1 and priority 99  
LOG_MSG[33564]: Service initialized with affinity 1 and priority 98  
this is service 1 implemented in a lambda expression  
  
this is service 1 implemented in a lambda expression  
  
this is service 2 implemented as a function  
  
this is service 1 implemented in a lambda expression  
  
this is service 1 implemented in a lambda expression  
  
this is service 2 implemented as a function  
  
this is service 1 implemented in a lambda expression  
  
this is service 1 implemented in a lambda expression  
  
this is service 2 implemented as a function  
  
this is service 1 implemented in a lambda expression  
  
this is service 1 implemented in a lambda expression  
  
this is service 2 implemented as a function  
  
this is service 1 implemented in a lambda expression  
  
this is service 1 implemented in a lambda expression  
  
LOG_MSG[33564]: Services stopped, exiting...  
this is service 2 implemented as a function  
nalin@raspberrypi:~/Desktop/excercise3b $
```

## Implement your prior fib10 and fib20 synthetic workload services in this C++ model

The Code is linked in the appendix as *Fibo\_Sequencer.hpp* and *Fibo\_Sequencer.cpp*

Brief description of code flow-

First a simple iteration count function is implemented to get an estimated value of the number of iterations we need to run the fibonacci function to get an estimate of 10ms and 20ms. This test runs for **PRE\_TEST** which can be controlled with a macro. These values will be later used to generate the required synthetic load.

```
void run_iteration_test()
{
    int accum_iter_fib10 = 0;
    int accum_iter_fib20 = 0;
    // Warm-up computation
    FIB_TEST(seqIterations, FIB_TEST_CYCLES);

    for (int i = 0; i < PRE_TEST; i++)
    {
        struct timespec start_time, finish_time, thread_dt;

        clock_gettime(CLOCK_REALTIME, &start_time); // Get start time
        FIB_TEST(seqIterations, FIB_TEST_CYCLES);
        clock_gettime(CLOCK_REALTIME, &finish_time); // Get end time

        delta_t(&finish_time, &start_time, &thread_dt);
        double run_time_calc = (thread_dt.tv_sec * 1000.0) + (thread_dt.tv_nsec /
1000000.0);
        int req_iterations_fib10 = (int)(10 / run_time_calc);
        int req_iterations_fib20 = (int)(18.5 / run_time_calc);
        accum_iter_fib10 += req_iterations_fib10;
        accum_iter_fib20 += req_iterations_fib20;
    }
    ten_ms_count = (int)accum_iter_fib10 / PRE_TEST;
    twenty_ms_count = (int)accum_iter_fib20 / PRE_TEST;
    syslog(LOG_INFO, "RUNNING fib iteration test! %d %d", ten_ms_count,
twenty_ms_count);
}
```

```
}
```

Next we create an instance of the Sequencer class and start adding our services. We have added a field to identify the service which is currently running. The addService call will trigger the call to Constructor and create threads and assign priority with the semaphore in a blocked state. A small delay is added to ensure that the constructors are called and only then the start services function runs.

```
Sequencer sequencer{};
sequencer.addService(fib10, 0, 98, 20, 1);
sequencer.addService(fib20, 0, 97, 50, 2);
usleep(10000); //a sleep for the initialization to complete
sequencer.startServices();
std::this_thread::sleep_for(std::chrono::milliseconds(END_TEST_MS));
sequencer.stopServices();
```

The core logic of the Sequencer is similar to the approach used in assignment 1. The sequencer runs with the highest priority of 99 and manually kicks off the services by calling the release functions for the services.


```
global_start_time = getCurrentTimeInMs();
while (_seq_running)
{
    syslog(LOG_INFO, "CI INSTANT! %f \n", getCurrentTimeInMs() -
global_start_time);
    auto &service = _services;
    service[0]->release();
    service[1]->release();

    usleep(slep20Ms);
    service[0]->release();
    syslog(LOG_INFO, "sequencer! %f \n", getCurrentTimeInMs() -
global_start_time);

    usleep(slep20Ms);
    service[0]->release();
```

```
        syslog(LOG_INFO, "sequencer! %f \n", getCurrentTimeInMs() -  
global_start_time);  
  
        usleep(slep10MS);  
        service[1]->release();  
        syslog(LOG_INFO, "sequencer! %f \n", getCurrentTimeInMs() -  
global_start_time);  
  
        usleep(slep10MS);  
        service[0]->release();  
        syslog(LOG_INFO, "sequencer! %f \n", getCurrentTimeInMs() -  
global_start_time);  
  
        usleep(slep20Ms);  
        service[0]->release();  
        syslog(LOG_INFO, "sequencer! %f \n", getCurrentTimeInMs() -  
global_start_time);  
        usleep(slep20Ms);  
    }
```





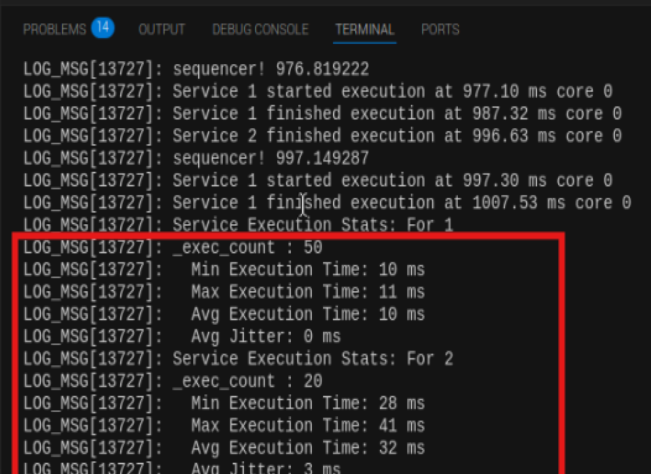
```
LOG_MSG[5555]: CI INSTANT! 0.000296
LOG_MSG[5555]: Service 1 started execution at 0.21 ms core 0
LOG_MSG[5555]: Service 1 finished execution at 10.27 ms core 0
LOG_MSG[5555]: Service 2 started execution at 10.43 ms core 0
LOG_MSG[5555]: sequencer! 20.226405
LOG_MSG[5555]: Service 1 started execution at 20.34 ms core 0
LOG_MSG[5555]: Service 1 finished execution at 30.38 ms core 0
LOG_MSG[5555]: Service 2 finished execution at 39.28 ms core 0
LOG_MSG[5555]: sequencer! 40.395459
LOG_MSG[5555]: Service 1 started execution at 40.54 ms core 0
LOG_MSG[5555]: sequencer! 50.549151
LOG_MSG[5555]: Service 1 finished execution at 50.71 ms core 0
LOG_MSG[5555]: Service 2 started execution at 50.77 ms core 0
LOG_MSG[5555]: sequencer! 60.667511
LOG_MSG[5555]: Service 1 started execution at 60.93 ms core 0
LOG_MSG[5555]: Service 1 finished execution at 70.92 ms core 0
LOG_MSG[5555]: Service 2 finished execution at 79.71 ms core 0
LOG_MSG[5555]: sequencer! 80.990082
LOG_MSG[5555]: Service 1 started execution at 81.21 ms core 0
LOG_MSG[5555]: Service 1 finished execution at 91.62 ms core 0
LOG_MSG[5555]: CI INSTANT! 101.238172
LOG_MSG[5555]: Service 1 started execution at 101.51 ms core 0
LOG_MSG[5555]: Service 1 finished execution at 111.69 ms core 0
LOG_MSG[5555]: Service 2 started execution at 112.16 ms core 0
LOG_MSG[5555]: sequencer! 121.542910
LOG_MSG[5555]: Service 1 started execution at 121.70 ms core 0
LOG_MSG[5555]: Service 1 finished execution at 131.82 ms core 0
LOG_MSG[5555]: Service 2 finished execution at 141.67 ms core 0
LOG_MSG[5555]: sequencer! 141.755167
LOG_MSG[5555]: Service 1 started execution at 141.80 ms core 0
LOG_MSG[5555]: sequencer! 152.241431
LOG_MSG[5555]: Service 1 finished execution at 152.48 ms core 0
LOG_MSG[5555]: Service 2 started execution at 152.57 ms core 0
LOG_MSG[5555]: sequencer! 162.438568
LOG_MSG[5555]: Service 1 started execution at 162.67 ms core 0
LOG_MSG[5555]: Service 1 finished execution at 172.82 ms core 0
LOG_MSG[5555]: Service 2 finished execution at 182.09 ms core 0
LOG_MSG[5555]: sequencer! 182.755639
LOG_MSG[5555]: Service 1 started execution at 183.00 ms core 0
LOG_MSG[5555]: Service 1 finished execution at 193.69 ms core 0
LOG_MSG[5555]: Service Execution Stats: For 1
LOG_MSG[5555]: _exec_count : 10
LOG_MSG[5555]: Min Execution Time: 9 ms
LOG_MSG[5555]: Max Execution Time: 10 ms
LOG_MSG[5555]: Avg Execution Time: 9 ms
LOG_MSG[5555]: Service Execution Stats: For 2
LOG_MSG[5555]: _exec_count : 4
LOG_MSG[5555]: Min Execution Time: 28 ms
LOG_MSG[5555]: Max Execution Time: 29 ms
```

With the help of logs placed before and after `_doService` calls we can understand the start and end times of the Services.

Since service 1 runs every 20Ms and Service 2 runs every 50Ms, Over an LCM of the period 100Ms we expect that Service 1 runs 5 iterations and Service 2 should run for 2 iterations.

Also since Service 1 runs at a higher priority it preempts Service2 at  $t=20$ ,  $t=60$ . **This is expected and shown highlighted in red.**

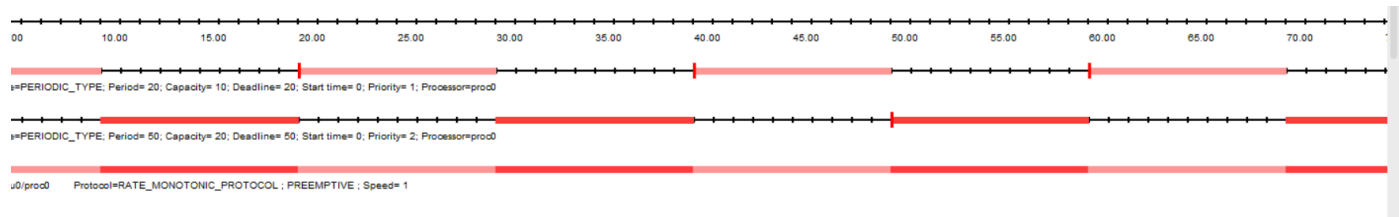
Enhance the Service class to record execution time statistics for the service (from release to completion of the release), including at least min/max/average/execution time jitter/start time jitter for the service. Log the statistics for each service during shutdown. Document your results and comment on how useful you do or don't see this enhancement.



```
LOG_MSG[13727]: sequencer! 976.819222
LOG_MSG[13727]: Service 1 started execution at 977.10 ms core 0
LOG_MSG[13727]: Service 1 finished execution at 987.32 ms core 0
LOG_MSG[13727]: Service 2 finished execution at 996.63 ms core 0
LOG_MSG[13727]: sequencer! 997.149287
LOG_MSG[13727]: Service 1 started execution at 997.30 ms core 0
LOG_MSG[13727]: Service 1 finished execution at 1007.53 ms core 0
LOG_MSG[13727]: Service Execution Stats: For 1
LOG_MSG[13727]: _exec_count : 50
LOG_MSG[13727]: Min Execution Time: 10 ms
LOG_MSG[13727]: Max Execution Time: 11 ms
LOG_MSG[13727]: Avg Execution Time: 10 ms
LOG_MSG[13727]: Avg Jitter: 0 ms
LOG_MSG[13727]: Service Execution Stats: For 2
LOG_MSG[13727]: _exec_count : 20
LOG_MSG[13727]: Min Execution Time: 28 ms
LOG_MSG[13727]: Max Execution Time: 41 ms
LOG_MSG[13727]: Avg Execution Time: 32 ms
LOG_MSG[13727]: Avg Jitter: 3 ms
```

At the end of the time period (which was set as 1000ms for this run) we print a log of each service. **The execution count of each service is in line with expectations as well.**

To confirm and support the results of the code, a simple cheddar analysis was also performed and validates the start times and end times of each service.



We think by adding these logs at the end of each test a good overall picture of the system can be established. The service start time and end times were especially important as they helped us debug a few issues while implementing the fib counts logic for service2. Ideally for calculating the number of test iterations for the fib20 case we calculate the time taken by 1 run and extrapolate to get the value for a workload of 20ms. However on doing this the service2 overshoots and misses its deadline which was later caught using these logs and minor adjustments to the interaction number were made to prevent this.

```
139     int req_iterations_fib10 = (int)(10 / run_time_calc);
140     int req_iterations_fib20 = (int)(20 / run_time_calc);
141     accum_iter_fib10 += req_iterations_fib10;
142     accum_iter_fib20 += req_iterations_fib20;
143 }
144 ten_ms_count = (int)accum_iter_fib10 / PRE_TEST;
145 twenty_ms_count = (int)accum_iter_fib20 / PRE_TEST;
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

```
LOG_MSG[13966]: priority 97
LOG_MSG[13966]: period 50
LOG_MSG[13966]: Initializing service...
LOG_MSG[13966]: Initializing service...
LOG_MSG[13966]: Service initialized with affinity 0 and priority 97
LOG_MSG[13966]: Service initialized with affinity 0 and priority 98
LOG_MSG[13966]: C:\INSTANT! 0.000518
LOG_MSG[13966]: Service 1 started execution at 0.30 ms core 0
LOG_MSG[13966]: Service 1 finished execution at 10.51 ms core 0
LOG_MSG[13966]: Service 2 started execution at 10.65 ms core 0
LOG_MSG[13966]: sequencer! 20.310234
LOG_MSG[13966]: Service 1 started execution at 20.41 ms core 0
LOG_MSG[13966]: Service 1 finished execution at 30.45 ms core 0
LOG_MSG[13966]: sequencer! 40.423473
LOG_MSG[13966]: Service 1 started execution at 40.51 ms core 0
LOG_MSG[13966]: sequencer! 50.549072
LOG_MSG[13966]: Service 1 finished execution at 50.78 ms core 0
LOG_MSG[13966]: Service 2 finished execution at 51.70 ms core 0
LOG_MSG[13966]: Service 2 started execution at 51.87 ms core 0
LOG_MSG[13966]: sequencer! 60.789761
LOG_MSG[13966]: Service 1 started execution at 60.97 ms core 0
LOG_MSG[13966]: Service 1 finished execution at 71.11 ms core 0
LOG_MSG[13966]: sequencer! 80.992961
LOG_MSG[13966]: Service 1 started execution at 81.19 ms core 0
LOG_MSG[13966]: Service 1 finished execution at 91.29 ms core 0
LOG_MSG[13966]: Service 2 finished execution at 93.52 ms core 0
LOG_MSG[13966]: Service Execution Stats: For 1
LOG_MSG[13966]: _exec_count : 5
LOG_MSG[13966]: Min Execution Time: 10 ms
LOG_MSG[13966]: Max Execution Time: 10 ms
LOG_MSG[13966]: Avg Execution Time: 10 ms
```

Since we are also capturing the execution count of each service we can also identify if any service has missed its deadlines since these are known values.

4. [5 points extra credit] Enhance the Sequencer class to sort the services by period. Don't sort manually, either use an ordered data structure or a built-in algorithm. Optimize the sequencer service processing by using an iterator to go through the list in order and saving the location in the list between service release periods (which requires a sorted list).

[partial attempt]

There are various options of having a sorted list of services below is an option that was attempted.

```
void addService(Args &&...args)
{
```

```
_services.emplace_back(std::make_unique<Service>(std::forward<Args>(args)...));  
    //use a lambda comparator function  
    std::sort(_services.begin(), _services.end(), [](const auto &a, const  
auto &b)  
        { return a->get_period() < b->get_period(); });  
}
```

The sort function in c++ takes a third argument which can be a custom comparator function. This is used to sort the list in an ascending order based on the period. There are other more advanced data structures such as map, priority queue which can store data in an ordered fashion.

Next the release of the services was fired according to elapsed time and next release time is recorded. However the scheduling results were not as per expectations and some more work is needed.

```
std::vector<uint64_t> next_release_time(_services.size());  
for (size_t i = 0; i < _services.size(); ++i)  
{  
    next_release_time[i] = 0; //simulate ci instant  
}  
  
while (_seq_running)  
{  
    double elapsed_time= getCurrentTimeInMs() - global_start_time;  
    for (size_t i = 0; i < _services.size(); ++i)  
    {  
        if (elapsed_time >= next_release_time[i])  
        {  
            _services[i]->release();  
            next_release_time[i] += _services[i]->get_period(); //  
schedule next release  
        }  
    }  
    usleep(100);  
}  
}
```

## Appendix:

### File 3b.cpp

```
/*  
  
 * 3b.cpp - Contains implementation for service and scheduler classes  
  
 *  
  
 * Original Authour-Steve Rizer 3/16/2025  
  
 * Modified: Nalin Saxena/Bella Wolf  
  
 *  
  
 * Refrence and acknowledgement- For developing the code for this excerise we  
utilized chatgpt for getting the correct  
  
 * syntax. Code related to timer_service is leveraged using chatgpt  
  
 *  
  
 * The suggestion for using unique_ptr for storing the services was provided by  
the insructor -Steve Rizer  
  
 */  
  
#include <cstdint>  
  
#include <cstdio>  
  
#include "3b.hpp"  
  
#include <csignal>  
  
volatile sig_atomic_t stop_flag = 0;
```

```
void signalHandler(int signum)
{
    if (signum == SIGINT)
    {
        syslog(LOG_INFO, "CAUGHT SIGNAL!");

        stop_flag = 1;
    }
}

void service2()
{
    std::puts("this is service 2 implemented as a function\n");
}

int main()
{
    std::signal(SIGINT, signalHandler);

    // Example use of the sequencer/service classes:

    Sequencer sequencer{};

    openlog("LOG_MSG", LOG_PID | LOG_PERROR, LOG_USER);

    syslog(LOG_INFO, "SYSLOG TEST MESSAGE!");

    sequencer.addService([]()
```

```
        { std::puts("this is service 1 implemented in a lambda  
expression\n"); }, 1, 99, 5);  
  
    sequencer.addService(service2, 1, 98, 10);  
  
    sequencer.startServices();  
  
    // todo: wait for ctrl-c or some other terminating condition  
  
    while (!stop_flag)  
  
    {  
  
        std::this_thread::sleep_for(std::chrono::milliseconds(100));  
    }  
  
    sequencer.stopServices();  
  
    syslog(LOG_INFO, "Services stopped, exiting...");  
  
    closelog();  
}
```

## File 3b.hpp

```
/*  
 * 3b.hpp - Contains implementation for service and scheduler classes  
 *  
 * Original Authour-Steve Rizor 3/16/2025  
 * Modified: Nalin Saxena/Bella Wolf  
 *  
 * Refrence and acknowledgement- For developing the code for this excerise we  
utilized chatgpt for getting the correct  
 * syntax. Code related to timer_service is leveraged using chatgpt  
 */
```

```
* The suggestion for using unique_ptr for storing the services was provided by
the instructor -Steve Rizer
*/

#pragma once

#include <cstdint>
#include <functional>
#include <thread>
#include <iostream>
#include <vector>
#include <syslog.h>

// The service class contains the service function and service parameters
// (priority, affinity, etc). It spawns a thread to run the service, configures
// the thread as required, and executes the service whenever it gets released.
class Service
{
public:
    template <typename T>
    Service(T &&doService, uint8_t affinity, uint8_t priority, uint32_t period) :
        _doService(doService),
        _affinity(affinity),
        _priority(priority),
        _period(period),
        _semaphore(0)
    {
        // todo: store service configuration values
        // todo: initialize release semaphore
    }
};
```



```
        // Start the service thread, which will begin running the given function
immediately
        syslog(LOG_INFO, "constructor called\n");
        syslog(LOG_INFO, "affinity %d \n", static_cast<int>(_affinity));
        syslog(LOG_INFO, "priority  %d \n", static_cast<int>(_priority));
        syslog(LOG_INFO, "period %d  \n", static_cast<int>(_period));
        _service = std::jthread(&Service::_provideService, this);
    }

    void stop()
    {
        // todo: change state to "not running" using an atomic variable
        // (heads up: what if the service is waiting on the semaphore when this
happens?)
        _running = false;
        _semaphore.release();
    }

    void release()
    {
        // todo: release the service using the semaphore
        _semaphore.release();
    }

    int get_period()
    {
        return _period;
    }

private:
    std::function<void(void)> _doService;
    std::jthread _service;
    int _affinity;
    int _priority;
```

```
int _period;
std::binary_semaphore _semaphore;
std::atomic<bool> _running{true}; // Atomic boolean initialized to true

void _initializeService()
{
    // todo: set affinity, priority, sched policy
    // (heads up: the thread is already running and we're in its context right
now)
    syslog(LOG_INFO, "Initializing service...");
    pthread_t threadID = pthread_self(); // Get current thread ID
    cpu_set_t cpuset;
    struct sched_param param;
    CPU_ZERO(&cpuset);
    CPU_SET(_affinity, &cpuset);
    if (pthread_setaffinity_np(threadID, sizeof(cpu_set_t), &cpuset) != 0)
    {
        syslog(LOG_ERR, "Failed to set CPU affinity for thread.");
    }
    int policy = SCHED_FIFO; // FIFO policy (use SCHED_OTHER for normal
priority)
    param.sched_priority = _priority;

    if (pthread_setschedparam(threadID, policy, &param) != 0)
    {
        syslog(LOG_ERR, "Failed to set scheduling parameters.");
    }

    syslog(LOG_INFO, "Service initialized with affinity %d and priority %d",
_affinity, _priority);
}

void _provideService()
{

```

```
        _initializeService();
        // todo: call _doService() on releases (sem acquire) while the atomic
running variable is
        while (_running)
        {
            // try to acquire sem
            _semaphore.acquire();
            _doService();
        }
    }
};

// The sequencer class contains the services set and manages
// starting/stopping the services. While the services are running,
// the sequencer releases each service at the requisite timepoint.
class Sequencer
{
public:
    template <typename... Args>
    void addService(Args &&...args)
    {
        // Add the new service to the services list,
        // constructing it in-place with the given args

        _services.emplace_back(std::make_unique<Service>(std::forward<Args>(args)...));
    }

    void startServices()
    {
        // todo: start timer(s), release services
        _seq_running = true;
        timer_thread = std::jthread(&Sequencer::timer_service, this);
    }

    void stopServices()
```

```
{
    // todo: stop timer(s), stop services
    _seq_running = false;
    for (auto &service : _services)
    {
        service->stop();
    }
}

private:
    std::vector<std::unique_ptr<Service>> _services; // Store services as
unique_ptrs
    std::atomic<bool> _seq_running{false};
    std::jthread timer_thread;
//below Section of code written with chatgpt
    void timer_service()
    {
        using Clock = std::chrono::steady_clock;
        auto start_time = Clock::now();

        std::vector<uint32_t> service_periods;
        std::vector<std::chrono::milliseconds> next_release_times;

        // Initialize the next release times for each service
        for (const auto &service : _services)
        {
            service_periods.push_back(service->get_period());
            next_release_times.push_back(std::chrono::milliseconds(service->get_period()));
        }

        while (_seq_running)
        {
            auto now = Clock::now();
```

```
        auto elapsed_ms =
std::chrono::duration_cast<std::chrono::milliseconds>(now - start_time);

        for (size_t i = 0; i < _services.size(); ++i)
        {
            if (elapsed_ms >= next_release_times[i])
            {
                _services[i]->release();
// Release the service
                next_release_times[i] +=
std::chrono::milliseconds(service_periods[i]); // Schedule next release
            }
        }

        std::this_thread::sleep_for(std::chrono::milliseconds(1)); // Small
sleep to avoid busy waiting
    }
}
};
```

## File Fibo\_Sequencer.cpp

```
/*

* Fibo_Sequencer.cpp - This is a C++ version of the canonical pthread service
example. It intends

* to abstract the service management functionality and sequencing for ease

* of use. Much of the code is left to be implemented by the student

*

*/
```

```
* Original Authour-Steve Rizer 3/16/2025

* Modified: Nalin Saxena/Bella Wolf

*/

#include <stdint>

#include <stdio>

#include "Fibo_Sequencer.hpp"

#include <csignal> // For signal handling

#define FIB_LIMIT_FOR_32_BIT 47

#define FIB_TEST_CYCLES 1800

#define NSEC_PER_SEC (1000000000)

#define PRE_TEST 200

#define TEST_DURATION_MS 1000

uint32_t seqIterations = FIB_LIMIT_FOR_32_BIT;

uint32_t idx = 0, jdx = 1;

uint32_t fib = 0, fib0 = 0, fib1 = 1;

uint32_t fib10Cnt = 0, fib20Cnt = 0;

volatile int ten_ms_count = 0;

volatile int twenty_ms_count = 0;

volatile sig_atomic_t stop_flag = 0;
```

```
void signalHandler(int signum)
{
    if (signum == SIGINT)
    {
        syslog(LOG_INFO, "CAUGHT SIGNAL!");

        stop_flag = 1;
    }
}

/*following macro taken as is from vx works example
* */

#define FIB_TEST(seqCnt, iterCnt)
for (idx = 0; idx < iterCnt; idx++)
{
    fib = fib0 + fib1;

    while (jdx < seqCnt)
    {
        fib0 = fib1;

        fib1 = fib;

        fib = fib0 + fib1;

        jdx++;
    }
}
```

```
    }  
}  
  
/*below code taken as is from : Sam siewert posix clcock example  
  
* */  
  
int delta_t(struct timespec *stop, struct timespec *start, struct timespec  
*delta_t)  
{  
  
    int dt_sec = stop->tv_sec - start->tv_sec;  
  
    int dt_nsec = stop->tv_nsec - start->tv_nsec;  
  
    if (dt_sec >= 0)  
    {  
  
        if (dt_nsec >= 0)  
        {  
  
            delta_t->tv_sec = dt_sec;  
  
            delta_t->tv_nsec = dt_nsec;  
        }  
  
        else  
        {  
  
            delta_t->tv_sec = dt_sec - 1;  
  
            delta_t->tv_nsec = NSEC_PER_SEC + dt_nsec;  
        }  
    }  
}
```



```
    else

    {

        if (dt_nsec >= 0)

        {

            delta_t->tv_sec = dt_sec;

            delta_t->tv_nsec = dt_nsec;

        }

        else

        {

            delta_t->tv_sec = dt_sec - 1;

            delta_t->tv_nsec = NSEC_PER_SEC + dt_nsec;

        }

    }

    return (1);
}

/*Fib 10 service to generate 10 ms of workload

*

* Parameters:

*         None

*

* Returns:
```

```
*          None

*

*/

void fib10()

{

    // Warm-up computation

    FIB_TEST(seqIterations, FIB_TEST_CYCLES);

    // Simulate workload

    for (int i = 0; i < ten_ms_count; i++)

    {

        FIB_TEST(seqIterations, FIB_TEST_CYCLES);

    }

}

/*Fib 20 service to generate 20 ms of workload

*

* Parameters:

*

*          None

*

* Returns:

*

*          None
```

```
*  
  
*/  
  
void fib20()  
{  
  
    // Warm-up computation  
  
    FIB_TEST(seqIterations, FIB_TEST_CYCLES);  
  
    // Simulate workload  
  
    for (int i = 0; i < twenty_ms_count; i++)  
  
    {  
  
        FIB_TEST(seqIterations, FIB_TEST_CYCLES);  
    }  
}  
  
/*A function to get iterations required to run  
  
*for delay values  
  
* Parameters:  
  
*         None  
  
*  
  
* Returns:  
  
*         None  
  
*
```

```
*/  
  
void run_iteration_test()  
{  
  
    int accum_iter_fib10 = 0;  
  
    int accum_iter_fib20 = 0;  
  
    // Warm-up computation  
  
    FIB_TEST(seqIterations, FIB_TEST_CYCLES);  
  
    for (int i = 0; i < PRE_TEST; i++)  
    {  
  
        struct timespec start_time, finish_time, thread_dt;  
  
        clock_gettime(CLOCK_REALTIME, &start_time); // Get start time  
  
        FIB_TEST(seqIterations, FIB_TEST_CYCLES);  
  
        clock_gettime(CLOCK_REALTIME, &finish_time); // Get end time  
  
        delta_t(&finish_time, &start_time, &thread_dt);  
  
        double run_time_calc = (thread_dt.tv_sec * 1000.0) + (thread_dt.tv_nsec /  
1000000.0);  
  
        int req_iterations_fib10 = (int)(10 / run_time_calc);  
  
        int req_iterations_fib20 = (int)(18.5 / run_time_calc); // keeping this  
as 20 overshoots the service2 deadline  
  
        accum_iter_fib10 += req_iterations_fib10;  
    }  
}
```

```
        accum_iter_fib20 += req_iterations_fib20;
    }

    ten_ms_count = (int)accum_iter_fib10 / PRE_TEST;

    twenty_ms_count = (int)accum_iter_fib20 / PRE_TEST;

    syslog(LOG_INFO, "RUNNING fib iteration test! %d %d", ten_ms_count,
twenty_ms_count);
}

int main()
{

    openlog("LOG_MSG", LOG_PID | LOG_PERROR, LOG_USER);

    run_iteration_test();

    std::signal(SIGINT, signalHandler);

    // Example use of the sequencer/service classes:

    Sequencer sequencer{};

    sequencer.addService(fib10, 0, 98, 20, 1);

    sequencer.addService(fib20, 0, 97, 50, 2);

    usleep(10000); // a sleep for the initialization to complete

    sequencer.startServices();

    while (!stop_flag)

    {
```

```
        std::this_thread::sleep_for(std::chrono::milliseconds(100));  
    }  
  
    sequencer.stopServices();  
  
    syslog(LOG_INFO, "Services stopped, exiting...");  
  
    closelog();  
}
```

## File Fibo\_Sequencer.hpp

```
/*  
 * Fibo_Sequencer.hpp - Contains implementation for service and scheduler classes  
 *  
 * Original Authour-Steve Rizor 3/16/2025  
 * Modified: Nalin Saxena/Bella Wolf  
 *  
 * Refrence and acknowledgement- For developing the code for this excercise we  
utilized chatgpt for getting the correct  
 * syntax. Code related to logStats and timing was written with chatgpt  
 *  
 * The suggestion for using unique_ptr for storing the services was provided by  
the insructor -Steve Rizor  
 * The idea of using a run time test run_iteration_test and precomputing the  
value of iterations was also suggested by the  
 * instructor during a discussion in office hours.  
*/
```

```
*/

#pragma once

#include <stdint>
#include <functional>
#include <thread>
#include <iostream>
#include <vector>
#include <syslog.h>
#include <mutex>
#include <csignal>

double global_start_time;
double getCurrentTimeInMs(void)
{
    struct timespec currentTime = {0, 0};

    // Get the current time from the CLOCK_MONOTONIC clock
    clock_gettime(CLOCK_MONOTONIC, &currentTime);

    // Convert the time to milliseconds (seconds * 1000 + nanoseconds /
1,000,000)
    return ((currentTime.tv_sec) * 1000.0) + ((currentTime.tv_nsec) / 1000000.0);
}

// The service class contains the service function and service parameters
// (priority, affinity, etc). It spawns a thread to run the service, configures
// the thread as required, and executes the service whenever it gets released.
class Service
{
public:
    template <typename T>
    //Constructor for Service class uses an initialization list for setting up the
Service object
    Service(T &&doService, uint8_t affinity, uint8_t priority, uint32_t period,
int service_identifider) : _doService(doService),
```

```
_affinity(affinity),

_priority(priority),

_period(period),

_service_identifier(service_identifier),

_semaphore(0)//start in blocked state

{
    // todo: store service configuration values
    // todo: initialize release semaphore

    // Start the service thread, which will begin running the given function
immediately
    syslog(LOG_INFO, "constructor called for Service %d\n",
_service_identifier);
    syslog(LOG_INFO, "affinity %d \n", static_cast<int>(_affinity));
    syslog(LOG_INFO, "priority  %d \n", static_cast<int>(_priority));
    syslog(LOG_INFO, "period %d  \n", static_cast<int>(_period));
    _service = std::jthread(&Service::_provideService, this);
}

void stop()
{
    // todo: change state to "not running" using an atomic variable
    // (heads up: what if the service is waiting on the semaphore when this
happens?)
    _running = false;
    _semaphore.release();
}

void release()
{
    // todo: release the service using the semaphore
    _semaphore.release();
}
```



```
}  
//a getter to return private data  
int get_period()  
{  
    return _period;  
}  
//a getter to return private data  
int get_id()  
{  
    return _service_identifrier;  
}  
//prints logs for each service after stop is called from sequencer  
void logStats()  
{  
    auto avg_time = _total_time / _exec_count;  
    auto avg_jitter = _total_exec_jitter / _exec_count; // Calculate the  
average jitter  
    syslog(LOG_INFO, "Service Execution Stats: For %d", _service_identifrier);  
    syslog(LOG_INFO, "_exec_count : %d", _exec_count);  
    syslog(LOG_INFO, "  Min Execution Time: %ld ms", _min_time.count());  
    syslog(LOG_INFO, "  Max Execution Time: %ld ms", _max_time.count());  
    syslog(LOG_INFO, "  Avg Execution Time: %ld ms", avg_time.count());  
    syslog(LOG_INFO, "  Avg Jitter: %ld ms", avg_jitter.count()); // Log the  
average jitter  
}  
  
private:  
    std::function<void(void)> _doService;  
    std::jthread _service;  
    int _affinity;//core id the thread should be bound to  
    int _priority;  
    int _period;  
    int _service_identifrier;//a unique identifier for each service  
    std::binary_semaphore _semaphore;  
    std::atomic<bool> _running{true};  
    //logging data members  
    std::chrono::milliseconds _min_time{std::numeric_limits<long>::max()};  
    std::chrono::milliseconds _max_time{0};
```

```
std::chrono::milliseconds _total_time{0};
unsigned int _exec_count = 0;
std::chrono::milliseconds _total_exec_jitter{0};

//setting up the thread priority,affinity and scheduling policy to SCHED_FIFO
void _initializeService()
{
    // todo: set affinity, priority, sched policy
    // (heads up: the thread is already running and we're in its context
right now)
    syslog(LOG_INFO, "Initializing service...");
    pthread_t threadID = pthread_self(); // Get current thread ID
    cpu_set_t cpuset;
    struct sched_param param;
    CPU_ZERO(&cpuset);
    CPU_SET(_affinity, &cpuset);
    if (pthread_setaffinity_np(threadID, sizeof(cpu_set_t), &cpuset) != 0)
    {
        syslog(LOG_ERR, "Failed to set CPU affinity for thread.");
    }
    int policy = SCHED_FIFO; // sudo access
    param.sched_priority = _priority;

    if (pthread_setschedparam(threadID, policy, &param) != 0)
    {
        syslog(LOG_ERR, "Failed to set scheduling parameters.");
    }

    syslog(LOG_INFO, "Service initialized with affinity %d and priority %d",
_affinity, _priority);
}
void _provideService()
{
    _initializeService();
    while (_running)
    {
        _semaphore.acquire();
        if (!_running)
```

```
        break;

        auto start_time = std::chrono::steady_clock::now();

        auto start_time_ms = getCurrentTimeInMs() - global_start_time;
        int cpu = sched_getcpu();
        syslog(LOG_INFO, "Service %d started execution at %.2f ms core %d",
        _service_identifier, start_time_ms, cpu);
        //call the service function
        _doService();

        auto end_time = std::chrono::steady_clock::now();
        auto exec_time =
std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);

        auto end_time_ms = getCurrentTimeInMs() - global_start_time;
        syslog(LOG_INFO, "Service %d finished execution at %.2f ms core %d",
        _service_identifier, end_time_ms, cpu);

        std::chrono::milliseconds jitter(0);
        if (_service_identifier == 1)
        {
            jitter = exec_time - std::chrono::milliseconds(10);
        }
        else if (_service_identifier == 2)
        {
            jitter = exec_time - std::chrono::milliseconds(30);
        }
        // execution stats
        _exec_count++;
        _total_time += exec_time;
        _total_exec_jitter += abs(jitter);
        _min_time = std::min(_min_time, exec_time);
        _max_time = std::max(_max_time, exec_time);
    }
}
};
```

```
// The sequencer class contains the services set and manages
// starting/stopping the services. While the services are running,
// the sequencer releases each service at the requisite timepoint.
class Sequencer
{
public:
    template <typename... Args>
    void addService(Args &&...args)
    {
        // Add the new service to the services list,
        // constructing it in-place with the given args

        _services.emplace_back(std::make_unique<Service>(std::forward<Args>(args)...));
    }

    void startServices()
    {
        _seq_running = true;
        //manually trigger threads
        _trigger_thread = std::jthread(&Sequencer::manualServiceTriggering,
this);

        // Set thread scheduling policy and priority (SCHED_FIFO with priority 99
scheduler thread should run at max prio)
        setThreadRealtimePriority(_trigger_thread);
    }

    void stopServices()
    {
        _seq_running = false;
        for (auto &service : _services)
        {
            service->stop();
            service->logStats(); //display stats
        }
    }

private:
```

```
void manualServiceTriggering()
{
    if (!_seq_running)
        return;

    // manually release services
    global_start_time = getCurrentTimeInMs(); //at this instant be record our
initial time
    while (_seq_running)
    {
        syslog(LOG_INFO, "CI INSTANT! %f \n", getCurrentTimeInMs() -
global_start_time);
        auto &service = _services;
        service[0]->release();
        service[1]->release();

        usleep(slep20Ms);
        service[0]->release();
        syslog(LOG_INFO, "sequencer! %f \n", getCurrentTimeInMs() -
global_start_time);

        usleep(slep20Ms);
        service[0]->release();
        syslog(LOG_INFO, "sequencer! %f \n", getCurrentTimeInMs() -
global_start_time);

        usleep(slep10MS);
        service[1]->release();
        syslog(LOG_INFO, "sequencer! %f \n", getCurrentTimeInMs() -
global_start_time);

        usleep(slep10MS);
        service[0]->release();
        syslog(LOG_INFO, "sequencer! %f \n", getCurrentTimeInMs() -
global_start_time);

        usleep(slep20Ms);
        service[0]->release();
    }
}
```

```
        syslog(LOG_INFO, "sequencer! %f \n", getCurrentTimeInMs() -
global_start_time);
        usleep(slep20Ms);
    }
}
//setup up sequencer as RT thread
void setThreadRealtimePriority(std::jthread &thread)
{
    std::thread::native_handle_type handle = thread.native_handle();

    // Set scheduling policy to FIFO and priority to 99
    struct sched_param sched_param;
    sched_param.sched_priority = 99;
    if (pthread_setschedparam(handle, SCHED_FIFO, &sched_param) != 0)
    {
        std::cerr << "Failed to set real-time scheduling policy" <<
std::endl;
    }
    // Set the CPU affinity to core 0
    cpu_set_t cpu_set;
    CPU_ZERO(&cpu_set);
    CPU_SET(0, &cpu_set); // Pin the thread to core 0
    if (pthread_setaffinity_np(handle, sizeof(cpu_set_t), &cpu_set) != 0)
    {
        std::cerr << "Failed to set CPU affinity" << std::endl;
    }
}
std::vector<std::unique_ptr<Service>> _services;
std::atomic<bool> _seq_running{false};

std::jthread _trigger_thread; // manual triggering
const int slep10MS = 10000;    // 10 milliseconds sleep
const int slep20Ms = 20000;    // 20 milliseconds sleep
};
```