313551137 官劉翔

1. **Describe how you implemented the program in detail. (10%)**

   For Makefile:

   1. Basic Structure and Targets:

      ```
      .PHONY: all clean
      ```

      I defined two phony targets .PHONY: all clean to indicate that "all" and "clean" are not actual files

   2. Compiler Settings:

      ```
      CXX = g++
      CXXFLAGS = -Wall -Wextra -Werror -O3 -Wpedantic -pthread
      ```

      The most important flag here is -pthread - I spent quite some time debugging before realizing I needed this for the threads to work!

   3. File Management:

      ```
      SOURCE = sched_demo_313551137.cpp
      OBJS = $(SOURCE:.cpp=.o)
      DEPS = $(SOURCE:.cpp=.d)
      TARGET = sched_demo_313551137
      ```

      In this part, I defined the source file with my student ID and set up automatic object file generation (OBJS). Besides, I created dependency files (DEPS) for tracking header dependencies. In the end, I specified the target executable name.

   4. Dependency Handling

      ```
      -include $(DEPS)
      OUTPUT_OPTION = -MMD -MP -o $@
      ```

   5. Build Rules

      ```
      $(TARGET): $(OBJS)
      	$(CXX) $^ -o $@ $(CXXFLAGS)
      ```

      Created a rule to build the final executable. In details, the symbols $^: All prerequisites , $@: Target name.

   6. For cleaning up

      ```
      clean:
      	@rm -f $(TARGET) $(OBJS) $(DEPS)
      ```

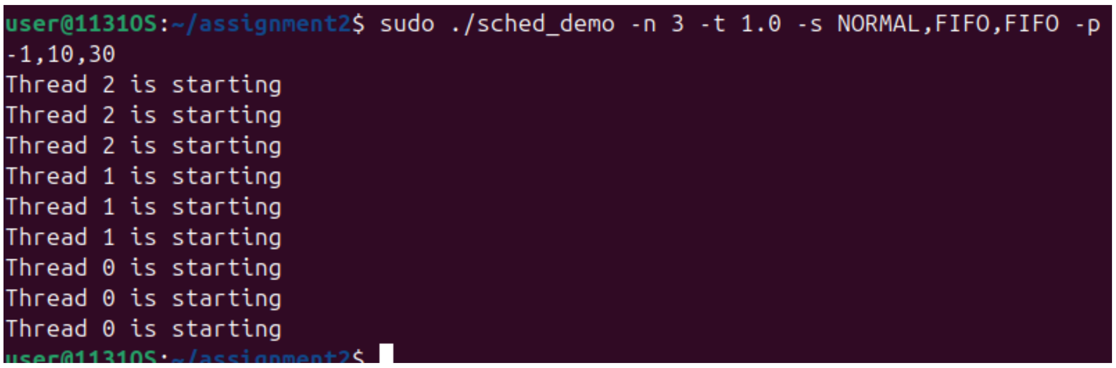
      I added a clean rule to remove all generated files. Besides, I find @

symbol is "quiet mode"! It hides the output of the command itself and only displays the results of the command execution.

For main.cpp:

I implemented this program using C++ to demonstrate different thread scheduling policies.
In the main function, I used getopt() to parse command line arguments, which allowed users to specify the number of threads (-n), time to wait (-t), scheduling policies (-s), and priorities (-p). For the scheduling policies and priorities, I split the comma-separated input strings into vectors for easier processing.
The core part of my implementation involves creating threads with different scheduling attributes. For each thread, I set up its attributes using pthread_attr_setinheritsched() with PTHREAD_EXPLICIT_SCHED, and then set different scheduling policies (either SCHED_FIFO or SCHED_OTHER) by if else based on the input. For FIFO threads, I also set their priorities using pthread_attr_setschedparam().
To ensure all threads run on the same CPU core, I set CPU affinity for both the main thread and worker threads using pthread_setaffinity_np(). I also used a barrier to synchronize all threads at the start of execution.
Change kernel.sched_rt_runtime_us
In the beginning, I observed the FIFO threads didn't actually execute in front of Normal thread, so I ask in forum and change the kernel.sched_rt_runtime_us to 1000000 to make sure that the result is suit on our goal.

2. **Describe the results of sudo ./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30 and what causes that. (10%)**

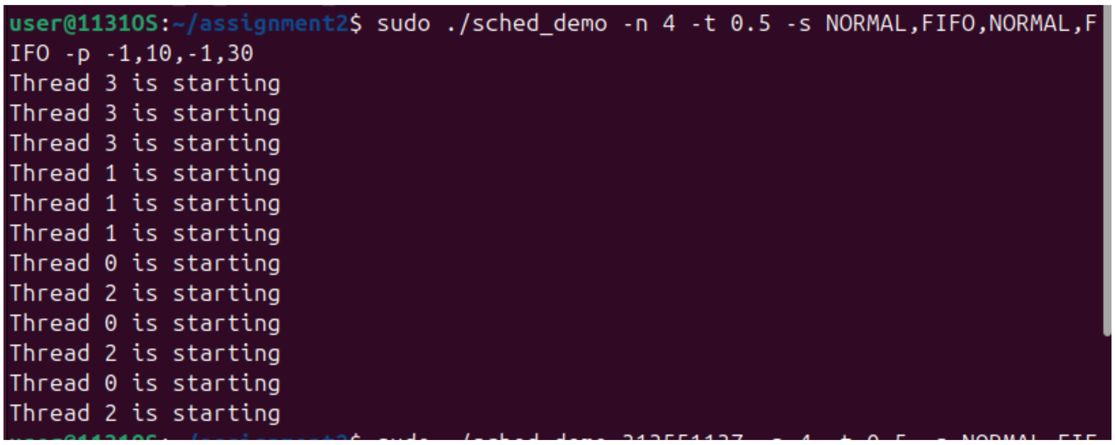


```
user@11310S:~/assignment2$ sudo ./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p
-1,10,30
Thread 2 is starting
Thread 2 is starting
Thread 2 is starting
Thread 1 is starting
Thread 1 is starting
Thread 1 is starting
Thread 0 is starting
Thread 0 is starting
Thread 0 is starting
user@11310S:~/assignment2$
```

When running the program with these parameters, I observed that Thread 2 (FIFO with priority 30) would complete all its iterations first, followed by Thread 1 (FIFO with priority 10), and finally Thread 0 (NORMAL) would execute last.

This happens because SCHED_FIFO is a real-time scheduling policy that always takes precedence over SCHED_NORMAL. Among the FIFO threads, the one with higher priority (30) runs first and cannot be preempted by the lower priority FIFO thread (10). The NORMAL thread only gets to run when both FIFO threads have completed their execution because SCHED_NORMAL uses the CFS (Completely Fair Scheduler) which has lower priority than real-time schedulers.

3. **Describe the results of sudo ./sched_demo -n 4 -t 0.5 -s NORMAL,FIFO,NORMAL,FIFO -p -1,10,-1,30, and what causes that. (10%)**



```
user@11310S:~/assignment2$ sudo ./sched_demo -n 4 -t 0.5 -s NORMAL,FIFO,NORMAL,F
IFO -p -1,10,-1,30
Thread 3 is starting
Thread 3 is starting
Thread 3 is starting
Thread 1 is starting
Thread 1 is starting
Thread 1 is starting
Thread 0 is starting
Thread 2 is starting
Thread 0 is starting
Thread 2 is starting
Thread 0 is starting
Thread 2 is starting
```

When running with these parameters, I observed that Thread 3 (FIFO with priority 30) would execute all its iterations first, followed by Thread 1 (FIFO with priority 10). After both FIFO threads completed, Thread 2 and Thread 0 (both NORMAL) would execute in an alternating pattern due to the CFS scheduler.

As similar as question 2, This behavior occurs because SCHED_FIFO threads always have higher priority than SCHED_NORMAL threads. Between the two FIFO threads, Thread 3 runs first because it has higher priority (30) than Thread 1 (priority 10). For the two NORMAL threads (Thread 2 and 0), they share CPU time fairly using the CFS scheduler, which tries to give each thread an equal amount of CPU time.

4. **Describe how did you implement n-second-busy-waiting? (10%)**

   I implemented the busy waiting function using C++'s chrono library to achieve accurate timing without using sleep functions. Here's how I did it:

```
void busy_wait(double seconds) {
    auto start = chrono::high_resolution_clock::now();
    while (true) {
        auto current = chrono::high_resolution_clock::now();
        chrono::duration<double> elapsed = current - start;
        if (elapsed.count() >= seconds) break;
    }
}
```

   Instead of using sleep() which would put the thread into sleep state and affect scheduling behavior, I used a busy loop that continuously checks the elapsed time. Besides, I used high_resolution_clock for better timing accuracy and the duration class to handle the time comparison.

5. **What does the kernel.sched_rt_runtime_us effect? If this setting is changed, what will happen?(10%)**

   The kernel.sched_rt_runtime_us parameter controls how much CPU time is allocated to real-time tasks (like SCHED_FIFO) in each period. The default value is usually 950000 microseconds out of each 1000000-microsecond period, meaning real-time tasks can use up to 95% of CPU time.

   If increased (e.g., to 990000), real-time threads get more CPU time, which means FIFO threads can run longer before letting NORMAL threads execute

   If decreased (e.g., to 800000), real-time threads get less CPU time, giving more opportunities for NORMAL threads to run

   Setting it too high (near 1000000) could make the system unresponsive because system tasks won't get enough CPU time. Thus, it may cause starve issue.