OS HW2 report

Student ID：314551123 / Name：蘇秭郁

- Answer the Q&As

1. Describe how you implemented the program in detail.

   (1) First, **getopt()** is utilized to parse the input arguments, and **strdup()** is employed to back up the parameter strings; this prevents **strtok_r()** from directly modifying the original content. Subsequently, **sched_setaffinity** is used to bind the process to a single CPU, which is essential for observing preemption and Round-Robin (RR) scheduling behaviors.

   (2) Next, the corresponding threads are created based on the input parameters:

      i. When configuring thread_attr, **pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED)** is invoked to ensure that the created threads do not inherit attributes from the main thread, but instead use the specifically defined attributes.

      ii. For the priority of SCHED_FIFO threads, the priority levels are configured via the **sched_param** structure.

      iii. To retrieve policies and priorities, **strtok_r()** is used instead of the standard strtok() to avoid thread-local storage conflicts and potential segmentation faults.

   (3) Furthermore, to ensure that all threads begin competing for the CPU at the same time, a **pthread_barrier_t** is implemented. After each thread is created, it calls **pthread_barrier_wait**, remaining blocked until all threads have been successfully initialized, at which point they begin execution simultaneously.

   (4) Regarding time measurement, since the busy-wait period must exclude the time the thread is preempted, **clock_gettime()** is used in conjunction with **CLOCK_THREAD_CPUTIME_ID**. This specific timer ensures that counting only occurs when the thread is actively utilizing the CPU.

Within the program, the CPU occupancy time is calculated through a busy-wait loop.

(5) Finally, **pthread_join()** is called within main() to wait for all worker threads to complete their execution. Once finished, the barrier is destroyed, and the allocated memory is released.

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

```
~ # ./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30
Thread 2 is running
Thread 2 is running
Thread 2 is running
Thread 1 is running
Thread 1 is running
Thread 1 is running
Thread 0 is running
Thread 0 is running
Thread 0 is running
```

Since Thread 1 and Thread 2 are assigned the FIFO scheduling policy while Thread 0 is set to NORMAL, Thread 1 and Thread 2 will be executed with higher priority. Furthermore, because Thread 2 has a higher priority level than Thread 1, Thread 2 will execute before Thread 1. Consequently, the final execution sequence will be Thread 2 → Thread 1 → Thread 0.

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

```
~ # ./sched_demo -n 4 -t 0.5 -s NORMAL,FIFO,NORMAL,FIFO -p -1,10,-1,30
Thread 3 is running
Thread 3 is running
Thread 3 is running
Thread 1 is running
Thread 1 is running
Thread 1 is running
Thread 2 is running
Thread 0 is running
Thread 2 is running
Thread 0 is running
Thread 2 is running
Thread 0 is running
```

Since Thread 1 and Thread 3 are assigned the FIFO scheduling policy, they will execute before Thread 0 and Thread 2. Within the FIFO group, Thread 3 has a higher priority than Thread 1, so Thread 3 will execute first. The remaining Thread 0 and Thread 2 engage in time-sharing execution, where the Completely Fair Scheduler (CFS) manages their resource allocation in an

interleaved fashion.

4. Describe how did you implement n-second-busy-waiting?

   The n-second busy waiting is implemented using **clock_gettime()** with the **CLOCK_THREAD_CPUTIME_ID** clock source. This timer measures the actual CPU time consumed by the thread, effectively excluding any duration where the thread is preempted. By polling the timer within a while loop without yielding the CPU, the program ensures the thread remains busy for exactly the specified duration.

5. What does the kernel.sched_rt_runtime_us effect? If this setting is changed (eg. 500000, 950000, 1000000), what will happen?

   The **kernel.sched_rt_runtime_us** parameter is a safety mechanism in Linux designed to prevent real-time threads from completely starving the CPU. It specifies the amount of time real-time (RT) threads are allowed to run within a specific period (defined by sched_rt_period_us, which defaults to 1 second).

   • Effects of diffenet settings:

   500,000 (0.5s): RT threads are capped at 50% CPU utilization per second. The remaining 50% is strictly reserved for non-RT tasks (e.g., normal processes, shell commands).

   950,000 (0.95s, Default): This is the system default. RT threads can consume up to 95% of the CPU. The 5% margin ensures that a system administrator can still access a shell to kill a runaway or bugged RT process.

   1,000,000 (1s): RT threads can occupy 100% of the CPU. This is dangerous because if a SCHED_FIFO process enters an infinite loop, the system will become completely unresponsive, preventing any keyboard or mouse input.