Lab2 Report
313553054 陳冠霖

(1) Describe How You Implemented the Program in Detail.(10%)

The program `sched_demo` was implemented to create multiple threads with specified scheduling policies and priorities, as well as demonstrate their execution. The implementation can be broken into several steps:

1. Parsing Command-Line Arguments

The program accepts command-line arguments to specify the number of threads (`-n`), the busy-wait time (`-t`), scheduling policies (`-s`), and priorities (`-p`). These arguments are processed using `getopt()`.

Options:
-n <num_threads>`: Number of threads to create.
-t <time_wait>`: Time (in seconds) for the busy wait.
-s <schedules>`: Comma-separated list of scheduling policies (e.g., `NORMAL,FIFO`).
-p <priorities>`: Comma-separated list of priorities (e.g., `-1,10`).

The parsed policies and priorities are stored as arrays (`policy_tokens` and `priority_tokens`) using `strtok()` for further processing.

2. Thread Information Structure

A structure `thread_info` was defined to hold information for each thread:
- `id`: Unique thread identifier.
- `policy`: Scheduling policy (`SCHED_OTHER` for NORMAL, `SCHED_FIFO` for FIFO).
- `priority`: Priority for real-time threads (`-1` for NORMAL threads).
- `time_wait`: The duration of busy waiting.

3. Setting Up Threads

The threads are created using the following steps:

(1) Barrier Initialization: A barrier (`pthread_barrier_t`) ensures that all threads start execution simultaneously after being initialized. This prevents some threads from starting early.

(2) Thread Attributes:
   Scheduling Policy and Priority:
   - Threads with the "NORMAL" policy use `SCHED_OTHER` with priority `0`.
   - Threads with the "FIFO" policy use `SCHED_FIFO` with the priority specified in

the command-line arguments.

CPU Affinity:
- All threads are pinned to CPU 0 using `CPU_SET` and `pthread_attr_setaffinity_np()` to control thread execution on a single core.

(3) Thread Creation: Each thread is created with the configured attributes (`pthread_attr_t`) and assigned the `thread_func()` function for execution.

4. Thread Execution

The function `thread_func()` is executed by each thread:

(1) Barrier Synchronization: Each thread waits at the barrier (`pthread_barrier_wait`) until all threads are ready.
(2) Task Execution:
- The thread prints a message (`Thread <id> is starting`) at the beginning of each iteration.
- It then performs a busy wait for the specified time (`time_wait`) using the `busy_wait()` function.

(3) Busy Waiting Implementation:
- The `busy_wait()` function measures CPU time (not wall-clock time) using `clock_gettime(CLOCK_THREAD_CPUTIME_ID)`.
- A loop checks elapsed time to ensure the thread only counts active processing time and excludes time spent preempted by the system.

Each thread repeats this process three times before exiting.

5. Memory Management

The program dynamically allocates memory for the thread array (`threads`) and thread information array (`tinfo`) based on the number of threads.
- Memory is freed after all threads complete their execution.

6. Cleaning Up Resources

- The barrier is destroyed using `pthread_barrier_destroy()` after all threads finish.
- Allocated memory for `threads` and `tinfo` is released to prevent memory leaks.

(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%)

```
brian-vm@brian-vm-VirtualBox:~/Documents/lab2/313553054$ 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 0 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
```

Real-Time Thread Execution

- Threads 1 and 2 are SCHED_FIFO with priorities 10 and 30, respectively.
- Under normal circumstances, these threads would monopolize the CPU, with Thread 2 (priority 30) executing first, followed by Thread 1 (priority 10).

Why Does Thread 0 (NORMAL) Execute?

- Real-time threads (Thread 1 and Thread 2) consume their allotted 95% of the CPU time (950 ms out of every 1 second for example).
- When the real-time runtime limit is reached, the kernel throttles real-time threads and schedules non-real-time threads (SCHED_OTHER), allowing Thread 0 to execute.

Interleaving of Threads

- The kernel enforces fairness by ensuring non-real-time threads are not completely starved.
- As a result, Thread 0 gets CPU time during the 5% reserved for non-real-time tasks or whenever real-time threads are throttled.

Result explanation:

- It is affected by kernel.sched_rt_runtime_us(the above concept)
- Thread 2 (FIFO, Priority 30) starts first because it has the highest real-time priority.
- After running for part of its real-time allocation, the kernel throttles it, allowing Thread 0 (NORMAL) to execute briefly.

- Thread 2 resumes and completes its busy-wait for the current iteration.
- Thread 1 (FIFO, Priority 10) then starts after Thread 2 finishes and exhibits similar behavior:
    - It is throttled when it reaches the real-time runtime limit, allowing Thread 0 to execute intermittently.

(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%)

```
brian-vm@brian-vm-VirtualBox:~/Documents/lab2/313553054$ sudo ./sched_demo -n 4
-t 0.5 -s NORMAL,FIFO,NORMAL,FIFO -p -1,10,-1,30
Thread 3 is starting
Thread 3 is starting
Thread 3 is starting
Thread 1 is starting
Thread 0 is starting
Thread 2 is starting
Thread 1 is starting
Thread 1 is starting
Thread 2 is starting
Thread 0 is starting
Thread 0 is starting
Thread 2 is starting
```

Results explanation:

-Similar as the previous question(affected by kernel.sched_rt_runtime_us)

1. Thread 3 Starts:
    - As the highest-priority real-time thread (`FIFO, Priority 30`), Thread 3 starts executing and busy-waits for 0.5 seconds.
    - It executes repeatedly until it reaches the real-time runtime limit defined by `kernel.sched_rt_runtime_us`.
2. Thread 1 Executes:
    - After Thread 3 is throttled, the next real-time thread (Thread 1, `FIFO, Priority 10`) starts running. It busy-waits for its allocated CPU time.
3. Threads 0 and 2 Run:
    - When both real-time threads are throttled, the kernel schedules the `NORMAL` threads (Threads 0 and 2) using the CFS.
    - These threads alternate based on their CPU shares, leading to interleaving in the output.
4. Interleaving of Threads:
    - The alternating execution of `NORMAL` threads with real-time threads occurs due to the kernel's enforcement of fairness through `sched_rt_runtime_us`.

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

```
void busy_wait(double seconds) {
    struct timespec start, current;
    clock_gettime(CLOCK_THREAD_CPUTIME_ID, &start);  // Get start time based on CPU time for the thread

    while (1) {
        clock_gettime(CLOCK_THREAD_CPUTIME_ID, &current);  // Get the current CPU time for the thread

        double elapsed = (current.tv_sec - start.tv_sec) +
                         (current.tv_nsec - start.tv_nsec) / 1e9;  // Convert nanoseconds to seconds

        if (elapsed >= seconds) break;
    }
}
```

Key Features

1. Thread-Specific Time(instead of world time):
    - The function uses `CLOCK_THREAD_CPUTIME_ID`, ensuring that only the CPU time actively used by the thread is measured.
    - This is crucial in multi-threaded or multi-tasking environments where the thread might be preempted or placed in a waiting state.
2. High-Precision Measurement:
    - Using `struct timespec`, the time is measured in both seconds (`tv_sec`) and nanoseconds (`tv_nsec`), allowing for precise control of the wait duration.
3. Simple and Efficient:
    - The busy-wait relies on repeated system calls to `clock_gettime`, which is lightweight for short durations.

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

Effect of `kernel.sched_rt_runtime_us`

1. Purpose:
    - Controls the maximum CPU time (in microseconds) a real-time (RT) task group can use within a given period (`kernel.sched_rt_period_us`).
    - Ensures fairness and prevents RT tasks from monopolizing the CPU.
2. Key Impact:
    - When RT tasks exhaust their allocated runtime within the period, they are throttled (paused) until the next period starts.
    - This allows non-real-time tasks (e.g., NORMAL tasks) to get CPU time.
3. Changing the Setting:
    - Increase:

        - Extends the runtime for RT tasks, allowing them to occupy the CPU longer.
        - Risk: Non-RT tasks may face starvation if the runtime becomes too high.
    - Decrease:
        - Reduces the runtime for RT tasks, limiting their CPU usage.
        - Benefit: Ensures more CPU time is available for non-RT tasks but might degrade RT task performance.
4. Effect on Mixed Scheduling:
    - In scenarios with both RT and NORMAL tasks, reducing `sched_rt_runtime_us` ensures NORMAL tasks have more consistent access to the CPU.
    - Increasing it gives RT tasks higher priority but risks starving NORMAL tasks.

(6) the result of test case:

```
brian-vm@brian-vm-VirtualBox:~/Documents/lab2/313553054$ sudo ./sched_test.sh ./
sched_demo ./sched_demo_313553054
Running testcase 0 : ./sched_demo -n 1 -t 0.5 -s NORMAL -p -1
Result: Success!
Running testcase 1 : ./sched_demo -n 2 -t 0.5 -s FIFO,FIFO -p 10,20
Result: Success!
Running testcase 2 : ./sched_demo -n 3 -t 1.0 -s NORMAL,FIFO,FIFO -p -1,10,30
Result: Success!
```