Real-Time Thread Programming on Raspberry Pi with PREEMPT_RT

YU CHUN, LIN 71135843

1. Project Description

- Goal: To get familiar with real-time programming on Raspberry Pi by:

1. Patching the Linux kernel to support PREEMPT_RT
2. Practicing with POSIX real-time extension APIs

- Expected results:

1. A patched Raspberry Pi OS kernel with PREEMPT_RT support
2. Sample programs demonstrating real-time thread programming capabilities

2. Experimental Setup

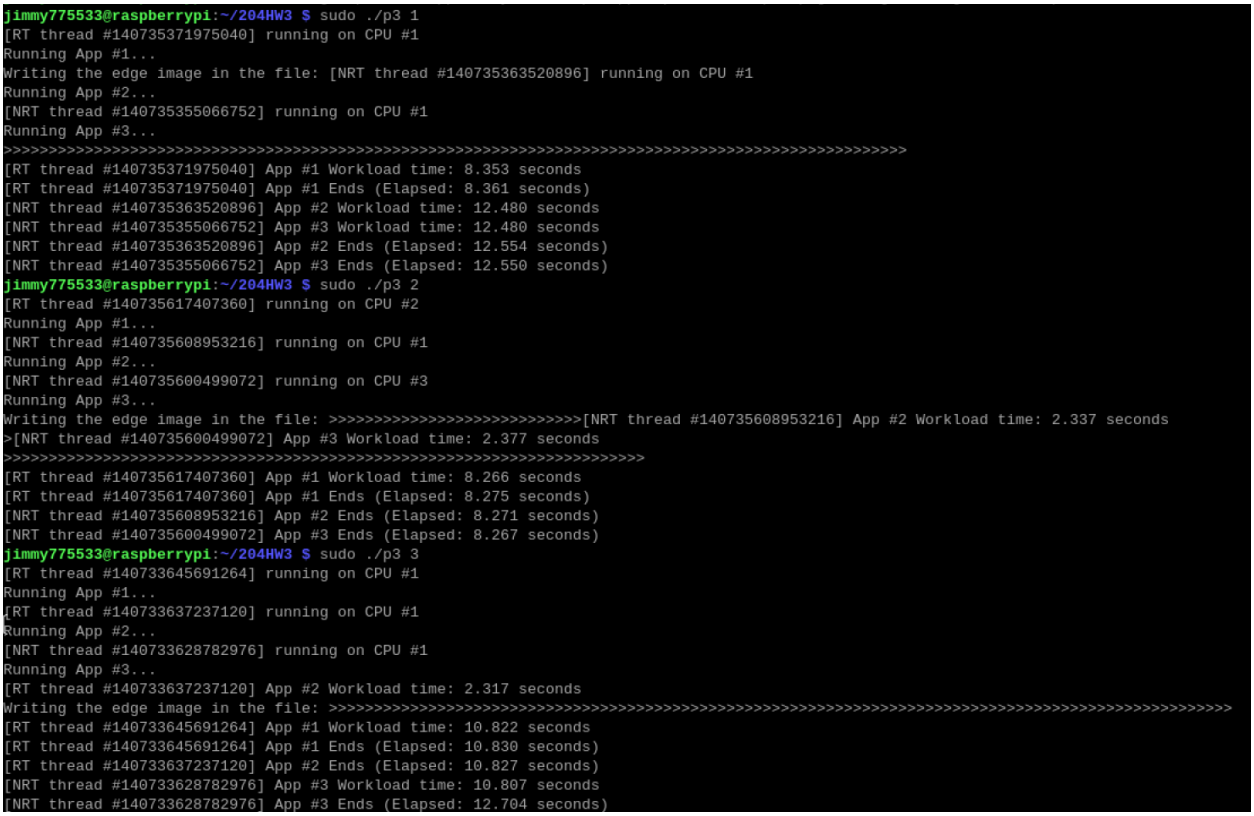
Methodology:

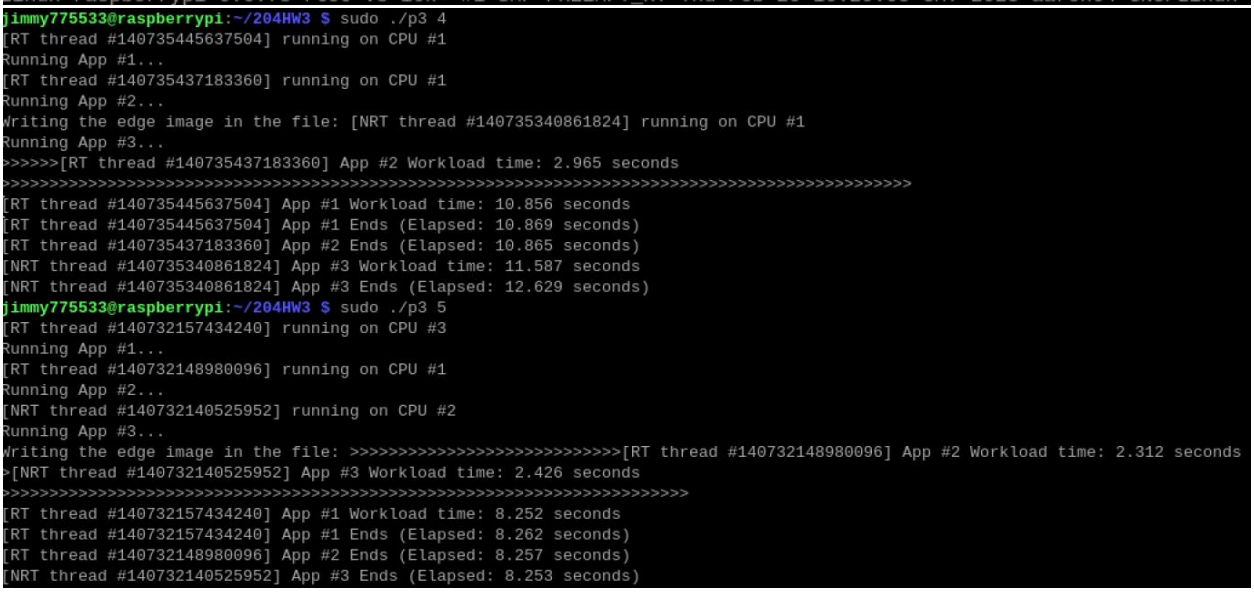
1. Checking current kernel build configuration
2. Patching Raspberry Pi OS for PREEMPT_RT
3. Building and installing the patched kernel
4. Developing sample RT thread programs using provided code template
5. Testing and profiling the sample programs under different scenarios

Key commands used:

- `uname -a` to check kernel build
- `git clone` to download kernel source
- `patch` to apply RT patch
- `make menuconfig` to configure kernel
- `make` to build kernel
- `sudo make modules_install` and related commands to install kernel

3. Results





The code implements a framework for running real-time (RT) and non-real-time (NRT) threads with different scheduling policies and CPU affinities. The main function switches between different test cases based on the `exp_id` command line argument.

Test Case 1 (`exp_id = 1`):

- 1 RT thread (App #1 running CannyP3) and 2 NRT threads (App #2, #3 running BusyCal) all pinned to CPU #1
- The RT thread has higher priority (80) and uses SCHED_FIFO policy
- As expected, the RT thread finishes much faster (8.361s) than the NRT threads (around 12.5s) despite sharing the CPU

Test Case 2 (`exp_id = 2`):

- Same thread setup as Case 1 but with CPU affinity disabled (`set_cpu = false`)
- This allows the threads to be scheduled on any available CPU
- The result is all apps finishing in roughly the same time (8.2-8.3s) as they can leverage multiple CPUs to run in parallel

Test Case 3 (`exp_id = 3`):

- 2 RT threads (App #1 running CannyP3, App #2 running BusyCal) with same priority (80) and SCHED_FIFO policy, and 1 NRT thread (App #3 running BusyCal), all pinned to CPU #1
- RT App #2 finishes much faster (2.3s) than RT App #1 (10.8s) because it starts after App #1 and, under SCHED_FIFO, runs uninterrupted until completion
- NRT App #3 has the lowest priority so finishes last around 12.7s

Test Case 4 (`exp_id = 4`):

- Similar to Case 3 but using SCHED_RR policy for the RT threads
- The two RT apps now have similar execution times (10.8-10.9s) as the CPU time is shared more evenly between them via round-robin scheduling
- The NRT App #3 still finishes last (12.6s) as it has the lowest priority

Test Case 5 (`exp_id = 5`):

- Same thread setup as Case 3 (2 RT with SCHED_FIFO, 1 NRT) but with CPU affinity disabled
- This allows each thread to be scheduled on a different CPU, minimizing resource contention
- Consequently, all apps finish around the same time (8.2-8.3s), demonstrating the performance benefits of dedicated CPUs for each thread

The key insights are:
1. RT threads take priority over NRT threads when sharing a CPU, as designed.
2. When multiple RT threads of the same priority share a CPU, SCHED_FIFO allows threads to monopolize the CPU based on start order, while SCHED_RR time slices between threads more fairly.
3. Allowing threads to run on separate CPUs (by disabling CPU affinity) significantly improves overall execution time by leveraging parallelism and minimizing CPU contention.
The results align with expectations based on the scheduling policies and CPU affinity settings in each test case, validating the real-time performance characteristics of the system.

4. Problems and Discussion (6 pts)

One of the main challenges faced during this project was patching the Raspberry Pi OS with the PREEMPT_RT patch. The kernel version plays a crucial role in this process, and mismatches between the kernel version and the patch version can lead to numerous compilation errors. I encountered this issue firsthand and spent a significant amount of time troubleshooting and resolving these errors. It's essential to ensure that the patch version corresponds exactly with the kernel version to avoid such problems.

Another challenge was the lengthy compilation time for the patched kernel. Building the kernel on the Raspberry Pi itself can take hours, and there's a risk of the Pi overheating during this process. To mitigate this, I set up proper cooling with a fan and monitored the CPU temperature throughout the build. However, this still required careful attention and patience.

Regarding the RT thread programming results, they generally aligned with expectations based on the scheduling policies and CPU affinity settings used in each test case. The RT threads consistently took priority over NRT threads when sharing a CPU, and the SCHED_FIFO and SCHED_RR policies behaved as anticipated in terms of allowing preemption and time slicing between same-priority RT threads.

One result that was perhaps even better than expected was the significant performance improvement observed when allowing threads to run on separate CPUs by disabling CPU affinity. The execution times reduced dramatically compared to sharing a single CPU, showcasing the immense benefits of leveraging parallelism across multiple cores.

To further improve the methodology, one approach could be to cross-compile the patched kernel on a more powerful machine instead of building it directly on the Raspberry Pi. This would greatly reduce the compilation time and avoid the risk of overheating. Additionally, using more realistic and diverse workloads in the test applications could provide more comprehensive insights into the real-time performance characteristics under different scenarios.

5. Conclusion (2 pts)

In summary, this project provided hands-on experience with patching the Raspberry Pi OS kernel with PREEMPT_RT and developing real-time thread applications. It highlighted the importance of matching the patch version with the kernel version to avoid compilation issues and the need for proper cooling during the lengthy build process.

The results from the test applications validated the expected real-time behavior, demonstrating prioritization of RT threads over NRT threads, the impact of SCHED_FIFO vs SCHED_RR policies on same-priority RT threads, and the significant performance gains achievable by allowing threads to run on dedicated CPUs.

Overall, the project offered valuable insights into the practical aspects of enabling and leveraging real-time capabilities on the Raspberry Pi platform. It emphasized the importance of careful configuration, patient troubleshooting, and a thorough understanding of scheduling policies and CPU affinity in order to build effective real-time systems.