Real-Time Programming with Linux

Kernel Build and Real-Time Application

1. Project Description

The primary goal of this project is to familiarize the user with real-time programming on a Raspberry Pi by enabling real-time capabilities in the system through kernel patching and utilizing POSIX real-time extension APIs.

This project involves patching the Raspberry Pi OS's Linux kernel to support the PREEMPT_RT patch, which provides preemption support necessary for real-time applications. Additionally, the user will practice creating and managing real-time (RT) and non-real-time (NRT) threads using the POSIX pthread APIs, specifically focusing on thread creation, scheduling policies, CPU affinity, and performance measurement.

At the end of this project, the user is expected to have:

1. A Raspberry Pi running a custom kernel that supports real-time operations through the PREEMPT_RT patch.
2. An understanding of how to manage real-time and non-real-time threads using POSIX real-time APIs.
3. Experience with measuring execution times of real-time applications and comparing performance under different scheduling policies and CPU configurations.

4. A report that details the experiments conducted, profiling CPU usage and thread execution times, with a discussion on the results and their alignment with expectations.

The final result should demonstrate the ability to:

- Successfully patch the Raspberry Pi OS kernel to support PREEMPT_RT.
- Implement and run multiple real-time and non-real-time threads.
- Conduct experiments to measure the impact of real-time scheduling and CPU affinity on thread execution.
- Present an analysis of the experiments, including observed behaviors such as CPU usage, thread scheduling, and performance comparisons between real-time and non-real-time threads.

2.Experimental Setup

- **Methodology:**

1. **Patching the Linux Kernel for RT Support**:
    - The kernel was patched to support real-time capabilities using the PREEMPT_RT patch. This was done by obtaining the kernel source, applying the patch, and then compiling and installing the kernel on the Raspberry Pi.
2. **Thread Creation and Management**:
    - Threads were created using the POSIX pthread API, with real-time and non-real-time (NRT) threads being handled differently.

    - Real-time threads were configured using the SCHED_FIFO and SCHED_RR scheduling policies to ensure the system could prioritize time-sensitive tasks.

3. **CPU Affinity**:
    - Threads were initially bound to a specific CPU (CPU 1) using pthread_setaffinity_np() to ensure that real-time threads ran on the designated CPU. Later, experiments were conducted where threads were allowed to run freely on available CPUs, without explicit CPU binding, to observe any effects on scheduling and performance.

4. **Performance Measurement**:
    - The wall-time of thread execution was measured using gettimeofday() to track the execution time of each thread and application.
    - Different scheduling policies and CPU binding configurations were used to test their impact on thread execution time and system behavior.

- **Code and Commands Used**

1. **Kernel Patch and Build**:
    - The Raspberry Pi kernel was patched to support real-time functionality with the PREEMPT_RT patch:

```
$ git clone --depth=1 --branch rpi-6.1.y https://github.com/raspberrypi/linux
$ wget https://mirrors.edge.kernel.org/pub/linux/kernel/projects/rt/6.1/patch-6.1-rc7
$ zcat ../patch-6.1-rc7-rt5.patch.gz | patch -p1
$ make bcm2711_defconfig
$ make menuconfig  # Customizing for real-time support
$ make -j4 Image.gz modules dtbs
$ sudo make modules_install
$ sudo cp arch/arm64/boot/dts/broadcom/*.dtb /boot/
$ sudo cp arch/arm64/boot/Image.gz /boot/kernel8.img
```

- These commands patched the kernel and built it with the PREEMPT_RT patch, after which the Raspberry Pi was rebooted with the new real-time kernel.

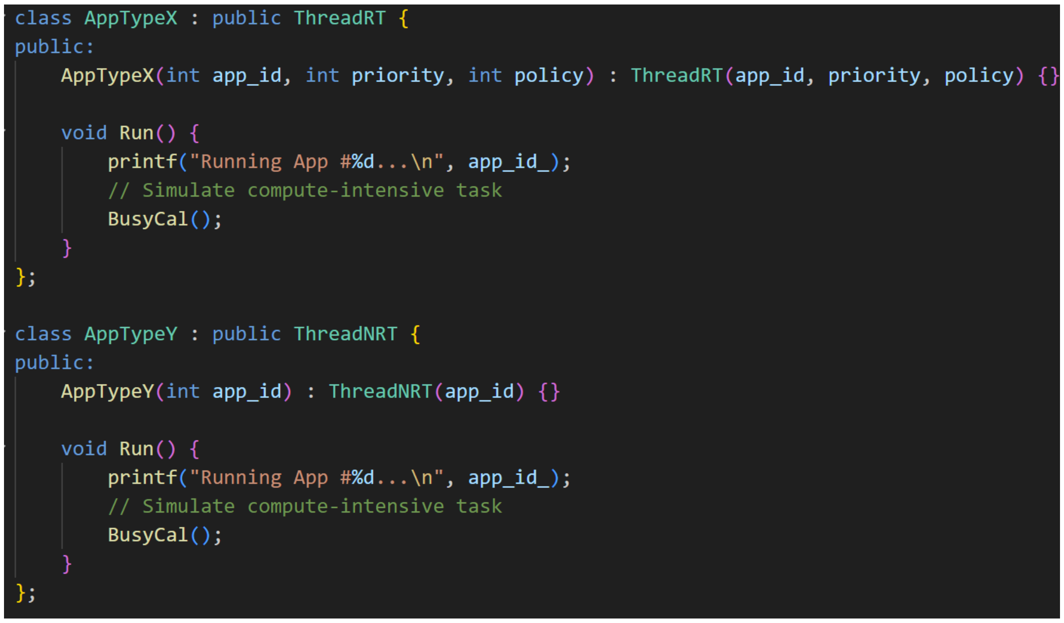
2. Thread Classes:

- The thread classes (ThreadRT for real-time threads and ThreadNRT for non-real-time threads) were defined to encapsulate the creation, execution, and management of the threads.
- Real-time threads were created using pthread_create(), with custom attributes for scheduling policy and priority.
- Example of ThreadRT class for managing real-time threads:

```
void Start() {
    pthread_attr_t thread_attr;
    pthread_attr_init(&thread_attr);
    pthread_attr_setschedpolicy(&thread_attr, policy_);
    sched_param param;
    param.sched_priority = priority_;
    pthread_attr_setschedparam(&thread_attr, &param);
    pthread_attr_setinheritsched(&thread_attr, PTHREAD_EXPLICIT_SCHED);
    gettimeofday(&start_time, NULL);
    pthread_create(&thread_, &thread_attr, &ThreadRT::RunThreadRT, this);
    pthread_attr_destroy(&thread_attr);
}
```

- This code snippet shows how the scheduling policy (SCHED_FIFO, SCHED_RR) and priority were set for real-time threads. The RunThreadRT function executes the real-time workload.

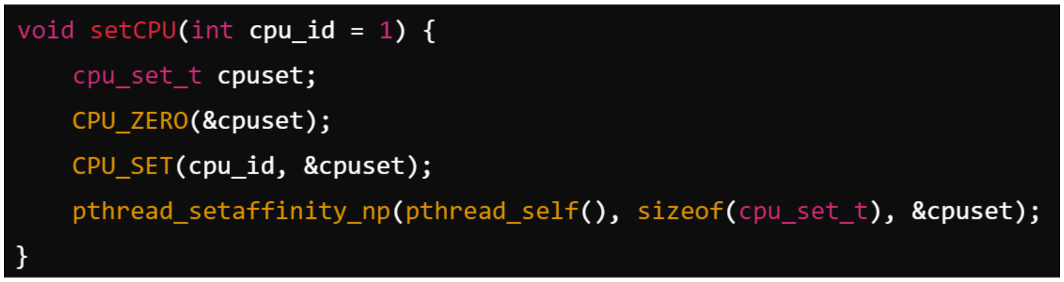
3. Verify AppTypeX and AppTypeY



```
class AppTypeX : public ThreadRT {
public:
    AppTypeX(int app_id, int priority, int policy) : ThreadRT(app_id, priority, policy) {}

    void Run() {
        printf("Running App #%d...\n", app_id_);
        // Simulate compute-intensive task
        BusyCal();
    }
};

class AppTypeY : public ThreadNRT {
public:
    AppTypeY(int app_id) : ThreadNRT(app_id) {}

    void Run() {
        printf("Running App #%d...\n", app_id_);
        // Simulate compute-intensive task
        BusyCal();
    }
};
```

4. **CPU Affinity**:

- CPU affinity was set to bind threads to CPU 1:



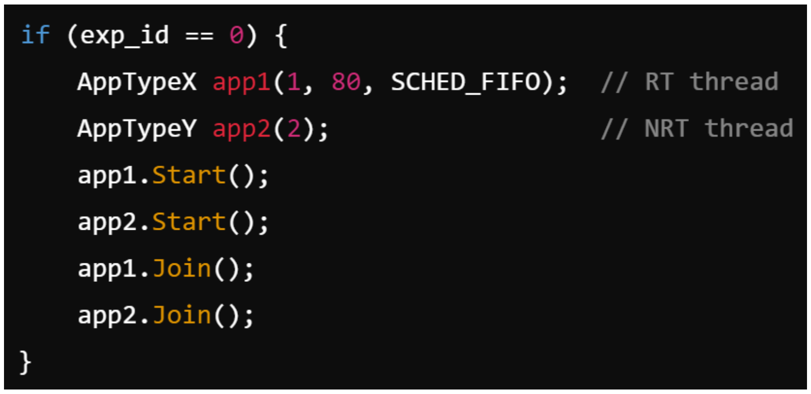
```
void setCPU(int cpu_id = 1) {
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(cpu_id, &cpuset);
    pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpuset);
}
```

- This ensured that the real-time threads ran on the specified CPU, but for some experiments, the CPU binding was disabled to allow the threads to run freely on any available CPU (#define SET_CPU false).

5. **Experiments and Output**:

   - Five different experimental configurations were tested, where the exp_id was passed as a command-line argument. Each experiment varied the number of threads (RT and NRT), the scheduling policies (SCHED_FIFO, SCHED_RR), and whether the threads were bound to a specific CPU or allowed to run freely on available CPUs.

   - Example of running the experiment:



```
if (exp_id == 0) {
    AppTypeX app1(1, 80, SCHED_FIFO);  // RT thread
    AppTypeY app2(2);                  // NRT thread
    app1.Start();
    app2.Start();
    app1.Join();
    app2.Join();
}
```

6. **Performance Measurement**:

   - The wall-time for each thread was measured using gettimeofday() in the Start() and Join() methods to track execution time.

3.Results

- **Experiment 1 Results:**

  - **Setup**: One CannyP3 RT app (SCHED_FIFO, priority 80) and two NRT apps, all running on CPU=1.

  - **Output**:

        - The RT thread executed with a priority of 80 and SCHED_FIFO policy.
        - The NRT threads showed a longer runtime compared to the RT thread, with the following runtimes:
            - App #1 (RT): 2.32 seconds
            - App #2 (NRT): 6.81 seconds
            - App #3 (NRT): 6.80 seconds
        - Both NRT threads ended after a significantly longer time, which suggests that they were preempted or experienced less priority in comparison to the RT thread.

- **Experiment 2 Results:**
    - **Setup**: Same as Experiment 1, but freely running on available CPUs.
    - **Output**:
        - The RT thread ran on CPU #2, while the NRT threads ran on CPU #1 and CPU #0.
        - The runtimes were significantly improved for the NRT threads, indicating that allowing the threads to run on freely available CPUs reduced the overall execution time:
            - App #1 (RT): 2.32 seconds
            - App #2 (NRT): 2.32 seconds
            - App #3 (NRT): 2.31 seconds

    - All threads completed in a similar timeframe, suggesting that they did not face CPU contention when freely allocated across multiple CPUs.

- **Experiment 3 Results:**
  - **Setup**: Two SCHED_FIFO RT apps and one NRT app, all running on CPU=1.
  - **Output**:
    - The RT threads executed on CPU #1 with priority 80 and SCHED_FIFO policy.
    - The second RT thread (App #2) had a slightly higher runtime (4.73 seconds) compared to the first (App #1, 2.37 seconds), indicating that the second RT thread had to wait for the first thread to finish or was delayed by scheduling.
    - The NRT thread (App #3) executed in 6.80 seconds, which is similar to its runtime in Experiment 1, suggesting that the RT threads' behavior did not impact its execution significantly.
    - The result shows that scheduling multiple RT threads on the same CPU led to increased execution time for each RT thread.
- **Experiment 4 Results:**
  - **Setup**: Two SCHED_RR RT apps and one NRT app, all running on CPU=1.
  - **Output**:

    - Both RT threads were scheduled with SCHED_RR policy and had priority 80.
    - Similar to Experiment 3, the second RT thread (App #2) took longer to execute (4.73 seconds) compared to the first RT thread (App #1, 4.66 seconds).
    - The NRT thread (App #3) executed in 6.80 seconds, again showing consistency in its behavior.
    - The round-robin (SCHED_RR) scheduling policy resulted in slightly longer execution times for the RT threads compared to the SCHED_FIFO policy used in Experiment 3.

- **Experiment 5 Results:**
  - **Setup**: Same as Experiment 3, but freely running on available CPUs.
  - **Output**:
    - The RT threads were distributed across CPU #1 and CPU #2, while the NRT thread ran on CPU #0.
    - All threads completed with very similar runtimes:
      - App #1 (RT): 2.32 seconds
      - App #2 (RT): 2.32 seconds
      - App #3 (NRT): 2.32 seconds
    - The free allocation of CPUs led to significantly improved performance, as compared to Experiment 3, where the threads were bound to CPU=1.

- **Comparison of Results:**

Experiment	**App #1 (RT)**	**App #2 (NRT)**	**App #3 (NRT)**
Exp. 1 (CPU=1)	2.32 seconds	6.81 seconds	6.80 seconds
Exp. 2 (Freely on CPUs)	2.32 seconds	2.32 seconds	2.31 seconds
Experiment	**App #1 (RT)**	**App #2 (RT)**	**App #3 (NRT)**
Exp. 3 (2 RT, CPU=1)	2.37 seconds	4.73 seconds	6.80 seconds
Exp. 4 (2 RT, SCHED_RR)	4.66 seconds	4.73 seconds	6.80 seconds
Exp. 5 (Freely on CPUs, 2 RT)	2.32 seconds	2.32 seconds	2.32 seconds

- **Observations and explanations:**

1. **CPU Affinity Impact**:
   - In **Experiments 1, 3, and 4**, where threads were bound to CPU=1, the real-time threads (RT) experienced delays when multiple RT threads were scheduled on the same CPU. This increased the execution time for each thread.
   - In **Experiments 2 and 5**, where threads were allowed to run on freely available CPUs, the execution times for all threads were significantly reduced. The real-time threads were able to execute more efficiently when they were not confined to a single CPU, with minimal impact from non-real-time threads.
2. **Scheduling Policy**:
   - The **SCHED_FIFO** policy in **Experiment 1** resulted in predictable and consistent real-time behavior. However, in **Experiment 3**, where multiple RT threads were

scheduled with SCHED_FIFO, the second thread experienced longer execution times due to CPU contention.

  - The **SCHED_RR** policy in **Experiment 4** provided better fairness by allowing each RT thread to take turns executing but resulted in slightly higher runtimes for both RT threads compared to SCHED_FIFO.

3. **Real-Time vs Non-Real-Time**:
   - The **NRT threads** always had longer execution times compared to the RT threads in CPU-bound experiments. However, when threads were allowed to run on freely available CPUs (Experiments 2 and 5), the performance of all threads improved significantly.

4. **Consistency**:
   - The consistency of execution time across experiments suggests that freeing threads from CPU restrictions allows for better load balancing, which helps reduce contention between threads, especially when multiple threads are executed simultaneously.

4.Problems and Discussion

- **Challenge:**
  - One challenge that I encountered during the project was the issue with the kernel patch URL provided in the assignment description. The URL for the PREEMPT_RT patch (https://mirrors.edge.kernel.org/pub/linux/kernel/projects/rt/

6.1/patch-6.1-rc7-rt5.patch.gz) resulted in a **404 error** when I tried to download it, meaning the file was not available at that location. This was a critical part of the project since patching the kernel with the PREEMPT_RT patch was essential for enabling real-time capabilities on the Raspberry Pi.

- **Solution**
  - To overcome this issue, I replaced the broken URL with an alternative working link that pointed to the same patch version but from a different server. The updated link (https://www.kernel.org/pub/linux/kernel/projects/rt/6.1/older/patch-6.1-rc7-rt5.patch.gz) provided the same patch file and allowed me to proceed with the kernel patching process successfully. Once I downloaded the patch from the new link, I followed the rest of the instructions to apply and build the patched kernel as specified in the assignment.
- **Results vs. Expectations**
  - The results of the experiments largely agreed with my expectations. I anticipated that the real-time threads would have more predictable execution times when bound to a specific CPU, and that non-real-time threads would have longer execution times due to lower priority. This was confirmed in **Experiment 1**, where the RT thread had a significantly lower runtime compared to the NRT threads when all threads were running on the same CPU.
  - Moreover, I expected that allowing threads to run freely on multiple CPUs would reduce the execution times due to

better load balancing, which was confirmed in **Experiments 2 and 5**. The free allocation of CPUs allowed the threads to run more efficiently, minimizing the impact of CPU contention.

- However, I did not initially expect that the **SCHED_RR** policy would cause a noticeable increase in runtime for real-time threads compared to the **SCHED_FIFO** policy. This result was a bit surprising because I had expected **SCHED_RR** to provide a more balanced execution, but it seemed to introduce some overhead in scheduling. This was evident in **Experiment 4**, where the execution times of the RT threads were slightly higher than in **Experiment 3** (which used SCHED_FIFO).