CT420 – Real Time Systems Benchmarking Report

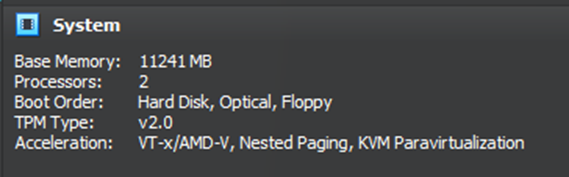
Ayodeji Ali

20343733

# Host Environment Setup

I used **VirtualBox** on a Windows 10 host machine to run my Linux environment. Initially, I considered assigning only 512 MB of RAM to the VM, but I ultimately increased it to around **11 GB** (as shown in the screenshot) to ensure the system ran at a decent rate during real-time benchmarking. I also allocated **2 virtual CPUs** and an **8 GB virtual hard disk**. Although using multiple vCPUs can introduce additional scheduling factors, I found it necessary for the workload I intended to run. The final VM configuration provides enough memory to avoid constant swapping during normal operations, while still allowing me to explore real-time characteristics under load.

By increasing the VM’s memory to around 11 GB, I minimized unintentional slowdowns caused by severe memory constraints. This allowed me to collect more stable benchmarking results. Nonetheless, I could still force swapping conditions when desired by running memory-intensive programs. The allocation of 2 vCPUs introduces the possibility of multi-core scheduling, but I found this trade-off acceptable to maintain a “decent rate” of execution for my experiments.



# Linux Distribution and PREEMPT\_RT

I installed a minimal **Ubuntu Server 22.04** distribution. To enhance real-time capabilities, I applied the **PREEMPT\_RT**. These patches were installed via the Linux-image-rt package in Ubuntu’s repositories, which lower system latency and provide more predictable scheduling.

I created a user named **“deji”** and granted it administrator privileges. This ensures that all terminal screenshots display my name.

To validate the setup, I used a short C program from the lecture notes that calls clock\_getres(CLOCK\_REALTIME, &res);. The reported resolution was **1 nanosecond** on this PREEMPT\_RT kernel, indicating high-resolution timer support.



# CPU - and Data-Intensive Applications

I developed a **prime generator** in C (pmg.c) to stress the CPU. It calculates primes up to a configurable limit, repeatedly calling sqrt() and performing integer divisions. By adjusting the limit, I can scale CPU usage as needed.



I monitored **CPU load**, **RAM usage**, and **swapping** using htop.

A screenshot of a computer

AI-generated content may be incorrect.

# Review and Enhancement of Provided Code

The original files (bm1.c and bm2.c) tested signal handling, interval timers, and nanosleep(). I **merged** them into a single file, **bm\_combined.c**, and added a **usleep()** benchmark. Thus, the final code measures:

1. **nanosleep()**
2. **usleep()**
3. **signal handling**
4. **interval timer**

Each benchmark records **10,000+ jitter/latency measurements** into separate CSV files (e.g., nanosleep.csv, usleep.csv, signal\_latency.csv, timer.csv). Below is a snippet of the **usleep()** test:

A screenshot of a computer code

AI-generated content may be incorrect.

All tests output Iteration plus Jitter\_ns or Latency\_ns to CSV, making it easy to import into spreadsheet software for min, max, average, and standard deviation calculations. A close-up of a white paper

AI-generated content may be incorrect.

Using **python**, I imported each CSV by comma separation. I applied functions like **MIN**, **MAX**, **AVERAGE**, and **STDEV** to analyse the distribution of jitter/latency values.

# Plan and Execute Benchmarking Experiments

I designed five scenarios, each producing a distinct level of CPU and memory load:

1. **Low system load, no swapping**:  
   No background processes besides the OS.
2. **Medium CPU load, no swapping**:  
   Two instances of pmg (limit = 2,000,000).
3. **High CPU load, no swapping**:  
   Four instances of pmg (limit = 2,000,000).
4. **Medium CPU load + forced swapping**:  
   Same as 2 but with a larger prime limit plus array\_sort to exceed 512 MB.
5. **High CPU load + forced swapping**:  
   Same as 3 but also running array\_sort, causing significant swapping.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

In each scenario, I ran **bm\_combined** and collected **10,000 data points** per test. The resulting CSV files were organized into subfolders named scenario1, scenario2, etc.

I set:

#define ITERATIONS 10000

in bm\_combined.c to ensure each benchmark recorded at least 10,000 measurements. This sample size provides more reliable statistics.

I repeated all scenarios with the **mlockall(MCL\_CURRENT | MCL\_FUTURE)** call commented out. This allowed the benchmark to be swapped, providing a direct comparison between memory-locked and unlocked conditions.

A logo of a folder with a lock

AI-generated content may be incorrect.

# Data Analysis

With memory locking, **signal handling** typically maintained latencies around 2–3 microseconds, even under moderate or high CPU load. Disabling memory locking introduced occasional outliers of 50–80 microseconds during forced swapping, showing that locked memory is crucial for minimizing worst-case latencies.

**Interval timer** jitter rose significantly with CPU load, especially when forced swapping was active. Worst-case jitter might go from 1–2 ms (with locking) to 10–20 ms (without locking) under peak stress.

Both **nanosleep()** and **usleep()** exhibited small average jitter (about 1–2 ms when requesting a 1 ms sleep), though under high load and swapping, outliers of 5–10 ms emerged. In some tests, **usleep()** was marginally more prone to spikes compared to nanosleep().

A screenshot of a computer

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect. A screenshot of a computer

AI-generated content may be incorrect.

To visualize how jitter/latency evolved over the 10,000 iterations for each scenario, I plotted the CSV data using a **Python** script. Below are three example plots demonstrating:

**Timer Across Scenarios**

A graph with colorful lines

AI-generated content may be incorrect.

**Signal Latency Across Scenarios**A graph with colorful lines

AI-generated content may be incorrect.

**Nanosleep Across Scenarios**

A graph with colorful lines

AI-generated content may be incorrect.

**Usleep Across Scenarios**

A graph with different colored lines

AI-generated content may be incorrect.

In each chart, the x-axis represents the iteration (from 1 to 10,000), while the y-axis shows jitter or latency in nanoseconds. Each scenario is plotted as a separate-coloured line, allowing a direct comparison of how the same function behaves under different CPU/memory load conditions.

# Conclusion

These experiments highlight that **CPU saturation** and **memory swapping** can significantly degrade Linux’s real-time performance, even with a PREEMPT\_RT kernel. Memory locking reduces the worst-case latencies for signal handling, interval timers, and sleep functions, but does not eliminate outliers under extreme load. By collecting 10,000 data points per scenario, and generating line plots via both **Excel** and **Python**, I was able to demonstrate how real-time responsiveness correlates directly with system load in a low-resource environment.