Name: Andrew Hayes
Student ID: 21321503
E-mail: a.hayes18@universityofgalway.ie

CT420

2025–03–19

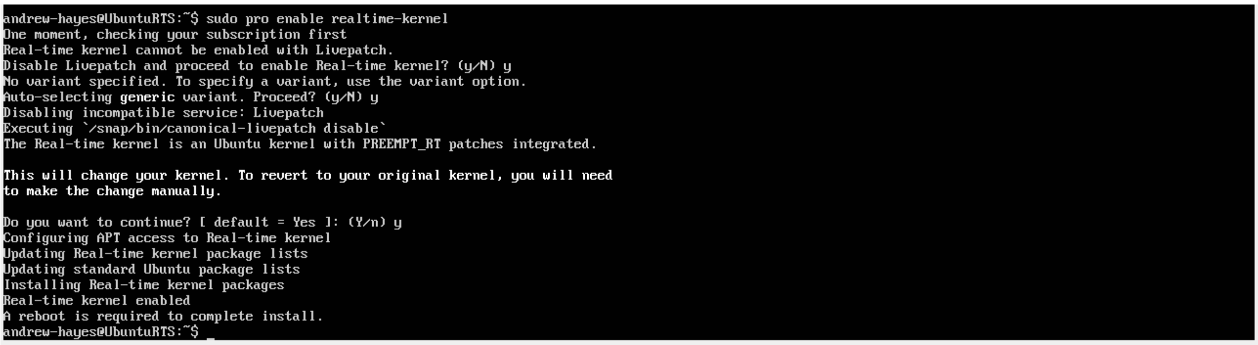
Assignment 2: POSIX Programming & Benchmarking

1 Host Environment

For my host environment, I chose to run Ubuntu Server 24.04.2 LTS using a VirtualBox hypervisor. I chose this operating system as I have sufficient Linux experience to feel confident using an operating system with no graphical interface (as opposed to Ubuntu Desktop), and the absence of a GUI means a smaller ISO file, memory footprint, & CPU footprint. I chose Ubuntu specifically because it's a Linux system with which I have previous experience, and is well-document with plenty of packages available to install if needs be. Ubuntu also makes it easy to install the `PREEMPT_RT` patches, which transform the standard Linux kernel into a fully preemptible, real-time kernel, which I felt was more suitable for this assignment, as the standard Linux kernel is not suitable for a hard real-time system due to its lack of preemption.

Figure 1: Virtual machine hardware configuration

I set the virtual machine to have a single CPU and set the amount of RAM to 2048MB which is the recommended minimum for Ubuntu Server[1]. I left the hard disk size at the default of 25GB as I saw no reason to change it. The real-time kernel with the `PREEMPT_RT` patches installed is available with Ubuntu Pro, which is free for personal use. After setting up an Ubuntu Pro account, I enabled the real-time kernel using the `pro` command.



```
andrew-hayes@UbuntuRTS:~$ sudo pro enable realtime-kernel
One moment, checking your subscription first
Real-time kernel cannot be enabled with Livepatch.
Disable Livepatch and proceed to enable Real-time kernel? (y/N) y
No variant specified. To specify a variant, use the variant option.
Auto-selecting generic variant. Proceed? (y/N) y
Disabling incompatible service: Livepatch
Executing `/snap/bin/canonical-livepatch disable`
The Real-time kernel is an Ubuntu kernel with PREEMPT_RT patches integrated.

This will change your kernel. To revert to your original kernel, you will need
to make the change manually.

Do you want to continue? [ default = Yes ]: (Y/n) y
Configuring APT access to Real-time kernel
Updating Real-time kernel package lists
Updating standard Ubuntu package lists
Installing Real-time kernel packages
Real-time kernel enabled
A reboot is required to complete install.
andrew-hayes@UbuntuRTS:~$ _
```

Figure 2: Enabling the real-time kernel with the `pro` command

Finally, I transferred over the following C file (taken from the lecture slides) via `scp`[3] to the virtual machine to get the clock resolution, which is 1 nanosecond:

```c
#include<unistd.h>
#include<time.h>
#include <stdio.h>

int main(){
    struct timespec clock_res;
    int stat;
    stat=clock_getres(CLOCK_REALTIME, &clock_res);
    printf("Clock resolution is %d seconds, %ld nanoseconds\n",clock_res.tv_sec,clock_res.tv_nsec);
    return 0;
}
```

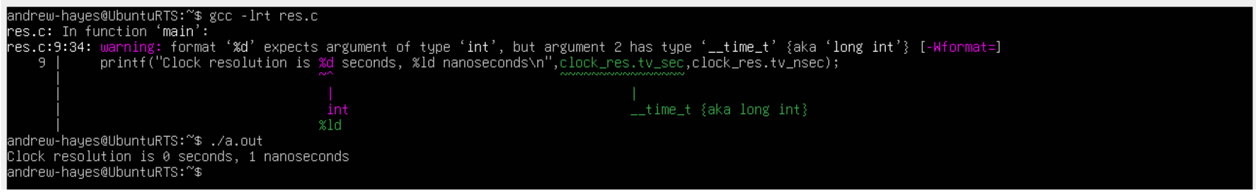


Figure 3: Getting the clock resolution of the virtual machine

2 Benchmarking Code

I combined the provided benchmarking programs `bm1.c` and `bm2.c` into a single file, and added logic to benchmark the `usleep()` function as well as outputting the relevant data to CSV files. Additionally, I updated the `#define ITERATIONS` constant to have value `10000` and I also tweaked the `while (!timer_expired)` loop to sleep for 100 nanoseconds in-between evaluations of the loop condition, as I found that the “busy waiting” was greatly slowing down the program when I ran it on my virtual machine. There is a potential drawback to this however: adding the `nanosleep()` to the while loop could artificially introduce a delay into obtaining the data, as there could be a maximum delay of 100 nanoseconds before the timer is registered as expired. However, since the busy wait was artificially increasing the runtime, and much more so than the version with the sleep, it too would introduce delay, and much more than the modified version, as the modified version ran around 10 times more quickly. Therefore, while this modification could potentially introduce noise to the data collected for the interval timer benchmark, it introduces less error than the busy wait, so I decided to include the modification.

```c
// Compile code with gcc -o merged merged.c -lrt -Wall -O2
// Execute code with sudo ./merged

#include <stdio.h>      // Standard I/O functions
#include <stdlib.h>     // Standard library functions
#include <time.h>       // Time-related functions
#include <signal.h>     // Signal handling
#include <sys/mman.h>   // Memory locking
#include <unistd.h>     // POSIX standard functions
#include <sched.h>      // Scheduling policies
#include <errno.h>      // Error handling
#include <string.h>     // String manipulation
#include <limits.h>     // Limits of integral types

// Constants
#define ITERATIONS 10000    // Number of benchmark iterations
#define NS_PER_SEC 1000000000L // Nanoseconds per second

// Global Variables
timer_t timer_id;  // Timer identifier
volatile sig_atomic_t timer_expired = 0;  // Flag for timer expiration
volatile sig_atomic_t signal_received = 0; // Flag for signal reception
struct timespec start, end, sleep_time; // Time structures for benchmarking

// Function to save benchmark results to a CSV file
void save_results(const char *filename, long long *data) {
    FILE *file = fopen(filename, "w");
    if (!file) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }
    fprintf(file, "Iteration,Latency/Jitter (ns)\n");
    for (int i = 0; i < ITERATIONS; i++) {
        fprintf(file, "%d,%lld\n", i, data[i]);
    }
    fclose(file);
}

// Signal handler for signal-based latency measurement
```

```c
void signal_handler(int signum) {
    signal_received = 1; // Mark signal as received
    clock_gettime(CLOCK_MONOTONIC, &end); // Capture end time
}

// Timer signal handler
void timer_handler(int signum) {
    timer_expired = 1; // Mark timer as expired
    clock_gettime(CLOCK_MONOTONIC, &end); // Capture end time
}

// Configures real-time scheduling with FIFO priority
void configure_realtime_scheduling() {
    struct sched_param param;
    param.sched_priority = sched_get_priority_max(SCHED_FIFO);
    if (sched_setscheduler(0, SCHED_FIFO, &param) == -1) {
        perror("sched_setscheduler");
        exit(EXIT_FAILURE);
    }
}

// Locks memory to prevent paging for real-time performance
void lock_memory() {
    if (mlockall(MCL_CURRENT | MCL_FUTURE) == -1) {
        perror("mlockall");
        exit(EXIT_FAILURE);
    }
}

// Measures jitter of nanosleep function
void benchmark_nanosleep() {
    long long jitter_data[ITERATIONS];
    sleep_time.tv_sec = 0;
    sleep_time.tv_nsec = 1000000; // 1 ms sleep

    for (int i = 0; i < ITERATIONS; i++) {
        clock_gettime(CLOCK_MONOTONIC, &start);
        nanosleep(&sleep_time, NULL);
        clock_gettime(CLOCK_MONOTONIC, &end);

        jitter_data[i] = ((end.tv_sec - start.tv_sec) * NS_PER_SEC + (end.tv_nsec - start.tv_nsec)) -
        ↪  sleep_time.tv_nsec;
    }
    save_results("nanosleep.csv", jitter_data);
}

// Measures latency of sending and handling a signal
void benchmark_signal_latency() {
    long long latency_data[ITERATIONS];
    signal(SIGUSR1, signal_handler); // Register signal handler

    for (int i = 0; i < ITERATIONS; i++) {
        clock_gettime(CLOCK_MONOTONIC, &start);
        kill(getpid(), SIGUSR1); // Send signal to itself
        while (!signal_received); // Wait for signal to be handled

        latency_data[i] = (end.tv_sec - start.tv_sec) * NS_PER_SEC + (end.tv_nsec - start.tv_nsec);
        signal_received = 0;
    }
    save_results("signal_latency.csv", latency_data);
}
```

```c

// Measures jitter of a real-time timer
void benchmark_timer() {
    long long jitter_data[ITERATIONS];
    struct sigevent sev;
    sev.sigev_notify = SIGEV_SIGNAL;
    sev.sigev_signo = SIGRTMIN;
    sev.sigev_value.sival_ptr = &timer_id;

    if (timer_create(CLOCK_MONOTONIC, &sev, &timer_id) == -1) {
        perror("timer_create");
        exit(EXIT_FAILURE);
    }

    struct itimerspec its;
    its.it_value.tv_sec = 0;
    its.it_value.tv_nsec = 1000000; // 1 ms
    its.it_interval = its.it_value;
    signal(SIGRTMIN, timer_handler);

    if (timer_settime(timer_id, 0, &its, NULL) == -1) {
        perror("timer_settime");
        exit(EXIT_FAILURE);
    }
    clock_gettime(CLOCK_MONOTONIC, &start);
    for (int i = 0; i < ITERATIONS; i++) {
        while (!timer_expired) {
            struct timespec ts = {0, 100};
            nanosleep(&ts, NULL);
        }

        clock_gettime(CLOCK_MONOTONIC, &end);
        jitter_data[i] = ((end.tv_sec - start.tv_sec) * NS_PER_SEC + (end.tv_nsec - start.tv_nsec)) -
        ↪  its.it_interval.tv_nsec;
        timer_expired = 0;
        start = end;
    }
    save_results("timer.csv", jitter_data);
}

// Measures jitter of usleep function
void benchmark_usleep() {
    long long jitter_data[ITERATIONS];

    for (int i = 0; i < ITERATIONS; i++) {
        clock_gettime(CLOCK_MONOTONIC, &start);
        usleep(1000); // Sleep for 1 ms
        clock_gettime(CLOCK_MONOTONIC, &end);

        jitter_data[i] = ((end.tv_sec - start.tv_sec) * NS_PER_SEC + (end.tv_nsec - start.tv_nsec)) -
        ↪  1000000;
    }
    save_results("usleep.csv", jitter_data);
}

// Main function to execute all benchmarks
int main() {
    configure_realtime_scheduling(); // Set high priority scheduling
    lock_memory(); // Prevent memory paging

    printf("Getting nanosleep benchmark\n");
```

```c
    benchmark_nanosleep();

    printf("Getting signal benchmark\n");
    benchmark_signal_latency();

    printf("Getting timer benchmark\n");
    benchmark_timer();

    printf("Getting usleep benchmark\n");
    benchmark_usleep();

    return 0;
}
```

Listing 1: `merged.c`

3 CPU & Data-Intensive Applications

To develop my CPU & data-intensive programs, I chose to use Python for ease of development. I chose `htop`[2] as my resource-monitoring tool as I have often used it in the past, it has easy to read & understand output, and shows you exactly what proportion of the CPU & memory is in use at that time. It also allows you to list processes by CPU consumption or memory consumption which is a useful option to have for this assignment.

```python
import multiprocessing
import time
import argparse
import os

def stress_cpu(workload: float):
    """
    Function to create CPU load. Uses a busy-wait method to simulate CPU usage.

    :param workload: The fraction of time (0.0 to 1.0) the CPU should be busy.
    """
    cycle_time = 0.1  # Total cycle time (100ms per iteration)
    busy_time = cycle_time * workload  # Time to stay busy
    idle_time = cycle_time - busy_time  # Time to stay idle

    while True:
        start_time = time.time()
        while (time.time() - start_time) < busy_time:
            pass  # Busy wait
        time.sleep(idle_time)  # Sleep to control CPU usage

def start_stress_test(load: str):
    """
    Starts CPU stress test based on load level.

    :param load: 'medium' (~50% load) or 'high' (~100% load)
    """
    num_cores = os.cpu_count() or 4  # Use all available CPU cores
    workload = 0.5 if load == "medium" else 1.0  # Set workload percentage

    print(f"Starting {load.upper()} CPU stress test on {num_cores} cores...")

    processes = []
    for _ in range(num_cores):
        p = multiprocessing.Process(target=stress_cpu, args=(workload,))
        p.start()
        processes.append(p)

```

```python
    try:
        for p in processes:
            p.join()
    except KeyboardInterrupt:
        print("Stopping stress test...")
        for p in processes:
            p.terminate()
            p.join()

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="CPU Stress Test Script")
    parser.add_argument("--load", choices=["medium", "high"], required=True, help="Choose CPU load level
    ↪  (medium or high)")
    args = parser.parse_args()

    start_stress_test(args.load)
```

Listing 2: `stress_cpu.py`

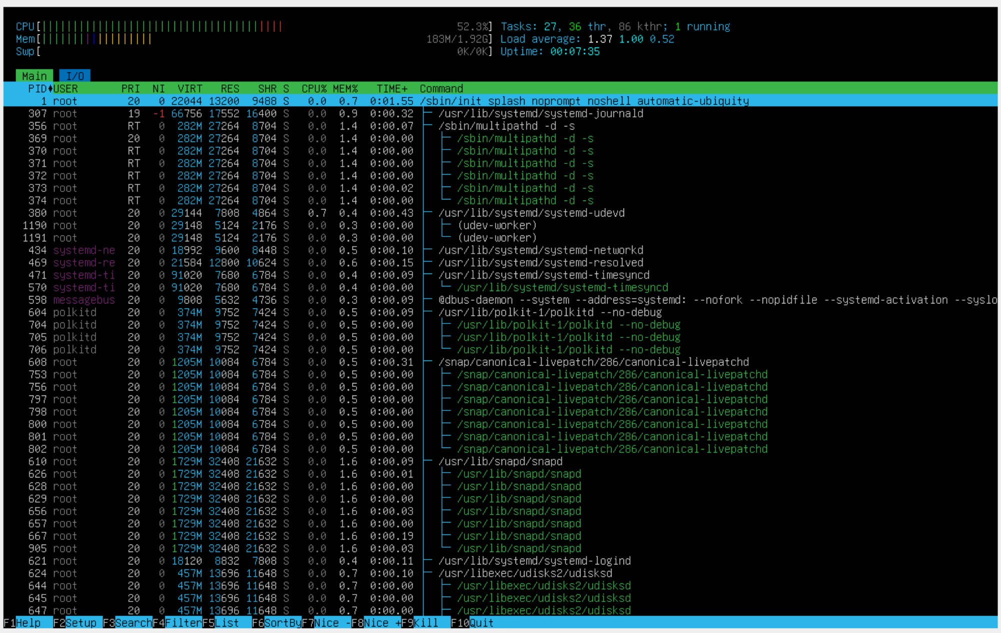


Figure 4: `htop` output when running `python3 stress_cpu.py --load medium`

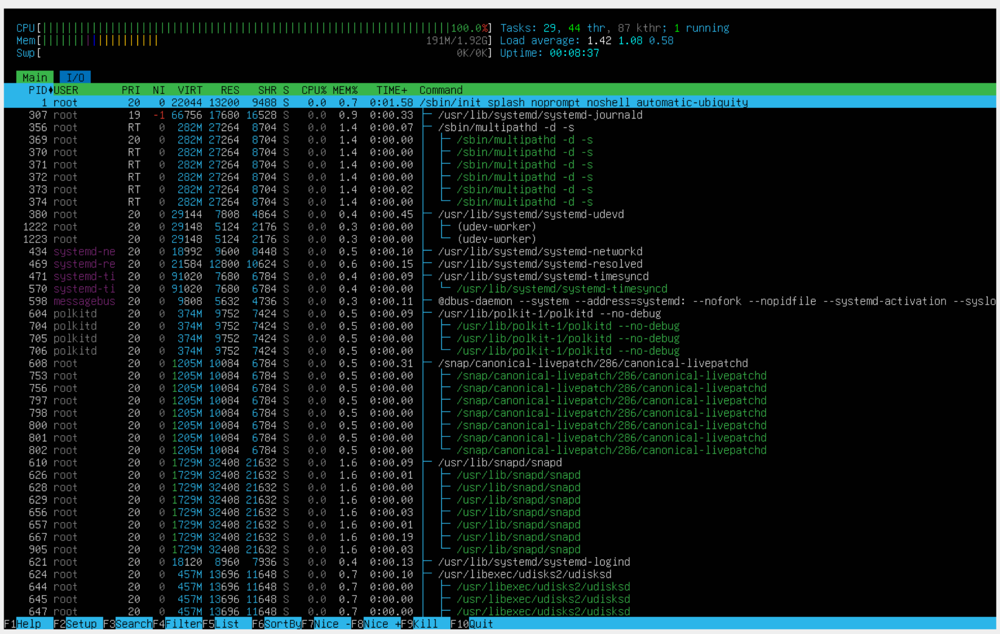


Figure 5: htop output when running `python3 stress_cpu.py --load high`

```python
import argparse
import time
import psutil

def stress_memory(target_usage: float):
    """
    Stress the system memory to a given percentage.

    :param target_usage: Target memory usage (0.0 to 1.0, where 1.0 is 100%)
    """
    total_memory = psutil.virtual_memory().total  # Get total RAM in bytes
    target_memory = int(total_memory * target_usage)  # Calculate target memory size

    print(f"Total Memory: {total_memory / (1024**3):.2f} GB")
    print(f"Target Memory Usage: {target_memory / (1024**3):.2f} GB ({target_usage * 100:.0f}%)")

    try:
        memory_hog = []  # List to store allocated memory chunks
        chunk_size = 100 * 1024 * 1024  # Allocate in 100MB chunks

        while sum(len(chunk) for chunk in memory_hog) < target_memory:
            memory_hog.append(bytearray(chunk_size))  # Allocate memory
            time.sleep(0.1)  # Small delay to allow system response

        print("Memory fully allocated. Holding...")
        while True:  # Keep the memory occupied
            time.sleep(1)

    except MemoryError:
        print("Memory limit reached. Exiting...")
    except KeyboardInterrupt:
        print("Memory stress test stopped.")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Memory Stress Test Script")
    parser.add_argument("--usage", type=float, default=1.0, help="Target memory usage (default: 1.0 for
    ↪  100%)")
    args = parser.parse_args()
```

```
    stress_memory(args.usage)
```

Listing 3: `stress_memory.py`

I found that the maximum `--usage` value I could set without getting the process killed by the Linux kernel's Out-Of-Memory (OOM) killer was `0.85`, so this is the value I used for my experiments.

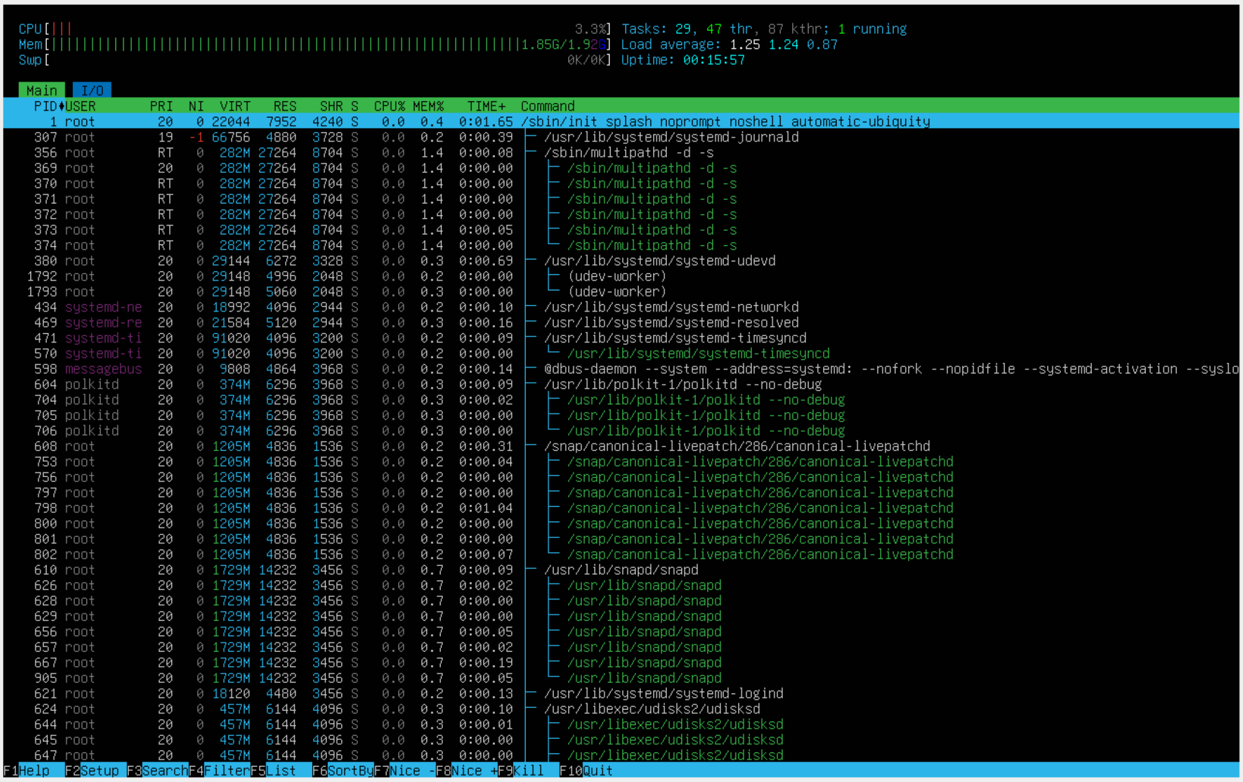


Figure 6: `htop` output when running `python3 stress_memory.py --usage 0.85`

4 Experiments

I ran the experiments in quick succession on the virtual machine by running the appropriate stresser script(s), forking it into the background using the `&` shell operator, and running the merged benchmark program. I then transferred the generated CSV files to my host machine using `scp`. To generate the plots, I wrote a Python script which will plot the mean, minimum, maximum, or standard deviation of the values collected in a bar chart for a number of given CSV files.

```python
import pandas as pd
import matplotlib.pyplot as plt
import os
import argparse

parser = argparse.ArgumentParser(description="Plot specified metric from CSV files.")
parser.add_argument("metric", choices=["min", "max", "mean", "std"], help="Metric to plot (min, max, mean,
↪  std)")
args = parser.parse_args()

metric_to_plot = args.metric.lower()
valid_metrics = {"min": "Min", "max": "Max", "mean": "Mean", "std": "Std"}

csv_files = [
    ("../../data/Locking Enabled/1. Low CPU Load, No Swap/usleep.csv",       "Locking Enabled, Low CPU
    ↪  Load, No Swap"),
    ("../../data/Locking Enabled/2. Medium CPU Load, No Swap/usleep.csv",    "Locking Enabled, Medium CPU
    ↪  Load, No Swap"),
```

```python
    ("../../data/Locking Enabled/3. High CPU Load, No Swap/usleep.csv",      "Locking Enabled, High CPU Load, No Swap"),
    ("../../data/Locking Enabled/4. Medium CPU Load, Swap/usleep.csv",       "Locking Enabled, Medium CPU Load, Swap"),
    ("../../data/Locking Enabled/5. High CPU Load, Swap/usleep.csv",         "Locking Enabled, High CPU Load, Swap"),
    ("../../data/Locking Disabled/1. Low CPU Load, No Swap/usleep.csv",      "Locking Disabled, Low CPU Load, No Swap"),
    ("../../data/Locking Disabled/2. Medium CPU Load, No Swap/usleep.csv",   "Locking Disabled, Medium CPU Load, No Swap"),
    ("../../data/Locking Disabled/3. High CPU Load, No Swap/usleep.csv",     "Locking Disabled, High CPU Load, No Swap"),
    ("../../data/Locking Disabled/4. Medium CPU Load, Swap/usleep.csv",      "Locking Disabled, Medium CPU Load, Swap"),
    ("../../data/Locking Disabled/5. High CPU Load, Swap/usleep.csv",        "Locking Disabled, High CPU Load, Swap")
]

column_name = "Latency/Jitter (ns)"

stats = {
    "Metric": [],
    "Label": [],
    "Value": []
}

for file, label in csv_files:
    if os.path.exists(file):
        df = pd.read_csv(file)

        if column_name not in df.columns:
            print(f"Warning: Column '{column_name}' not found in {file}. Available columns: {list(df.columns)}")
            continue

        values = df[column_name].dropna()
        if values.empty:
            print(f"Warning: Column '{column_name}' in {file} is empty after removing NaN values.")
            continue

        stats["Metric"].append(valid_metrics[metric_to_plot])
        stats["Label"].append(label)
        if metric_to_plot == "min":
            stats["Value"].append(values.min())
        elif metric_to_plot == "max":
            stats["Value"].append(values.max())
        elif metric_to_plot == "mean":
            stats["Value"].append(values.mean())
        elif metric_to_plot == "std":
            stats["Value"].append(values.std())
    else:
        print(f"Warning: File {file} not found.")

stats_df = pd.DataFrame(stats)

if stats_df.empty:
    print("Error: No valid data found. Ensure the column name is correct and files are properly formatted.")
else:
    fig, ax = plt.subplots(figsize=(16,4))
    ax.bar(stats_df["Label"], stats_df["Value"], color="black")
```

```python

    ax.set_xticklabels(stats_df["Label"], rotation=45, ha="right")
    ax.set_ylabel("Jitter (ns)")
    ax.set_title(f"{valid_metrics[metric_to_plot]} usleep()")

    plt.tight_layout()
    plt.show()
```

Listing 4: `barchart.py`

It's important to note that the plots which display the mean value for each experiment could be misleading: if there was a high degree of variance in the collected results, with positive & negative values, they could cancel each other out and result in a deceptively small mean.

4.1 Signal Handling

The experimental data collected for the signal handling metric surprised me, as it did not match my expected results. Since this benchmark measures the latency between sending a signal to a process and the process executing it in its signal handler function, I would expect the mean latency to increase as CPU & memory load were increased. As the CPU load increases, processes can be delayed in their execution due to scheduling, and processes may be preempted, causing higher latency. I would expect high memory consumption to have similar effects, especially when memory locking is disabled, as the process data may then be swapped out, which is extremely slow & costly.

However, as can be seen in the figures below, this wasn't really the case for my collected data. The variance in my charted results seem to just be artefacts of noise in the system and fluctuations in the experimental conditions, as they don't seem to follow any discernible pattern. The main reason why I think this may have happened is because of the `PREEMPT_RT` kernel patches that I installed, which turned the OS into a fully-preemptible RTS, resulting in more predictable response times, and better prioritisation of tasks; since the benchmark program runs with maximum priority, lower priority processes like my stresser scripts could get preempted in favour of the high priority benchmarking program, thus resulting in the benchmarking program not being majorly effected by the system load.

I found these results very surprising, but upon reflection, they make sense, and are indicative of the power of the Linux kernel for use in hard RTS applications when the `PREEMPT_RT` patches are applied.

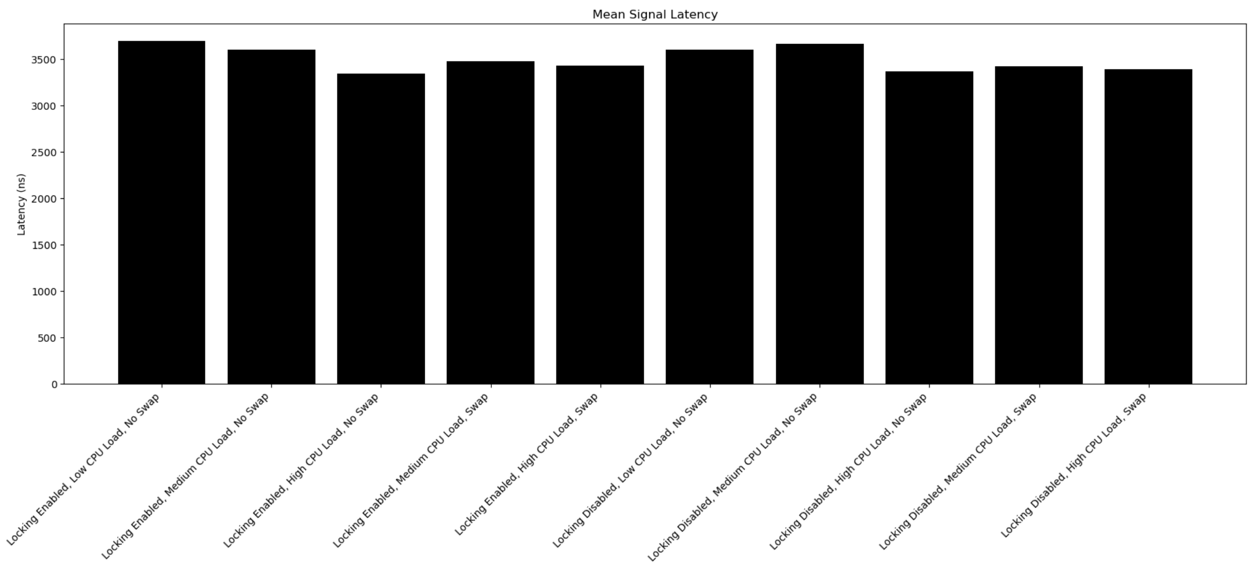


Figure 7: Mean latency for the signal handling benchmark

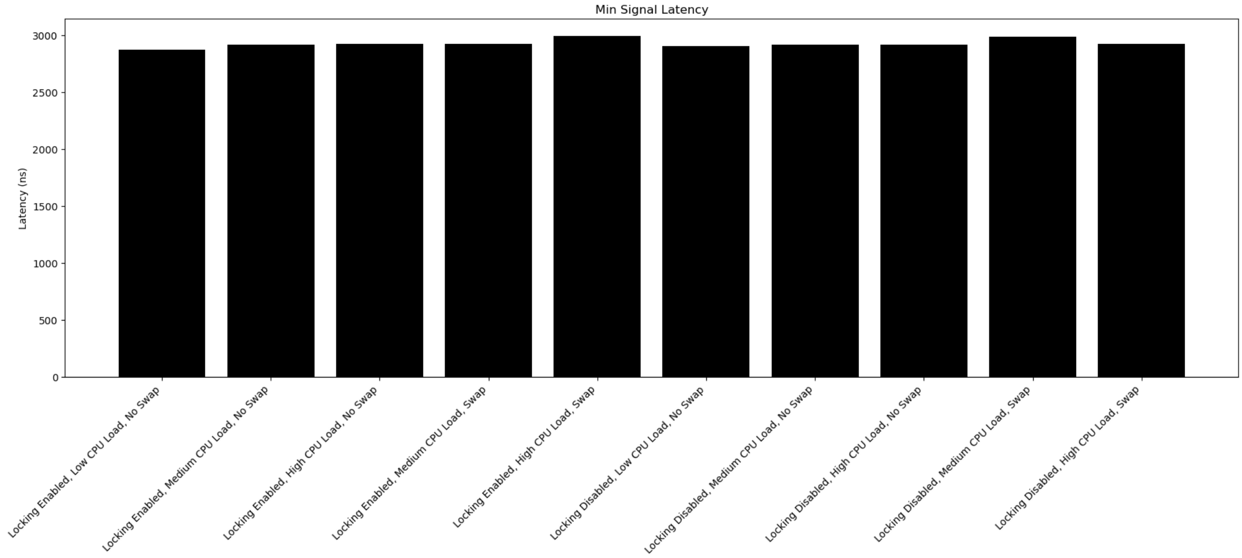


Figure 8: Minimum latency for the signal handling benchmark

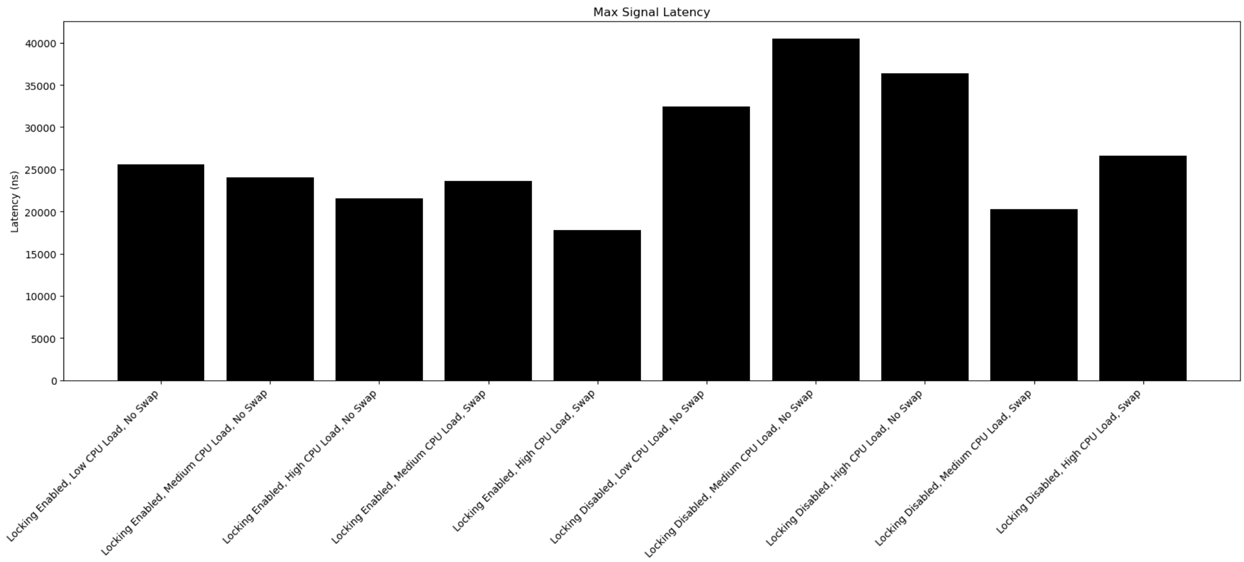


Figure 9: Maximum latency for the signal handling benchmark

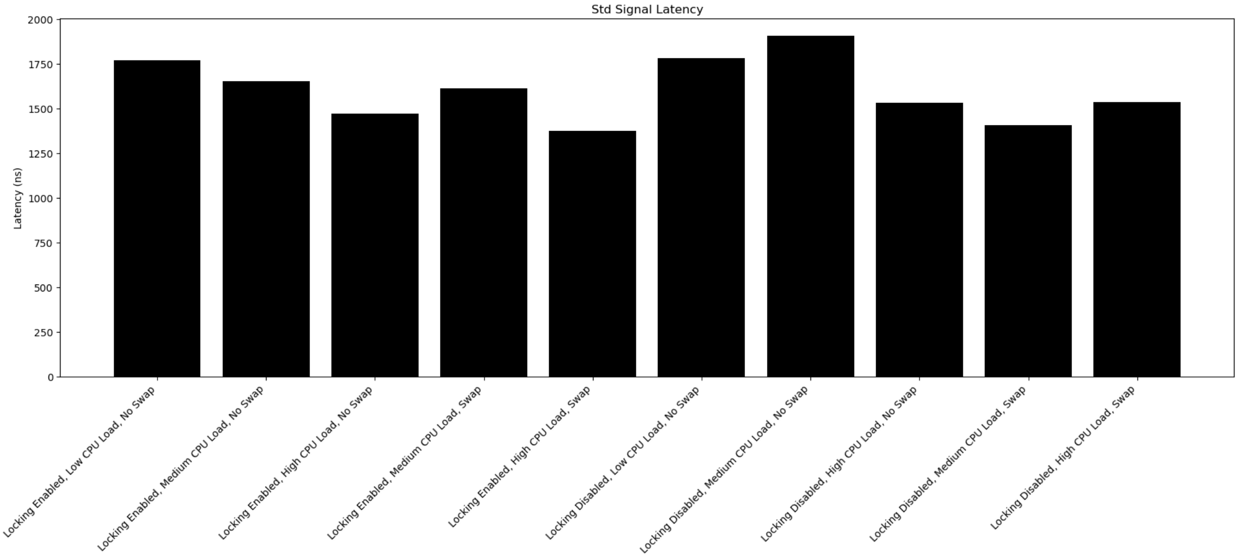


Figure 10: Standard deviation of latency for the signal handling benchmark

4.2 Interval Timer

Since the interval timer benchmark uses a POSIX interval timer to trigger a signal at precise intervals, I would expect the time interrupts to be precisely scheduled under low CPU load, and greater delay to appear under higher CPU load due to the CPU being busy. I would also expect swapping to worsen the jitter, as accessing the memory will be in the order of milliseconds rather than microseconds. However, as previously discussed, the `PREEMPT_RT` patches will help to mitigate these issues. We can see from the output data that, while not a clean trend upwards, there tends to be a higher jitter value for higher CPU loads. The most telling metric is the standard deviation; we can see from the standard deviation plot below that the variance in jitter trends upwards as CPU load & memory load increase, as one would expect.

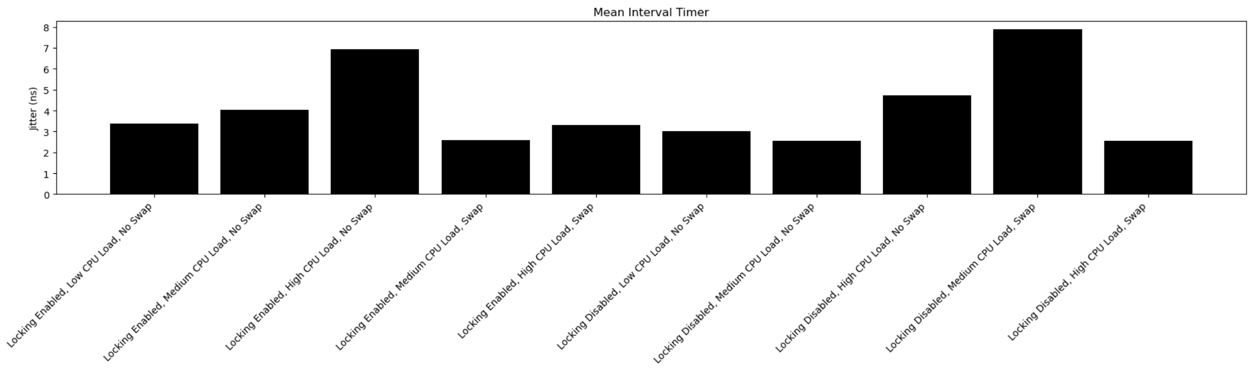


Figure 11: Mean jitter for the interval timer benchmark

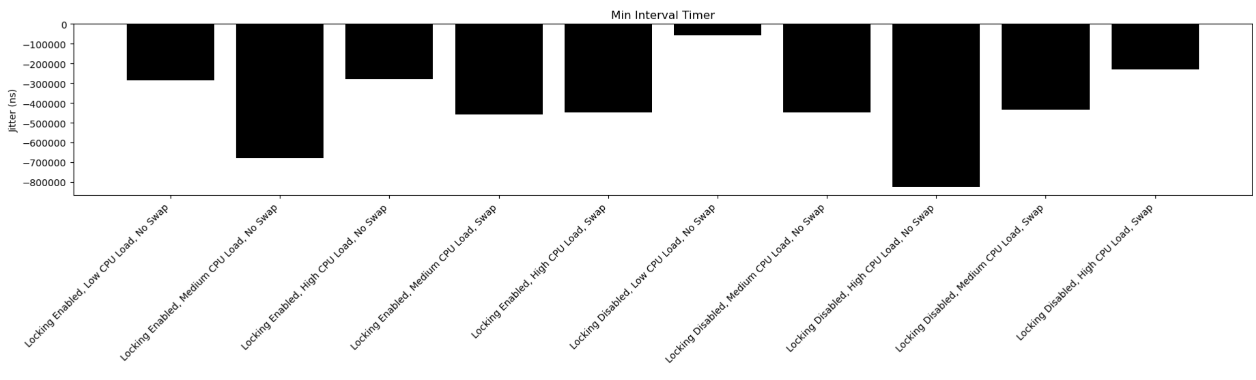


Figure 12: Minimum jitter for the interval timer benchmark

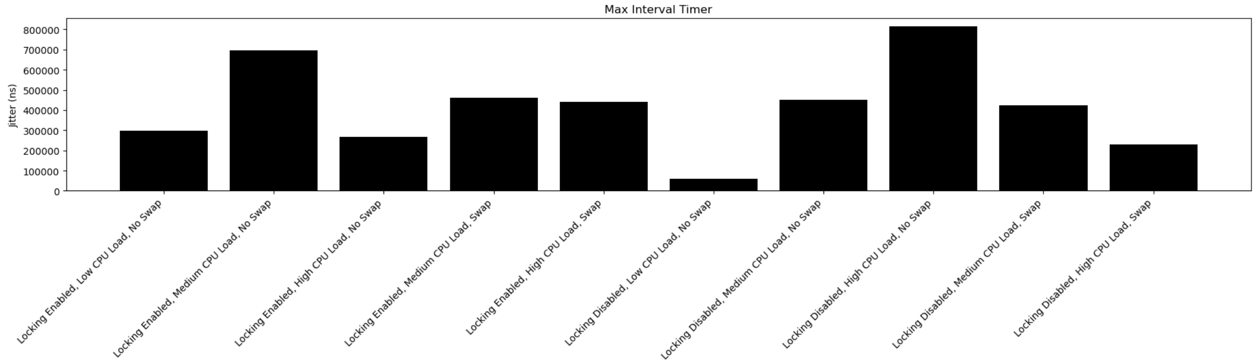


Figure 13: Maximum jitter for the interval timer benchmark

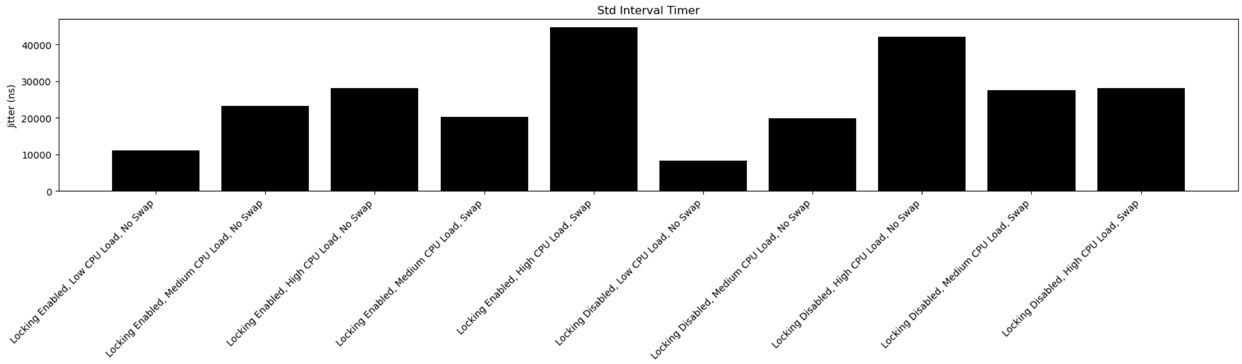


Figure 14: Standard deviation of jitter for the interval timer benchmark

4.3 `nanosleep()`

Since the `nanosleep()` benchmark measures the actual time elapsed versus the requested sleep duration, we would expect it to increase as the CPU load increases due to scheduling latency inducing jitter. Memory swapping adds large delays, and one would expect high CPU and high swap to cause erratic & unpredictable behaviour, making sleep times unreliable. The application of the `PREEMPT_RT` patches should increase the accuracy of sleep times, as the wake-ups will happen closer to the requested sleep duration and result in a lower maximum jitter value as the process can preempt other lower-priority tasks. The plotted charts don't bear a great deal of resemblance to the expected results, which is likely in large part due to the `PREEMPT_RT` patches, but also likely due to the large number of background tasks that are running at a given time on an Ubuntu system, which could be introducing noise into the data.

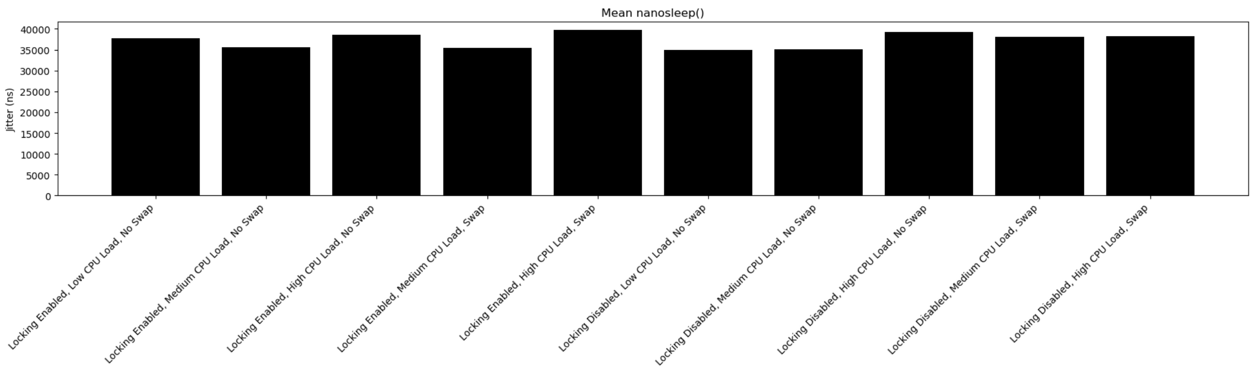


Figure 15: Mean jitter for the `nanosleep()` benchmark

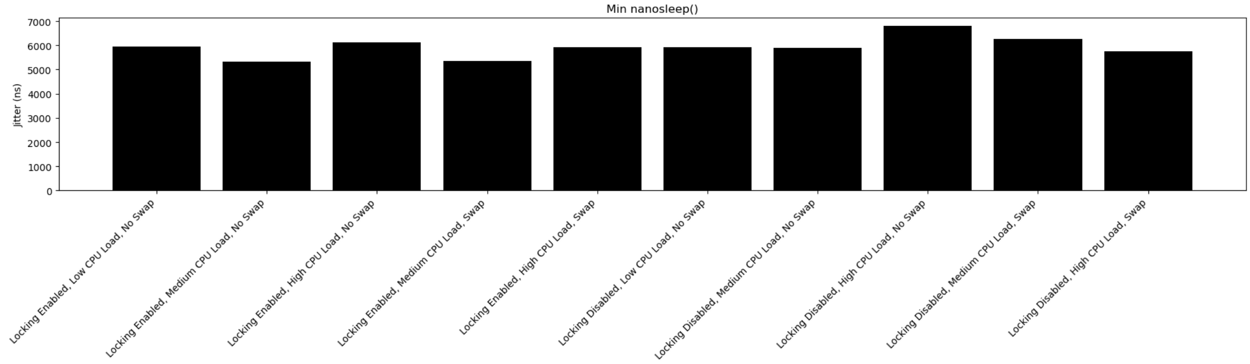


Figure 16: Minimum jitter for the `nanosleep()` benchmark

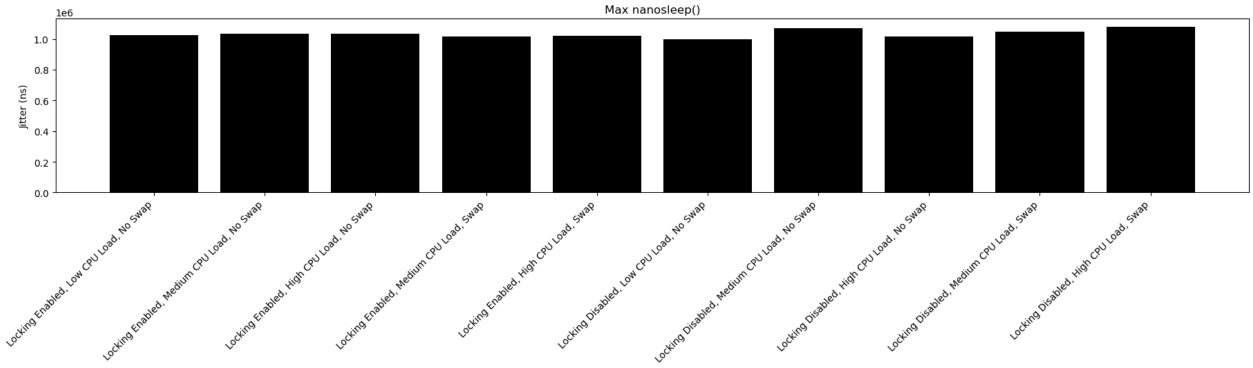


Figure 17: Maximum jitter for the `nanosleep()` benchmark

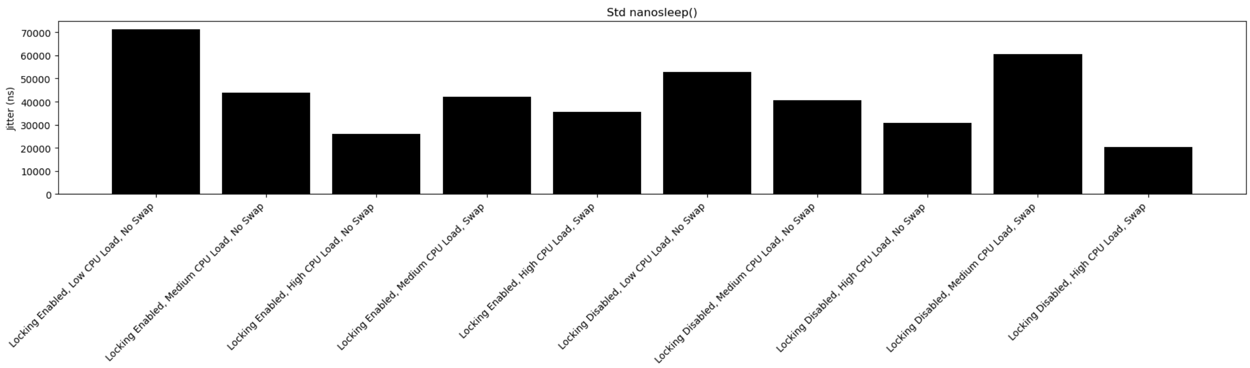


Figure 18: Standard deviation of jitter for the `nanosleep()` benchmark

4.4 `usleep()`

The `usleep()` function serves a similar role to `nanosleep()`, with the primary difference being that `usleep()` has precision in the microseconds (the `u` is an ASCII approximation of the μ symbol typically used to symbolise the “micro” prefix) rather than in the nanoseconds, and is thus far less precise. For this reason, greater jitter is to be expected. At low CPU usage, we would expect slightly worse performance than `nanosleep()`, and for this performance to decrease as CPU usage increases; similar behaviour is to be expected as memory usage increases also. Since `usleep()` relies on signals internally, it could potentially suffer more greatly under high CPU strain. The `PREEMPT_RT` patches can help to improve response times due to the preemptible kernel, but swapping will still cause performance issues.

The most interesting plot for this benchmark is the standard deviation plot below, as it corresponds pretty much exactly to what we would expect; clearly, `usleep()` derives less performance benefit from `PATCHES_RT` than `nanosleep()`. The jitter is lowest when locking is enabled, there is low CPU load, and no swap, and increases as the CPU load & memory load are increased. When locking is disabled, there is greater performance degradations between the low CPU/memory experiment and the subsequent experiments, with the high CPU, high memory, no locking experiment yielding the greatest standard deviation, and thus the least predictability.

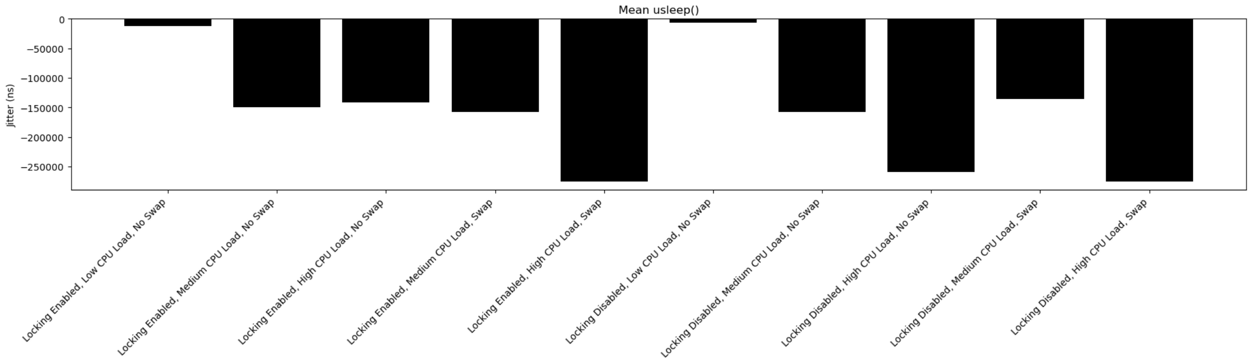


Figure 19: Mean jitter for the `usleep()` benchmark

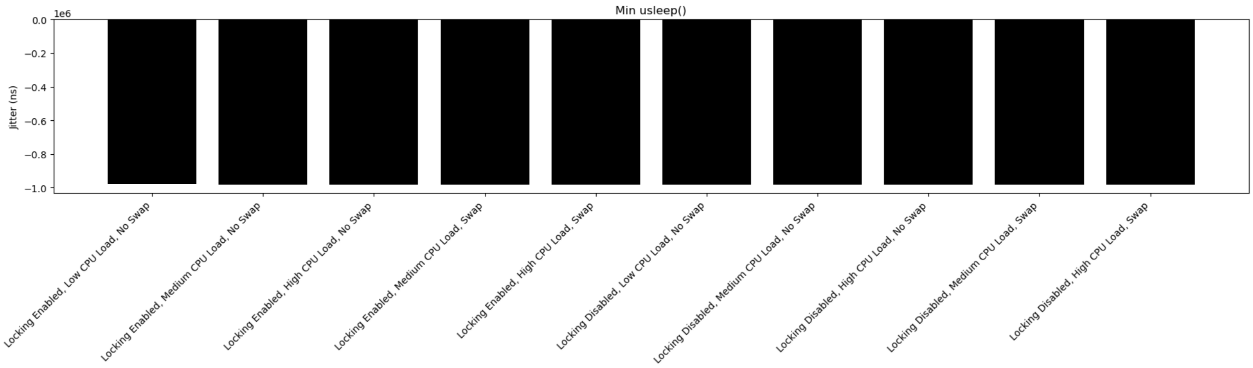


Figure 20: Minimum jitter for the `usleep()` benchmark

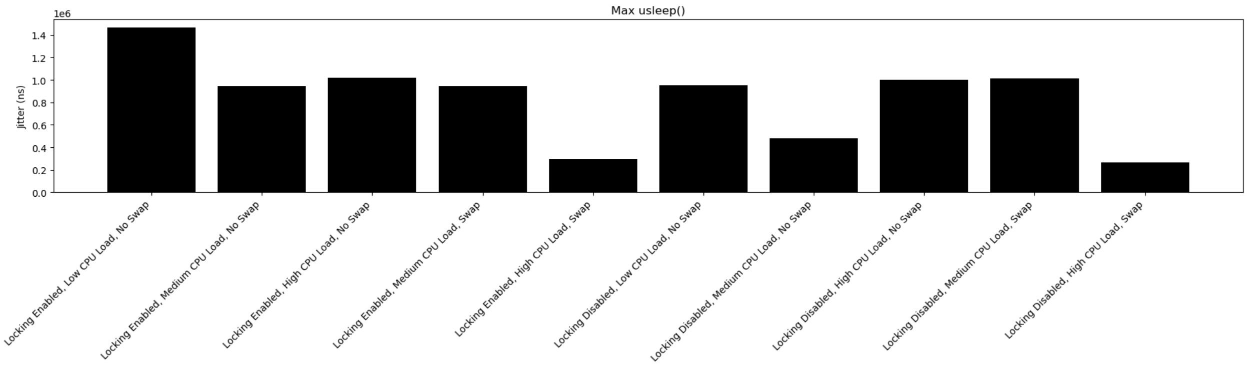


Figure 21: Maximum jitter for the `usleep()` benchmark

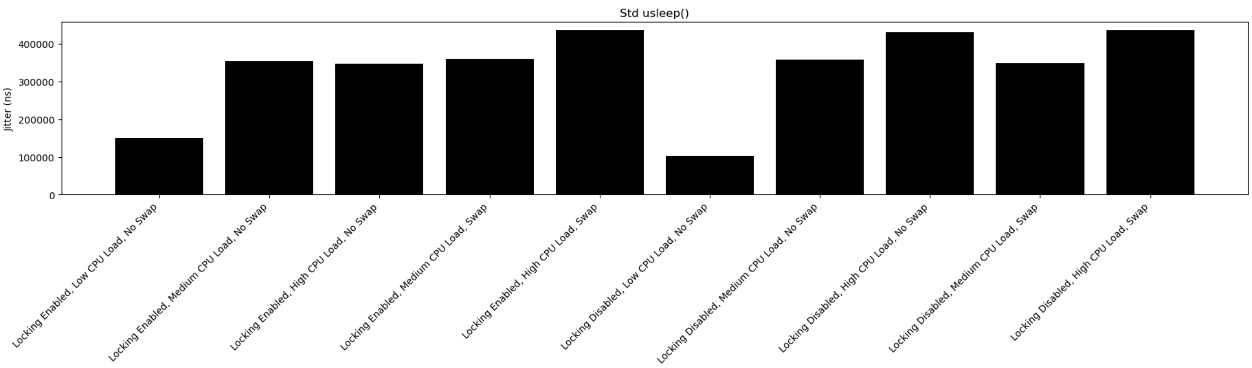


Figure 22: Standard deviation of jitter for the `usleep()` benchmark

5 Conclusions

To conclude, as CPU load & memory load increase, performance in terms of jitter & latency are to be expected to degrade. Memory locking helps to mitigate the negative effects of high memory consumption, by preventing the memory from being swapped. Using a fully preemptible kernel like the Linux kernel with the `PREEMPT_RT` patches applied can limit the negative effects of system strain, and help to ensure that deadlines are met, making such kernels a good choice for any kind of RTS, but particularly hard real-time systems.

References

[1] Canonical Group Ltd. *Basic Ubuntu Server Installation*. Accessed: 2025-03-18. 2025. URL: `https://documentation.ubuntu.com/server/tutorial/basic-installation/`.

[2] Hisham Muhammad. *htop(1)*. Accessed: 2025-03-18. 2025. URL: `https://www.man7.org/linux/man-pages/man1/htop.1.html`.

[3] Timo Rinne. *scp(1)*. Accessed: 2025-03-18. 2022. URL: `https://www.man7.org/linux/man-pages/man1/scp.1.html`.