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` 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;
}
```

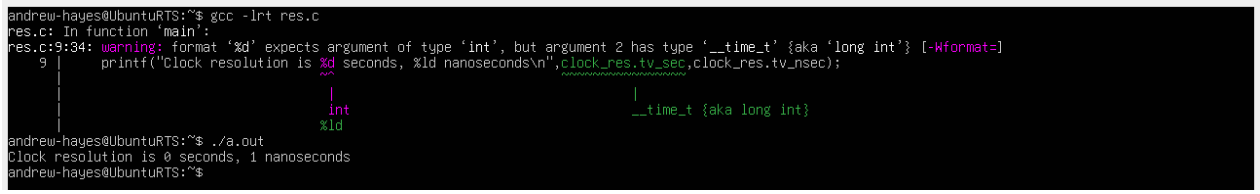

Figure 3: Getting the clock resolution of the virtual machine

2 CPU & Data-Intensive Applications

To develop my CPU & data-intensive programs, I chose to use Python for ease of development (and because any Python program will stress your CPU & memory no matter how simple 😉). I chose `htop` 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)

    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 1: `stress_cpu.py`

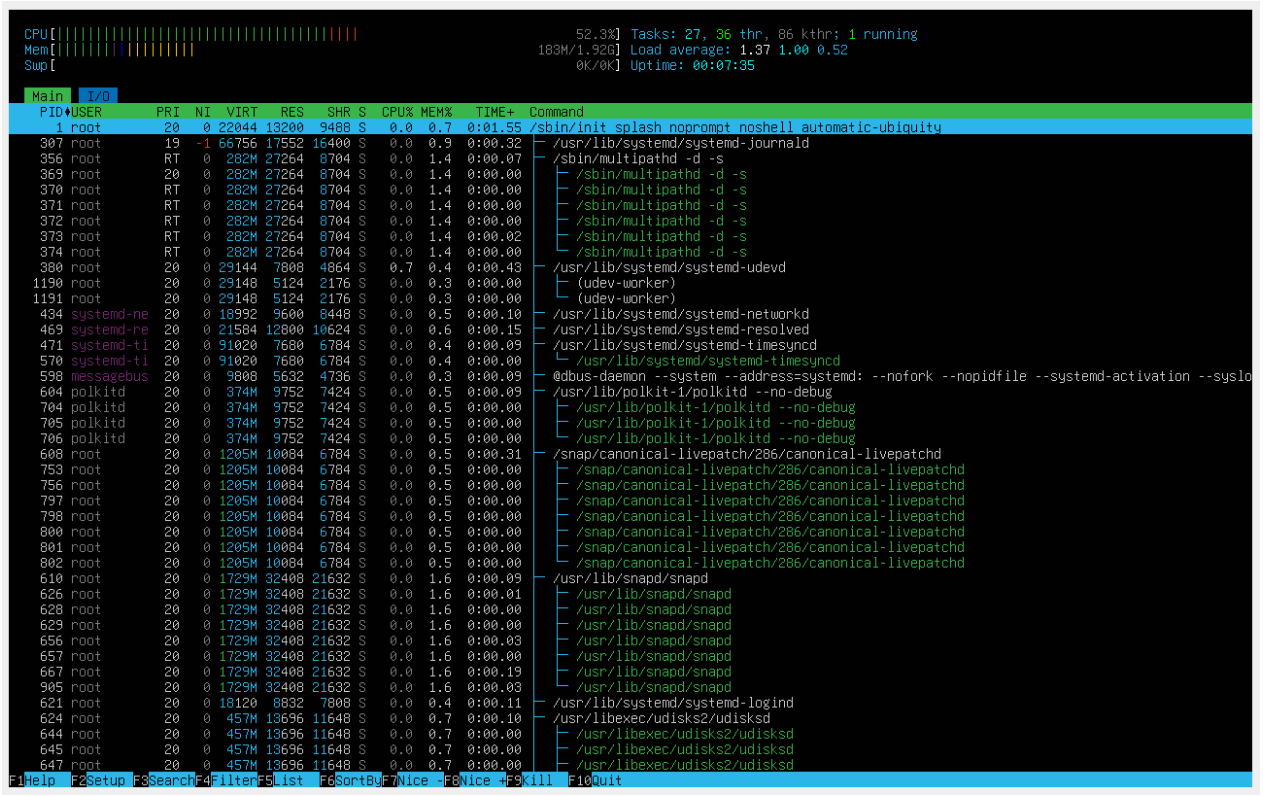

Figure 4: `htop` output when running `python3 stress_cpu.py --load medium`

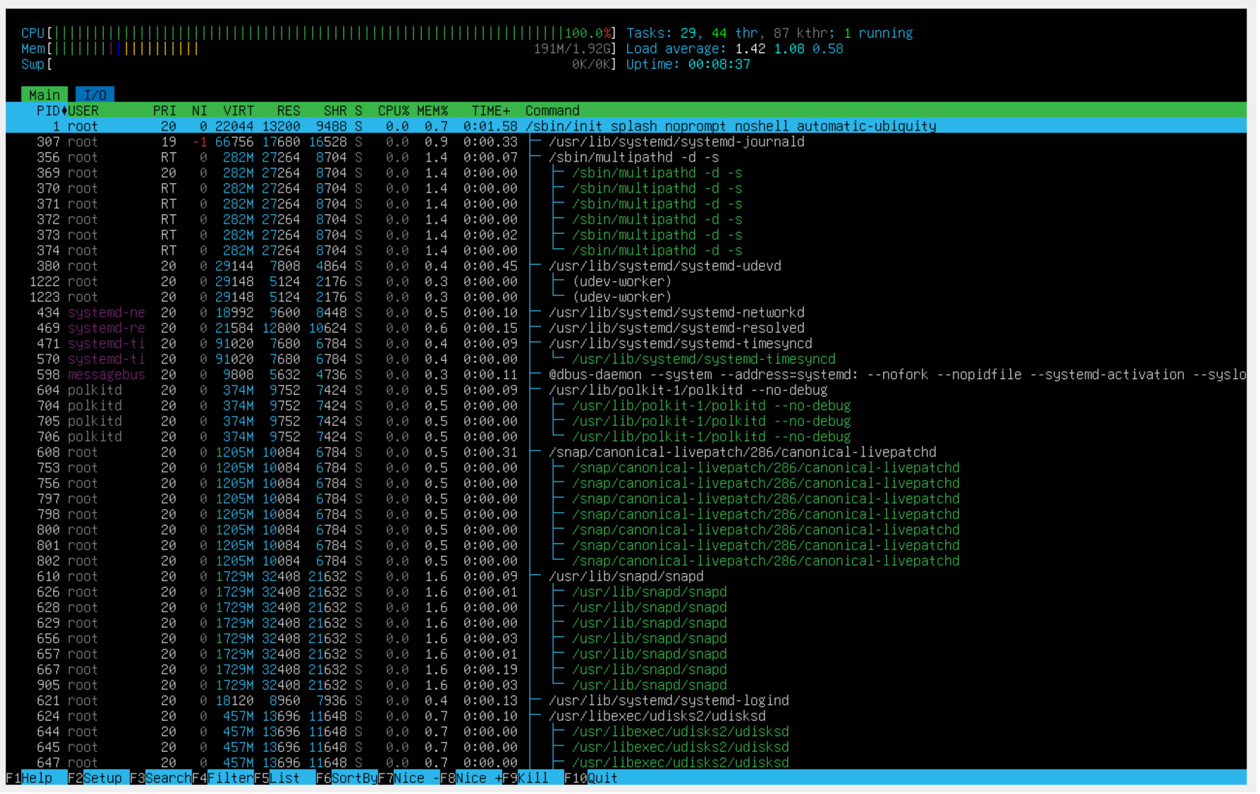

Figure 5: `htop` output when running `python3 stress_cpu.py --load high`

```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.")

```

```python
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 2: `stress_memory.py`

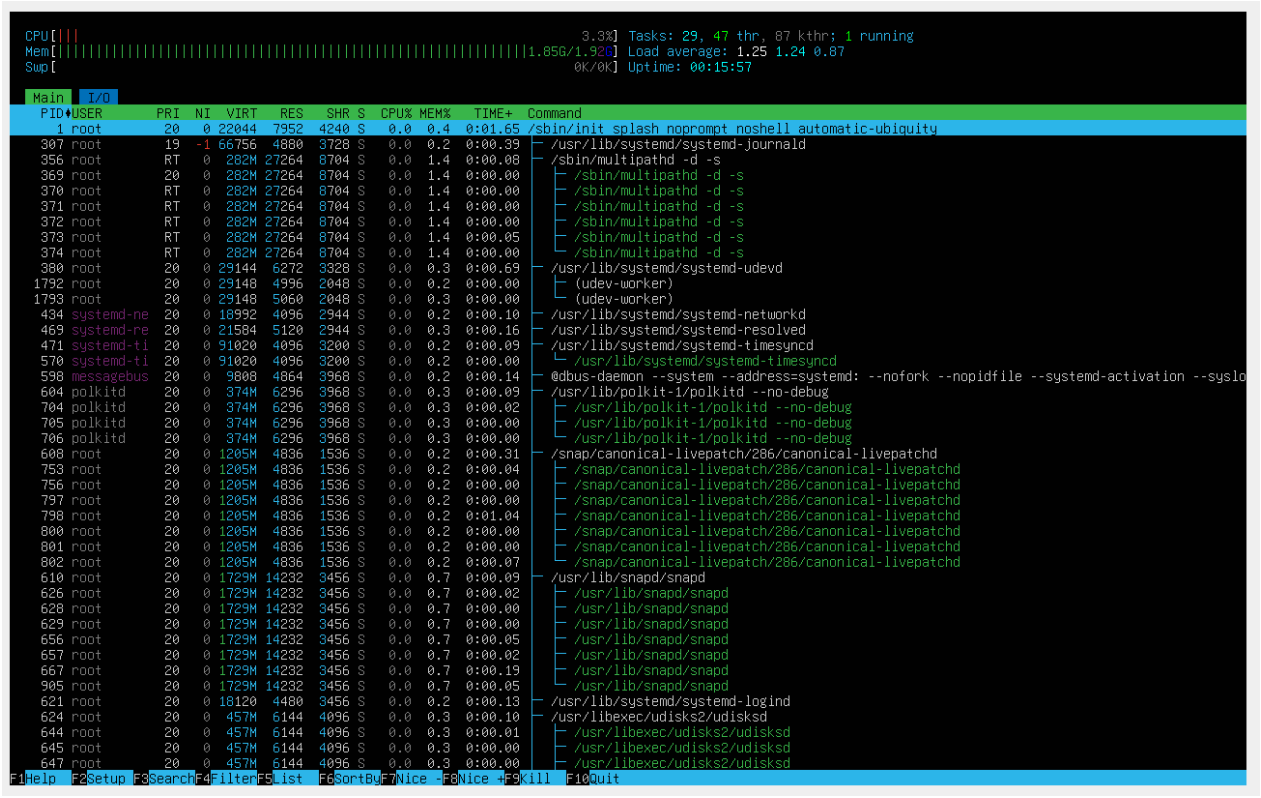

Figure 6: `htop` output when running `python3 stress_memory.py --usage 0.85`

References

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