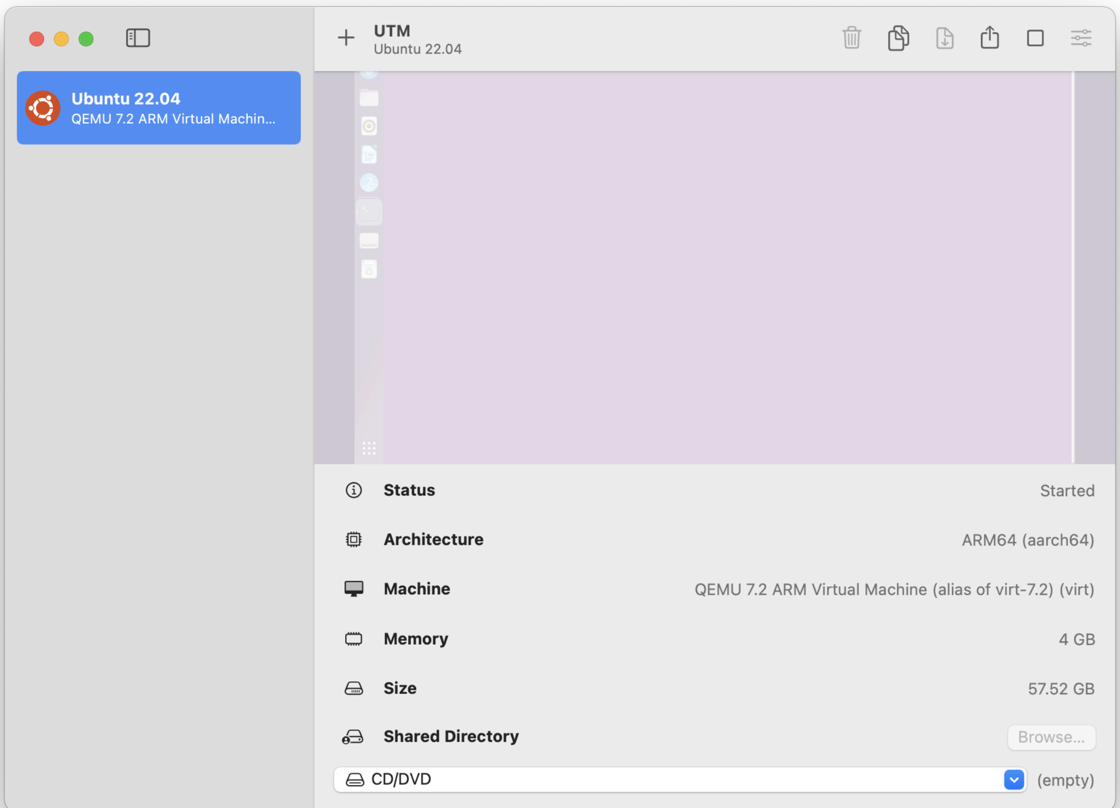
1.0 Kernel Setup

0.1 Hardware and OS of the Experiment Machine

- **Hardware**: UTM on Mac Pro 2023 M3



- **Operating System**: Ubuntu 22.04 LTS

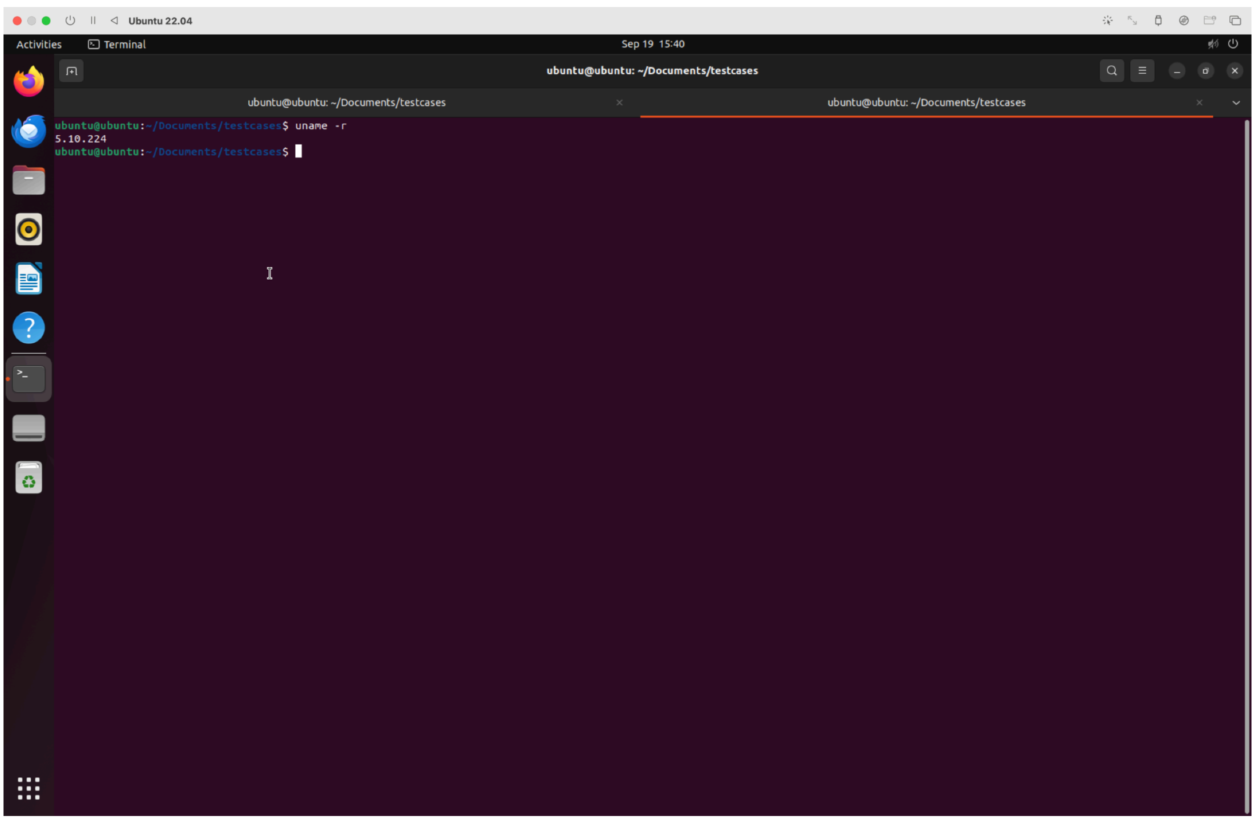
0.2 Steps to Set Up and Run the VM

I mainly followed the piazza post @15 to set up my machine. The command I used are the following.

sudo apt-get install libncurses5-dev gcc make git exuberant-ctags bc libssl-dev flex bison libelf-dev rsync
wget https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.10.224.tar.xz
tar xvf linux-5.10.224.tar.xz
cp /boot/config-`uname -r`* linux-5.10.224/.config
cd linux-5.10.224

```
scripts/config --disable SYSTEM_TRUSTED_KEYS
scripts/config --disable SYSTEM_REVOCATION_KEYS
scripts/config --disable SECURITY_LOCKDOWN_LSM
scripts/config --disable MODULE_SIG
scripts/config --disable MODULE_SIG_ALL
sudo apt install pahole
make -j$(nproc)
sudo make modules_install
sudo make install
sudo update-grub
sudo reboot
```

0.3 Screenshot of Running VM with Linux Kernel Version



0.4 Time to Complete the Setup

It took me 3 days to make the VM work. I spent two and a half days trying to configure it on the Google cloud machine, either through virsh or directly configure the OS of the compute engine. I did not make it work eventually and switched to local setup.

1.1 Understanding `ptrace`

In this task, I analyzed the kernel source code to understand how `PTRACE_PEEKDATA` and `PTRACE_POKEDATA` are handled by the kernel.

- **PTRACE_PEEKDATA**: This operation allows the tracer to read a word of data from the tracee's memory. The kernel function `generic_ptrace_peekdata` uses `ptrace_access_vm()` to read memory from the tracee's address space and then copies the data to the tracer's user-space buffer using `put_user()`.
- **PTRACE_POKEDATA**: This operation lets the tracer write data to the tracee's memory. The function `generic_ptrace_pokedata` writes the data at the specified address in the tracee's memory, again using `ptrace_access_vm()` with the `FOLL_WRITE` flag to force a write operation.
- **ptrace_access_vm()**: Both operations rely on this function to safely access the tracee's memory. It checks permissions and handles the memory read/write based on whether the operation is a peek or poke.

In summary, both operations enable the tracer to interact with the tracee's memory using kernel-level memory access routines while ensuring safety through validation and error handling.

1.2 Code Changes for Implementing Selective Memory Snapshotting

To implement the selective memory snapshotting feature, I made changes in three key files: `kernel/ptrace.c`, `include/linux/sched.h`, and `include/uapi/linux/ptrace.h`. Below is a summary of the major modifications:

2.1. `include/uapi/linux/ptrace.h`:

- **New `ptrace` Operations**: I defined three new `ptrace` request types:
  - `PTRACE_SNAPSHOT`: Used for taking a memory snapshot of a specific region in the tracee's memory.
  - `PTRACE_RESTORE`: Restores the memory of a tracee from a previously taken snapshot.
  - `PTRACE_GETSNAPSHOT`: Allows the tracer to retrieve a snapshot from the kernel space back to user space.

These definitions ensure the feature remains backward-compatible with the existing `ptrace` interface.

```
#define PTRACE_SNAPSHOT    0x5001
#define PTRACE_RESTORE     0x5002
```

```
#define PTRACE_GETSNAPSHOT 0x5003
```

2.2. `include/linux/sched.h`:

- **Snapshot Data Structure**: I extended the `task_struct` with a new field, `snapshot_list`, to store the list of snapshots associated with each tracee. This ensures each task (tracee) can store multiple snapshots, and the snapshots are properly maintained throughout the lifecycle of the tracee.

```
struct snapshot_list {
    struct list_head head;       // List head to store snapshots
};

struct task_struct {
    // Existing fields...
    struct snapshot_list snapshots; // List of snapshots for this task
    size_t total_snapshot_size;     // Total memory consumed by
snapshots
};
```

2.3. `kernel/ptrace.c`:

- **Handling New `ptrace` Requests**: I modified the core `ptrace_request()` function to handle the new requests: `PTRACE_SNAPSHOT`, `PTRACE_RESTORE`, and `PTRACE_GETSNAPSHOT`. These requests interact with the tracee's memory regions by reading or writing kernel-managed snapshots.
    - **`PTRACE_SNAPSHOT`**: This operation checks if the memory region is valid and writable using the Virtual Memory Areas (VMAs). It allocates kernel space to store the snapshot and copies the tracee's memory data into the snapshot.
    - **`PTRACE_RESTORE`**: For this request, the snapshot data is written back to the tracee's memory. After restoring the memory region, the snapshot is deleted from the list.
    - **`PTRACE_GETSNAPSHOT`**: This operation copies the snapshot from the kernel space back to the tracer's user space buffer, enabling the tracer to read the saved memory region.
- **Snapshot Management**: I implemented snapshot creation and deletion logic. Snapshots are stored in a kernel-linked list (`snapshot_list`). Each snapshot is dynamically allocated with the required memory region and metadata (start address, length).

- **Clean-up on Detach/Exit**: I added logic to clean up all snapshots when the tracer detaches from the tracee (`PTRACE_DETACH`) or the tracee exits. This ensures there are no memory leaks from leftover snapshots.

```
case PTRACE_SNAPSHOT:
    // Take a snapshot of a specified memory region
    ...
    break;

case PTRACE_RESTORE:
    // Restore the memory region from the snapshot
    ...
    break;

case PTRACE_GETSNAPSHOT:
    // Retrieve the snapshot data to user space
    ...
    break;

// Clean up snapshots during detach and exit
void __ptrace_unlink(struct task_struct *child) {
    // Delete all snapshots from the tracee
    ...
}
```

2.4 Summary:

The changes introduced support selective memory snapshotting for tracee processes, enabling the tracer to capture, restore, and inspect specific memory regions during debugging or tracing sessions. These changes were carefully integrated into the existing `ptrace` system without altering the standard `ptrace` interface, ensuring compatibility with existing functionality. The snapshot management system is robust, with proper memory allocation, access checks, and clean-up mechanisms implemented.

1.3 Test Screenshots