COL331 Assignment 1

Anubhav Pandey
2022CS51136

March 5, 2025

Table of Contents

Introduction

This assignment focuses on implementing resource tracking mechanisms within the Linux kernel, using versions `6.1.6` and `6.13.4` as the base environments. Initially, all tasks were developed and tested on kernel version `6.1.6`. However, since this version lacks an explicit syscall_64.tbl for ARM64 architecture, additional testing was conducted on the latest stable kernel version, `6.13.4`. Throughout this report, differences in implementation between the two kernel versions are highlighted, although these variations are minimal and do not affect functionality.

The implementation successfully operates on both kernel versions, ensuring compatibility and correctness. Note that code snippets included in this report are illustrative rather than exhaustive, showcasing key portions of the implementation.

1 Installing Linux

1.1 Attempting Dual Boot on a Windows Machine

The initial approach involved setting up QEMU on my Asus VivoBook 15, running a dual-boot configuration with Windows. I installed Ubuntu Server 24.04.2 LTS and partitioned a 50GB disk for this setup. Following the instructions provided in the problem statement, I attempted to compile the Linux kernel. However, the process was extremely slow, likely due to the limited computational resources available. With six cores allocated to the virtual machine, kernel compilation took approximately 4–5 hours. Given these constraints, it became evident that this approach was impractical for actual kernel development.

1.2 Attempting Installation on a Mac

Although Prof. Sarangi advised against using M-series Macs for this task, mentioning that he was not aware that anyone hadsuccessfully done it before, I had no alternative. My M3 MacBook Air was apparently the only powerful machine available to me.

The challenge was to determine an efficient way to set up the environment. After fairly extensive research, I discovered that cross-compilation using UTM, a hypervisor for macOS, was a possible solution. I installed QEMU, set up Ubuntu Server 24.04.2 LTS, and proceeded with kernel compilation.
However, even with 8 cores allocated, the process still took 2–2.5 hours, which was not a significant improvement.

Luckily, after a random realization that Apple was not the only company using ARM-based architecture, I investigated whether an ARM version of Ubuntu was available, because native compilation would be faster. Although there was no official ARM desktop release, I found an ARM version of Ubuntu Server. However, existing documentation for setting up a development environment on this platform was either sparse or unclear. After extensive experimentation and troubleshooting, I successfully set up my own workflow and documented the process.

My custom documentation, which I will include in the extra work section, outlines the detailed setup steps. Additionally, I figured out how to configure the ARM server to function similarly to a desktop environment, despite the lack of an official Ubuntu ARM desktop version!. This setup also allows for a more seamless coding experience using VS Code on the same machine.

The complete guide is available at the following link: My Installation Guide. This guide was also shared on Piazza, as instructed by Prof. Sarangi.

Key Takeaways from ARM Setup

- Native compilation on an ARM Ubuntu Server is significantly faster than emulated alternatives, a fresh compilation takes roughly 30-40 min.
- UTM provides a functional, if somewhat undocumented, path to kernel development on macOS.
- Cross-compilation was explored but found to be less practical compared to running an ARM-native setup.
- SSH/VS Code Remote Development enables a more seamless coding experience.
- The full installation guide includes detailed troubleshooting steps and optimizations.

Essential Facts for ARM Linux Kernel Setup

- Use UTM for virtualization: Apple Silicon requires an ARM-native hypervisor like UTM for optimal performance.
- Choose the right Ubuntu version: Only the ARM64 version of Ubuntu Server allows native kernel compilation.
- Allocate sufficient resources: At least 6-8 CPU cores and 8GB RAM are recommended for smooth compilation.
- Use make -j$(nproc): This command speeds up kernel compilation by using all available cores.
- Common UTM issues: Display problems can often be fixed by switching between ‘virtio-ramfb‘ and ‘virtio-gpu‘, it still may be "Display Not Active" state for 1-2 min at reboot if you haven’t given enough resources .

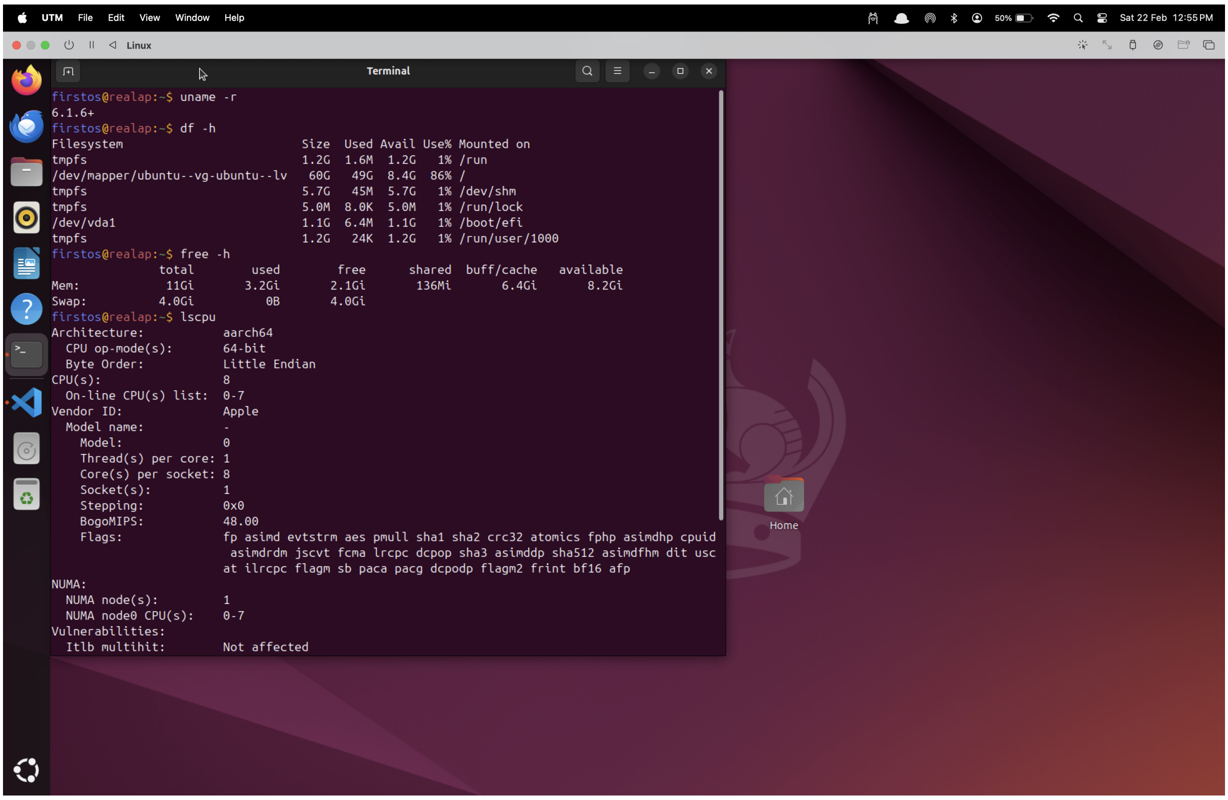

Figure 1: Screenshot of Ubuntu Desktop, with compiled into custom Linux kernel, with CPU and Memory Configuration

2 System Calls

For adding system calls, ideas from kernel.org doc and Brennan's Tut were used, there were no explicit documentation for ARM.

For system to recognise syscalls we include it in syscall_64.tbl & syscall_32.tbl (for 6.13.4 only, 6.1.6 automatically recognises it from `__NR_SYSCALLS`). All the syscall numbers shown here are for 6.13.4 kernel version.

In `arch/arm64/tools/syscall_64.tbl`:

```
467  common  register            sys_register
468  common  fetch               sys_fetch
469  common  deregister          sys_deregister
470  common  resource_cap        sys_resource_cap
471  common  resource_reset      sys_resource_reset
```

In `arch/arm64/tools/syscall_32.tbl`:

```
467  common  register            sys_register
468  common  fetch               sys_fetch
469  common  deregister          sys_deregister
470  common  resource_cap        sys_resource_cap
471  common  resource_reset      sys_resource_reset
```

In `include/uapi/asm-generic/unistd.h` (`__NR_SYSCALLS` was also updated):

```
#define __NR_sys_register         467
__SYSCALL(__NR_sys_register, sys_register)
#define __NR_sys_fetch            468
__SYSCALL(__NR_sys_fetch, sys_fetch)
#define __NR_sys_deregister       469
__SYSCALL(__NR_sys_deregister, sys_deregister)
#define __NR_sys_resource_cap     470
__SYSCALL(__NR_sys_resource_cap, sys_resource_cap)
#define __NR_sys_resource_reset   471
__SYSCALL(__NR_sys_resource_reset, sys_resource_reset)
```

Subsequently, the syscalls were integrated into `include/linux/syscalls.h`

```
asmlinkage long sys_register(pid_t pid);
asmlinkage long sys_fetch(...);
asmlinkage long sys_deregister(pid_t pid);
asmlinkage long sys_resource_cap(...);
asmlinkage long sys_resource_reset(pid_t pid);
```

Finally, the actual syscalls were implemented in a new file, `resource_tracker/resource_tracker.c`

```
SYSCALL_DEFINE1(register, pid_t, pid){...}
SYSCALL_DEFINE2(fetch,struct per_proc_resource __user *,stats,pid_t,pid){...}
SYSCALL_DEFINE1(deregister, pid_t, pid){...}
SYSCALL_DEFINE3(resource_cap,pid_t,pid,long,heap_quota,long,file_quota){...}
SYSCALL_DEFINE1(resource_reset, pid_t, pid){...}
```

And this file is added to the Makefile in the `resource_tracker` directory, and `resource_tracker` directory is added in the Kbuild file.

```
obj-y += resource_tracker.o resource_tracker_hooks.o ...
```

3 Resource Usage Tracker

Firstly we define per_proc_resource and pid_node structs in /include/linux/resource_tracker.h as asked in the problem statement, and declare following functions.

```c
void print_tracked_processes(struct seq_file *m);
void update_heap_usage(pid_t pid, long byte_change);
void update_openfile_count(pid_t pid, int change);
void cleanup_monitored_entry(pid_t pid);
```

3.1 `register`

The `register` syscall facilitates the registration of a process for resource monitoring. It first validates the provided PID and ensures the corresponding process exists using the `find_task_by_vpid` function. To maintain the validity of the task structure, the function increments the task's reference count before releasing the RCU read lock.

Once the process is verified, the syscall checks whether the PID is already being monitored. If not, it allocates memory for a new node using `kmalloc` and initializes the corresponding process resource structure. Initially, heap and file quotas are set to -1, indicating no restrictions. If the process is in a zombie or dead state, registration is aborted. Otherwise, the new node is appended to the monitored list.

```c
rcu_read_lock();
task = find_task_by_vpid(pid);
if (task)
    get_task_struct(task); // Increase reference count
rcu_read_unlock();
    ...
spin_lock(&monitored_lock);
    ...
node = kmalloc(sizeof(*node), GFP_KERNEL);
    ...
task_lock(task);
task->heap_quota = -1;
task->file_quota = -1;
task_unlock(task);

node->proc_resource->pid = pid;
node->proc_resource->heapsize = 0;
node->proc_resource->openfile_count = 0;

if (task->exit_state & EXIT_ZOMBIE || task->exit_state & EXIT_DEAD) {
    kfree(node->proc_resource);
    kfree(node);
    spin_unlock(&monitored_lock);
    return -3;
}

list_add_tail(&node->next_prev_list, &monitored_list);
put_task_struct(task); //decrease reference count
spin_unlock(&monitored_lock);
    ...
```

Synchronization and Safety

To ensure safe concurrent modifications of the monitored list, spinlocks are employed. Since the monitored list is shared among multiple processes, synchronization is crucial to prevent race conditions. Additionally, reference counting is utilized to safeguard the task structure from being prematurely freed by another process.

What if process is dead!!

Furthermore, before adding to list, the syscall verifies that the process is not in an exit state (zombie or dead), if dead returning -3 was chosen assuming the task is dead already and we do not monitor it. This design choice aligns with an additional mechanism implemented to automatically clean up the monitored list when a process exits via `do_exit`. This prevents a scenario where a newly assigned PID, corresponding to a different process, remains incorrectly associated with the previous process's monitoring entry, thereby preserving the integrity of the monitored list.

GFP_KERNEL

The syscall employs `GFP_KERNEL` as the allocation flag for `kmalloc`, which allows blocking memory allocations when required. Since this syscall runs in process context (rather than interrupt context), `GFP_KERNEL` is appropriate, as it permits sleeping while waiting for memory to be allocated. This ensures that the allocation is reliable and efficient.

3.2 `fetch`

The `fetch` syscall returns the current resource usage of a monitored process to user space. After verifying the PID and finding the corresponding node in the monitored list, it copies the resource usage data into user space memory.

```c
spin_lock(&monitored_lock);
list_for_each_entry(node, &monitored_list, next_prev_list) {
    if (node->proc_resource->pid == pid) {
        found = 1;
        break;
    }
}
spin_unlock(&monitored_lock);
   ...
struct per_proc_resource kernel_stats;
   ...
if (copy_to_user(stats, &kernel_stats, sizeof(struct per_proc_resource)))
    return -EFAULT;
```

copy_to_user

User doesn't have access to kernel space struct we defined, so we need to it copy it to the user space.

3.3 `deregister`

The `deregister` syscall removes a process from the monitored list. It searches for the given PID and, if found, removes it from the list, frees the allocated memory, and resets the heap and file quotas to -1. `kfree` is used to de-allocate the memory.

```c
spin_lock(&monitored_lock);
list_for_each_entry_safe(node, tmp, &monitored_list, next_prev_list) {
    if (node->proc_resource->pid == pid) {
        list_del(&node->next_prev_list);
        kfree(node->proc_resource);
        kfree(node);
        ...
        if(task){
            task_lock(task);
            task->heap_quota = -1;
            task->file_quota = -1;
            task_unlock(task);
        }
        break;
```

```c
    }
}
spin_unlock(&monitored_lock);
```

4 Resource Usage Limiter

For the resource limiter, firstly `heap_quota` and `file_quota` were added in the `task_struct`.

```c
unsigned long heap_quota;
unsigned long file_quota;
```

Note that the design choice of changing the types from `long` to `unsigned long` for these two entries was made because in `proc_resource` we are storing bytes of memory, which can range from 0 to $2^{64}-1$, but `long` would restrict `heap_quota` to $2^{32}-1$. Then we define following syscalls.

4.1 `resource_cap`

The `resource_cap` syscall sets resource quotas for a monitored process. It ensures the PID is being monitored and that quotas have not already been set. The function updates the heap and file quotas and calls update functions to reflect the changes.

```c
rcu_read_lock();
task = find_task_by_vpid(pid);
if (task)
    get_task_struct(task);
rcu_read_unlock();
    ...
spin_lock(&monitored_lock);
list_for_each_entry(node, &monitored_list, next_prev_list) {
    if (node->proc_resource->pid == pid)
        break;
}
if (!node) {
    spin_unlock(&monitored_lock);
    return -22;
}
spin_unlock(&monitored_lock);
    ...
task_lock(task);
task->heap_quota = heap_quota;
task->file_quota = file_quota;
task_unlock(task);
update_heap_usage(task->pid, 0);
update_openfile_count(task->pid, 0);
put_task_struct(task);
```

What happens if quota is long and not unsigned long?

We start observing process sometime after it being spawn, now suppose its acquires some heap, and then we register it, and set its resource cap to some value say 5MB. Now, after this if the process frees the previously aquired memory, since proc_resource.heapsize if of type unsigned long, it becomes very large number, but our heap quota is 5, so our program will kill that process, although this case was handled by setting heapsize to min(heapsize, heapsize+change) if change ¡ 0, but this gives the idea, that its more logical to set quotas to largest number that can be heapsize take. Also note that unsigned long (-1) is automatically $2^{64}-1$, so any process, with -1 heap quota will essentially have no limit.

Quota Enforcement and Updating list functions

update_heap_usage & update_openfile_count are functions which are engine of whole implementation. They check the current status of openfile count and heap usage, and if its beyond limits, then the process is killed using SIGKILL

4.2 resource_reset

The `resource_reset` syscall resets the resource quotas of a process back to -1 (no limit). It first ensures the process exists and is being monitored before modifying its quotas.

```
rcu_read_lock();
task = find_task_by_vpid(pid);
if (task)get_task_struct(task);
rcu_read_unlock();
    ...
spin_lock(&monitored_lock);
list_for_each_entry(node, &monitored_list, next_prev_list) {
    if (node->proc_resource->pid == pid)
        break;
}
if (!node) {...}
spin_unlock(&monitored_lock);
task_lock(task);
task->heap_quota = -1;
task->file_quota = -1;
task_unlock(task);
put_task_struct(task);
```

Ensuring Safe Reset

Since processes might still be consuming resources, resetting quotas without proper handling could lead to unexpected behavior. It would be semantically correct to reset the quota values back to -1.

5 Helper Functions: The Engine of Implementation

5.1 update_heap_usage

Defined in /resource_tracker/resource_tracker_hooks.c, it has a simple base case to not reduce heapsize below zero, because we are not supposed to track memory allocated before sys_register. And if heap quota is exceeded, KILL the program.

```
spin_lock(&monitored_lock);
list_for_each_entry(node, &monitored_list, next_prev_list) {
    if (node->proc_resource->pid == pid) {
        if(node->proc_resource->heapsize + byte_change >
           node->proc_resource->heapsize && byte_change < 0){
            node->proc_resource->heapsize = 0;
        }else{
            node->proc_resource->heapsize += byte_change;
        }
        ...
        spin_unlock(&monitored_lock);
        rcu_read_lock();
        task = find_task_by_vpid(pid);

        if (task && task->heap_quota != -1 &&
            node->proc_resource->heapsize >
               ((unsigned long) task->heap_quota * 1024 * 1024)) {
            ...
```

```
            rcu_read_unlock();
            send_sig(SIGKILL, task, 0);
            return;
        }
        rcu_read_unlock();
        return;
    }
}
spin_unlock(&monitored_lock);
```

5.2 update_openfile_count

Defined in /resource_tracker/resource_tracker_hooks.c, defined exactly how update_heap_usage is defined, only changing heapsize to openfile count.

```
if (task && task->file_quota != -1 &&
    (node->proc_resource->openfile_count >
     (unsigned long) task->file_quota)) {
       node->proc_resource->openfile_count, task->file_quota);
    rcu_read_unlock();
    send_sig(SIGKILL, task, 0);
    return;
}
```

5.3 cleanup_monitored_entry

Defined in /resource_tracker/resource_tracker.c The cleanup function removes a monitored process entry when the process terminates.

```
spin_lock(&monitored_lock);
list_for_each_entry_safe(node, tmp, &monitored_list, next_prev_list) {
    if (node->proc_resource->pid == pid) {
        list_del(&node->next_prev_list);
        kfree(node->proc_resource);
        kfree(node);
        break;
    }
}
spin_unlock(&monitored_lock);
```

Preventing stale entries

Number of unique PID that can be allocated is limited, stored in `/proc/sys/kernel/pid_max`, which means that a specific PID should be properly removed from our monitored list when the process dies.

6 Updating Syscalls

The problem statement asks to monitor the heap memory allocated through brk and mmap with `MAP_ANONYMOUS` or `MAP_PRIVATE` flags and files opened through open, openat, openat2. We update the syscalls to call helper functions we defined.

6.1 mmap

Defined in arch/arm64/kernel/sys.c, we only check if bits of `MAP_ANONYMOUS` or `MAP_PRIVATE` are active, if yes, then update the list.

```
SYSCALL_DEFINE6(mmap, ...){
    ... //existing
```

```
    long retval =  ksys_mmap_pgoff(addr, len, prot, flags, fd, off >>
       PAGE_SHIFT);

    if(retval != MAP_FAILED && (flags & MAP_ANONYMOUS) && (flags & MAP_PRIVATE)){
        update_heap_usage(current->pid, len);
    }
    return retval;
}
```

6.2 brk

Defined at mm/mmap.c, as I worked with both versions of linux kernel, there was slight difference in the way brk syscall was defined, so following is for v6.13.4, although semantically both versions boil down equivalent.

```
SYSCALL_DEFINE1(brk, unsigned long, brk)
{       ...   //existing code
    if (brk <= mm->brk) { //update if memory shrinks
        ... //existing code
        update_heap_usage(current->pid, newbrk - oldbrk);
        goto success_unlocked;
    }
    ... //existing code
    mm->brk = brk;
    update_heap_usage(current->pid, newbrk - oldbrk);
    if (mm->def_flags & VM_LOCKED)
        populate = true;
    ... //existing code
}
```

6.3 open, openat, openat2

These syscalls are defined in fs/open.c All the three functions eventually call `do_sys_openat2` function, so on sucessfull opening of file update_openfile_count function is called

```
static long do_sys_openat2(int dfd, const char __user *filename,struct open_how
   *how){
        ... //existing code
    if(fd >=0 ){
        update_openfile_count(current->pid, 1); // all 3 syscalls open, openat,
           openat2 call this functin
    }
    return fd;
}
```

6.4 close

This goes to additional work, we also modify close syscall, to maintain the files closed also. close syscall is also defined in fs/open.c.

```
SYSCALL_DEFINE1(close, unsigned int, fd){
    ... //existing code
    retval = filp_flush(file, current->files);
    if (retval == 0) update_openfile_count(current->pid, -1);
    ... //existing code
```

7 Helper Modules

Till now everything works, so we go ahead and add following modules.

7.1 cleanup_kprobe module

This modules is to cleanup any registered process, whether it dies through SIGKILL, normal process completion, etc, though cleanup_monitored_entry function, as already mentioned previously that pool of unique PIDs is limited. Use of kprobe and do_exit can be read here. When a process dies, do_exit function is called.

```
static struct kprobe kp = {
    .symbol_name = "do_exit",
};

static int handler_pre(struct kprobe *p, struct pt_regs *regs)
{
    pid_t exiting_pid = current->pid;
    cleanup_monitored_entry(exiting_pid);
    return 0;
}

static int __init cleanup_kprobe_init(void)
{
    int ret;
    kp.pre_handler = handler_pre;
    ret = register_kprobe(&kp);
    ... //analyse ret and print if kprobe sucessfully registered
    return 0;
}

static void __exit cleanup_kprobe_exit(void)
{
    unregister_kprobe(&kp);
    pr_info("Cleanup kprobe unregistered\n");
}
```

Functionalities so far are exhaustive, that is, they are semantically correct. This ensures that only intended processes are tracked and that no registered process has stale or incorrect data at any given time.

7.2 tracker module

This module is to visualize the linked list of monitored process as a table using proc file system similar to top or htop.

```
static int tracker_proc_show(struct seq_file *m, void *v){
    print_tracked_processes(m);
    return 0;
}
static int tracker_proc_open(struct inode *inode, struct file *file)
{
    return single_open(file, tracker_proc_show, NULL);
}
static const struct proc_ops tracker_proc_ops = {
    .proc_open    = tracker_proc_open,
    .proc_read    = seq_read,
    .proc_lseek   = seq_lseek,
    .proc_release = single_release,
};
static int __init tracker_init(void){
    struct proc_dir_entry *entry;
    entry = proc_create("tracker_status", 0, NULL, &tracker_proc_ops);
    if (!entry) {...}
    ...
}
static void __exit tracker_exit(void)
{remove_proc_entry("tracker_status", NULL);...}
```

where print_tracked_processes safely reads through monitored list, and prints value in the proc/tracker_staus file using seq_printf.

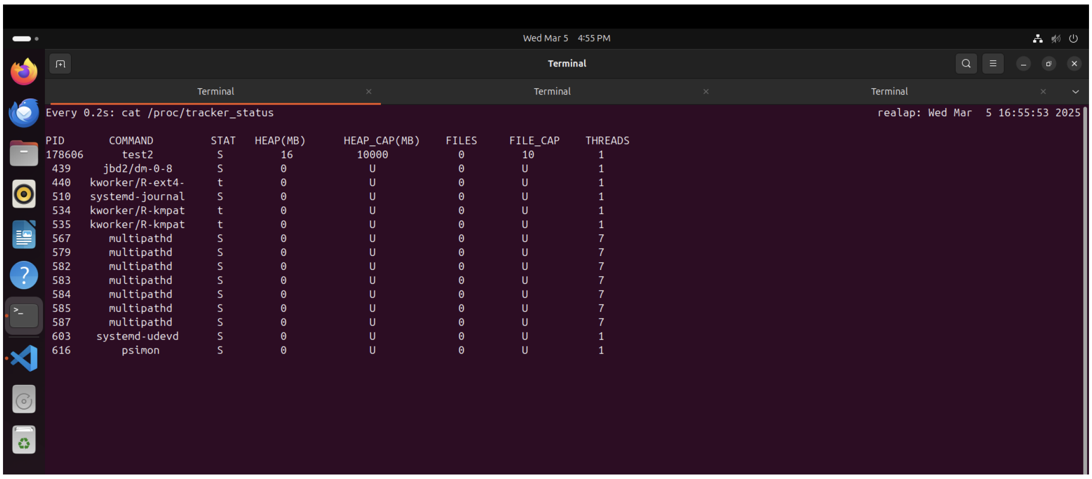

Figure 2: proc/tracker_status showing all the monitored processes, here U means No limit set

The Extra List

- Creating detailed documentation on setting up Linux Kernel development on macOS.
- Writing a module for cleaning up dead processes as soon as they terminate using ‘`kprobe`‘ for correctness.
- Implementing a module involving ‘`proc`‘ file system to observe the list of all tracked modules in real-time.
- Developing a resource tracker compatible with both Linux versions `v6.1.6` and `v6.13.4`.
- Some other design choices like datatype of heaplimit.
- Handling very important edge case of what happend if a process dies at the time of registration.

8 Submission Details

The diff file was generated by using following commands,

```
git add . //this stages all changes
git diff --staged > res_usage.patch
```

This automatically ignores all the executables ignored by .gitignore. This patch is compatible with the kernel v6.13.4 and can be applied, with the benefit of being much smaller than the patch generated by the command mentioned in the assignment.

The directory structure is as follows:

```
assignment1_hard_2022CS51136
├── res_usage.patch
├── res_usage_6_1_6.patch
├── report.pdf
└── modified_files
```

All the modified files are in modified_files folder. For starting the modules, go inside resource_tracker, and call `sudo make modules`. Note that final submission is with respect to linux v6.13.4.

Appendix

Directory strcture

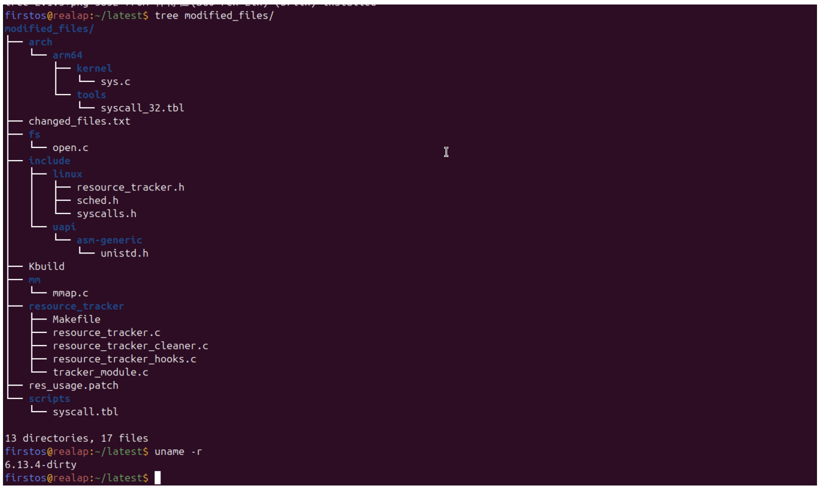

Figure 3: Tree structure of modified_files

Working v6.1.6

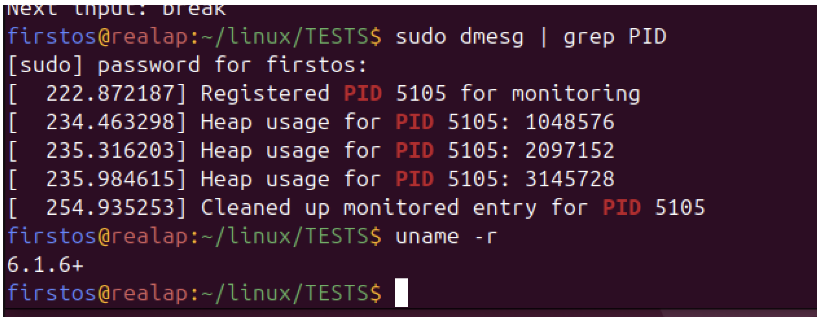

Figure 4: System calls noted using printk

Set of Bootable partitions

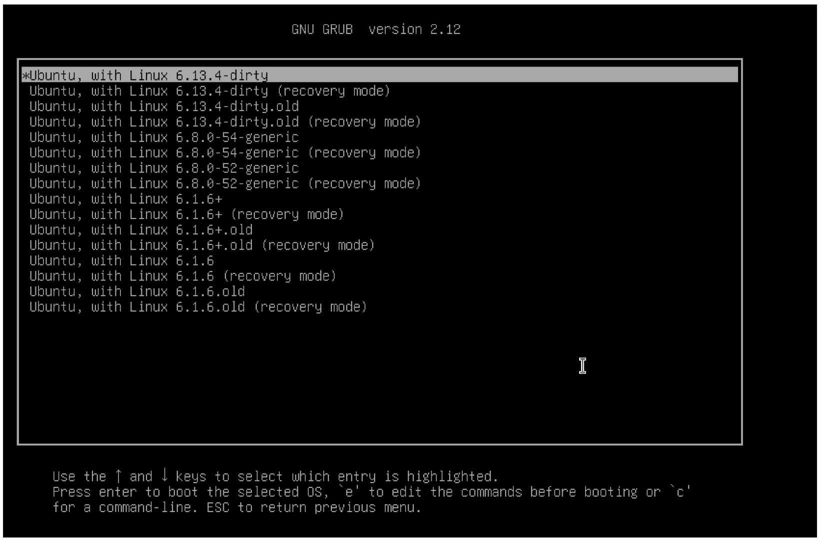

Figure 5: Set of Bootable partitions