COL331 A1

Sarthak Panda (2022CS51217)

March 2025

1 Introduction

The current implementation in total required modification of 10 files. Which we have added to patch files and are discussed in following sections.

File	Status
Kbuild	Modified
arch/x86/entry/syscalls/syscall_64.tbl	Modified
arch/x86/kernel/sys_x86_64.c	Modified
arch/x86/um/syscalls_64.c	Modified
fs/open.c	Modified
hello/Makefile	New File
hello/hello.c	New File
include/linux/hello.h	New File
include/linux/syscalls.h	Modified
mm/mmap.c	Modified

2 Data structure of the Monitoring List

```
#ifndef HELLO_H
#define HELLO_H
#include <linux/list.h>
#include <linux/types.h>
struct pid_node {
    struct per_proc_resource* proc_resource;
    struct list_head next_prev_list;
};
struct per_proc_resource {
    pid_t pid;
    unsigned long heapsize;
    unsigned long openfile_count;
    long heap_quota;
    long file_quota;
    bool quotas_defined;
};
extern struct list_head monitored_pids_head_node;
extern struct mutex monitored_list_mutex;
#endif
```

Listing 1: Kernel code snippet from hello/hello.h

I have updated the per_proc_resource node structure to handle both the resource usage tracker and limiter and added three more attributes (heap_quota, file_quota, and quotas_defined). Further, in the hello.h file, I kept the declaration of the monitored_pids_head_node, which is a global variable representing the sentinel node of the list to be monitored. It is subsequently initialized by the macro LIST_HEAD(monitored_pids_head_node); in hello.c. Similarly, monitored_list_mutex is a mutex lock declared in hello.h, which will be used while accessing the monitoring list (since multiple threads can access the list, it becomes essential to use locks). Again, it is initialized and defined in 'hello.c' by the macro DEFINE_MUTEX(monitored_list_mutex);.

3 Resource Usage Tracker

3.1 sys_register

```c
SYSCALL_DEFINE1(register, pid_t, pid) /* in hello/hello.c */
{
    struct task_struct *task;
    struct pid_node *node;
    struct list_head *pos;
    printk(KERN_INFO "Syscall sys_register called with pid: %d\n", pid);

    if (pid < 1) {
        printk(KERN_INFO "Syscall sys_register: PID %d is less than 1 \n", pid);
        return -22;
    }

    rcu_read_lock();
    task = find_task_by_vpid(pid);
    rcu_read_unlock();
    if (!task) {
        printk(KERN_INFO "Syscall sys_register: PID %d does not exist\n", pid);
        return -3;
    }

    mutex_lock(&monitored_list_mutex);
    list_for_each(pos, &monitored_pids_head_node) {
        node = list_entry(pos, struct pid_node, next_prev_list);
        if (node->proc_resource->pid == pid) {
            mutex_unlock(&monitored_list_mutex);
            printk(KERN_INFO "Syscall sys_register: PID %d is already monitored\n", pid);
            return -23;
        }
    }

    node = kmalloc(sizeof(struct pid_node), GFP_KERNEL);
    if (!node) {
        mutex_unlock(&monitored_list_mutex);
        printk(KERN_ERR "Syscall sys_register: Memory allocation failed for PID %d\n", pid);
        return -ENOMEM;
    }

    node->proc_resource = kmalloc(sizeof(struct per_proc_resource), GFP_KERNEL);
    if (!node->proc_resource) {
        kfree(node);
        mutex_unlock(&monitored_list_mutex);
        printk(KERN_ERR "Syscall sys_register: Memory allocation failed for PID %d\n", pid);
        return -ENOMEM;
    }

    node->proc_resource->pid = pid;
    unsigned long heapsize = 0;
    node->proc_resource->heapsize = 0;
    node->proc_resource->openfile_count = 0;
    node->proc_resource->heap_quota = -1;
    node->proc_resource->file_quota = -1;
    node->proc_resource->quotas_defined = false;

    list_add(&node->next_prev_list, &monitored_pids_head_node);
    mutex_unlock(&monitored_list_mutex);
    return 0;
}
```

Listing 2: Kernel code snippet for sys$_{r}egisterinhello/hello.c$

The first important check that sys_register needs to perform is to check if the pid is valid and there is an task_stuct corresponding to it. For which we use find_task_by_vpid (resource link) .This function is essentially is a wrapper that finds the process by its virtual PID (vpid) within the PID namespace of the current process. It calls find_task_by_pid_ns() with the provided vpid and the active PID namespace (obtained via task_active_pid_ns(current)), ensuring that the lookup is within the current namespace boundaries.

```
struct task_struct *find_task_by_vpid(pid_t vnr)/*in kernel/pid.c*/
{
    return find_task_by_pid_ns(vnr, task_active_pid_ns(current));
}
```

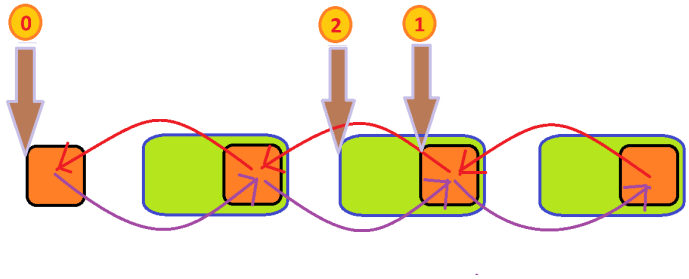
Listing 3: Kernel code snippet for find_task_by_vpid

Further, it is important to note that rcu_read_lock() is essential before accessing the task_struct, because otherwise it can possibly be concurrently updated or deleted by writer threads. Unlike mutexes, which serialize access and block other threads (or CPUs) from entering the critical section, rcu_read_lock() is very lightweight and allows multiple readers to run concurrently. It doesn't enforce exclusive access; instead, it ensures that any memory being read won't be reclaimed until all readers exit their critical sections. In read-mostly scenarios (which are common in kernel code), rcu_read_lock() offers far lower overhead than a mutex.

```
/**
 * list_for_each    -   iterate over a list
 * @pos:    the &struct list_head to use as a loop cursor.
 * @head:   the head for your list.
 */
#define list_for_each(pos, head) \
    for (pos = (head)->next; !list_is_head(pos, (head)); pos = pos->next)
```

Listing 4: Kernel code snippet for list_for_each in /include/linux/list.h

Now next important thing is before we iterate over the monitoring list, we must lock the earlier defined lock as multiple threads simultaneously updating the list may lead to inconsistent state. Further, one essential thing to note is that list_for_each in list.h actually starts from actual head and not sentinel node [arrow 0 denotes it in below figure] that it takes as argument (this is because if we see in Listing 3, pos starts from (head)->next).



Next list_entry is used to get the pointer to starting location of actual node/container of list head (i.e. it shifts the pointer from 1 to 2).

```
/**
 * list_entry - get the struct for this entry
 * @ptr:    the &struct list_head pointer.
 * @type:   the type of the struct this is embedded in.
 * @member: the name of the list_head within the struct.
 */
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

Listing 5: Kernel code snippet for list_entry in /include/linux/list.h

Now if pid already exists then return -23, else we move on allocate space in kernel space using kmalloc, The GFP_KERNEL flag is used to get free pages in kernel space.(As mentioned in link here it should be our first choice when trying to allocate in kernel space.). Next if allocations are successful we set the node parameters and add it to list using list_add(in list.h) then unlock mutex and return.

3.2 sys_fetch

```
SYSCALL_DEFINE2(fetch, struct per_proc_resource __user *, stats, pid_t, pid)
{
    struct pid_node *node;
    struct list_head *pos;
    struct per_proc_resource kstats;
    if (!stats || !access_ok(stats, sizeof(struct per_proc_resource))) {
        return -22;
    }
    mutex_lock(&monitored_list_mutex);
    list_for_each(pos, &monitored_pids_head_node) {
        node = list_entry(pos, struct pid_node, next_prev_list);
        if (node->proc_resource->pid == pid) {
            kstats = *node->proc_resource;
            kstats.heapsize = kstats.heapsize / (1024 * 1024);
            mutex_unlock(&monitored_list_mutex);
            if (copy_to_user(stats, &kstats, sizeof(struct per_proc_resource))) {
                printk(KERN_ERR "sys_fetch: copy_to_user failed for pid %d\n", pid);
                return -22;
            }
            return 0;
        }
    }
    mutex_unlock(&monitored_list_mutex);
    return -22;
}
```

Listing 6: Kernel code snippet for sys_fetch in /hello/hello.c

In the start of sys_fetch we sanity checks the "stats" which is pointer to the user space if that is not null and can be accessed. Further similar to the sys_register we lock the list and then traverse if we find the provided pid in the monitoring list we simply copy that to user space using copy_to_user (in /include/linux/uaccess.h) function (and internally since we are storing the heap-size in bytes we convert it into MB before copying it to location pointed by stats). If fetch went successful we return 0 otherwise in all other cases we return -22.

3.3 sys_deregister

```
SYSCALL_DEFINE1(deregister, pid_t, pid)
{
    bool found = false;
    struct pid_node *node;
    struct list_head *pos, *n;
    if(pid<1){
        return -22;
    }
    mutex_lock(&monitored_list_mutex);
    list_for_each_safe(pos, n, &monitored_pids_head_node) {
        node = list_entry(pos, struct pid_node, next_prev_list);
        if (node->proc_resource->pid == pid) {
            list_del(&node->next_prev_list);
            kfree(node->proc_resource);
            kfree(node);
            found=true;
            break;
        }
    }
    mutex_unlock(&monitored_list_mutex);
    return found ? 0 : -3;
}
```

Listing 7: Kernel code snippet for sys_deregister in /hello/hello.c

Most of things are similar previous syscalls, the important thing is to iterate the list using list_for_each_safe which enables us to delete a node in between of a traversal as mentioned in kernel code of list.h, further we remove the node from monitoring list using list_del and before returning we free up allocated space for the previously registered node using kfree.

```
/**
 * list_for_each_safe - iterate over a list safe against removal of list entry
 * @pos:    the &struct list_head to use as a loop cursor.
 * @n:      another &struct list_head to use as temporary storage
 * @head:   the head for your list.
 */
#define list_for_each_safe(pos, n, head) \
    for (pos = (head)->next, n = pos->next; \
         !list_is_head(pos, (head)); \
         pos = n, n = pos->next)
```

Listing 8: Kernel code snippet for list_for_each_safe in /include/linux/list.h

4 Resource Usage Limiter

4.1 sys_resource_cap

```
SYSCALL_DEFINE3(resource_cap, pid_t, pid, long, heap_quota, long, file_quota) {
    struct task_struct *task;
    struct pid_node *node = NULL;
    struct list_head *pos, *n;
    bool should_kill = false;
    int found = 0;
    rcu_read_lock();
    task = find_task_by_vpid(pid);
    rcu_read_unlock();
    if (!task) {
        printk(KERN_INFO "Syscall sys_register: PID %d does not exist\n", pid);
        return -3;
    }
    mutex_lock(&monitored_list_mutex);
    list_for_each_safe(pos, n, &monitored_pids_head_node) {
        node = list_entry(pos, struct pid_node, next_prev_list);
        if (node->proc_resource->pid == pid) {
            found = 1;
            break;
        }
    }
    if (!found) {
        mutex_unlock(&monitored_list_mutex);
        return -22;
    }
    if (node->proc_resource->quotas_defined) {
        mutex_unlock(&monitored_list_mutex);
        return -23;
    }
    node->proc_resource->heap_quota = heap_quota*1024*1024;
    node->proc_resource->file_quota = file_quota;
    node->proc_resource->quotas_defined = true;
    if ((node->proc_resource->file_quota >= 0 && node->proc_resource->openfile_count > node->
        proc_resource->file_quota)
        ||(node->proc_resource->heap_quota >= 0 && node->proc_resource->heapsize > node->proc_resource
            ->heap_quota))
    {
        printk(KERN_INFO "File/Heap quota exceeded for PID %d. Sending SIGKILL.\n", pid);
        list_del(&node->next_prev_list);
        kfree(node->proc_resource);
```

```
        kfree(node);
        should_kill = true;
    }
    mutex_unlock(&monitored_list_mutex);
    if(should_kill){
        printk(KERN_INFO "RES_CAP killing the process...\n");
        force_sig(SIGKILL);
    }
    return 0;
}
```

Listing 9: Kernel code snippet for sys_resource_cap in /hello/hello.c

Most of the strategies used to implement it are discussed in previous sections, only important thing to note if quotas/cap that we are setting up for the current process are already exceeded then we free the node corresponding to it and send the SIGKILL to current process using force_sig (in /kernel/signal.c)

4.2 sys_resource_reset

```
SYSCALL_DEFINE1(resource_reset, pid_t, pid)
{
    struct task_struct *task;
    struct pid_node *node = NULL;
    struct list_head *pos;
    rcu_read_lock();
    task = find_task_by_vpid(pid);
    rcu_read_unlock();
    if (!task) {
        printk(KERN_INFO "sys_resource_reset: PID %d does not exist\n", pid);
        return -3;
    }
    mutex_lock(&monitored_list_mutex);
    list_for_each(pos, &monitored_pids_head_node) {
        node = list_entry(pos, struct pid_node, next_prev_list);
        if (node->proc_resource->pid == pid) {
            node->proc_resource->heap_quota = -1;
            node->proc_resource->file_quota = -1;
            node->proc_resource->quotas_defined = false;
            mutex_unlock(&monitored_list_mutex);
            printk(KERN_INFO "sys_resource_reset: Reset quotas for pid %d\n", pid);
            return 0;
        }
    }
    mutex_unlock(&monitored_list_mutex);
    return -22;
}
```

Listing 10: Kernel code snippet for sys_resource_reset in /hello/hello.c

5 Management of File Count and Heap-size

5.1 Updation of File Count

```
static bool update_filecount(void) {
    bool should_kill = false;
    pid_t curr_pid = current->pid;
    struct pid_node *node;
    struct list_head *pos, *n;
    mutex_lock(&monitored_list_mutex);
    list_for_each_safe(pos, n, &monitored_pids_head_node) {
        node = list_entry(pos, struct pid_node, next_prev_list);
        if (node->proc_resource->pid == curr_pid) {
            node->proc_resource->openfile_count++;
            if (node->proc_resource->quotas_defined &&
                node->proc_resource->file_quota >= 0 &&
                node->proc_resource->openfile_count > node->proc_resource->file_quota) {
                printk(KERN_INFO "update_filecount: File quota exceeded for PID %d. Sending SIGKILL.\n
                    ", curr_pid);
                list_del(&node->next_prev_list);
                kfree(node->proc_resource);
                kfree(node);
                should_kill = true;
            }
            break;
        }
    }
    mutex_unlock(&monitored_list_mutex);
    return should_kill;
}
```

Listing 11: Kernel code snippet for file count in /fs/open.c

```
SYSCALL_DEFINE3(open, const char __user *,
    filename, int, flags, umode_t, mode)
{
    if (force_o_largefile())
        flags |= O_LARGEFILE;
    return do_sys_open(AT_FDCWD, filename, flags,
         mode);
}
```

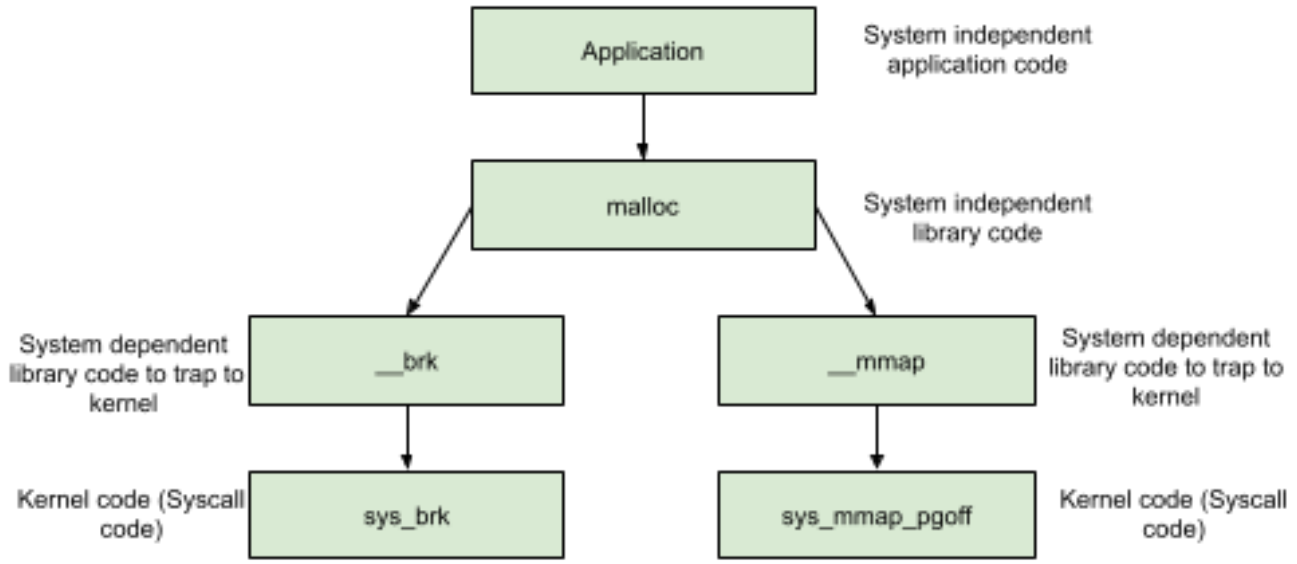
Listing 12: Old Kernel code for open syscall in fs/open.c

```
SYSCALL_DEFINE3(open, const char __user *,
    filename, int, flags, umode_t, mode)
{
    int fd;
    if (force_o_largefile())
        flags |= O_LARGEFILE;
    fd = do_sys_open(AT_FDCWD, filename, flags,
        mode);
    if (fd >= 0) {
        if (update_filecount()) {
            printk(KERN_INFO "OPEN killing the
                process...\n");
            force_sig(SIGKILL);
        }
    }
    return fd;
}
```

Listing 13: New Kernel code for open syscall in fs/open.c

Here to account for the files opened by processes we call update_filecount function whenever open/openat/openat2 syscalls are invoked (In above listings we have given the modification made to open syscall a similar modification has been made to openat & openat2), then the update_filecount goes and check if current process which invoked then open syscall is in monitoring list, if yes then goes and updates the file count for it, after which it checks if the current file count is within the bounds of quotas/caps set, if not in bounds then it kills the process after returning from the function call (indicated by setting should_kill to True)

5.2 Updation of Heap-size




```c
static bool update_heapsize(unsigned long delta_bytes)
{
    bool should_kill = false;
    struct pid_node *node;
    struct list_head *pos, *n;
    pid_t curr_pid = current->pid;
    unsigned long delta = delta_bytes;
    if (delta <= 0)
    {
        return should_kill;
    }
    mutex_lock(&monitored_list_mutex);
    list_for_each_safe(pos, n, &monitored_pids_head_node) {
        node = list_entry(pos, struct pid_node, next_prev_list);
        if (node->proc_resource->pid == curr_pid) {
            printk(KERN_INFO "Updating Heap Size (Using MMAP) (For PID: %d): %lu \n", curr_pid,
                delta_bytes);
            node->proc_resource->heapsize += delta;
            if (node->proc_resource->quotas_defined &&
                node->proc_resource->heap_quota >= 0 &&
                node->proc_resource->heapsize > node->proc_resource->heap_quota) {
                printk(KERN_INFO "Heap quota exceeded for PID %d. Sending SIGKILL.\n", curr_pid);
                list_del(&node->next_prev_list);
                kfree(node->proc_resource);
                kfree(node);
                should_kill = true;
            }
            break;
        }
    }
    mutex_unlock(&monitored_list_mutex);
    return should_kill;
}
```

Listing 14: Kernel code snippet for accounting heap-size in /mm/mmap.c & /arch/x86/kernel/sys$_x$86$_6$4.c

```
SYSCALL_DEFINE6(mmap, unsigned long, addr,
    unsigned long, len,
        unsigned long, prot, unsigned long, flags
            ,
        unsigned long, fd, unsigned long, off)
{
    if (off & ~PAGE_MASK)
        return -EINVAL;

    return ksys_mmap_pgoff(addr, len, prot, flags
        , fd, off >> PAGE_SHIFT);
}
```

Listing 15: Old Kernel code for mmap syscall in /arch/x86/kernel/sys_x86_64.c

```
SYSCALL_DEFINE6(mmap, unsigned long, addr,
    unsigned long, len,
        unsigned long, prot, unsigned long, flags
            ,
        unsigned long, fd, unsigned long, off)
{
    unsigned long result;
    if (off & ~PAGE_MASK)
        return -EINVAL;
    result = ksys_mmap_pgoff(addr, len, prot,
        flags, fd, off >> PAGE_SHIFT);
    if (!IS_ERR_VALUE(result) && ((flags & (
        MAP_ANONYMOUS | MAP_PRIVATE)) == (
        MAP_ANONYMOUS | MAP_PRIVATE)))
    {
        unsigned long aligned_len = PAGE_ALIGN(
            len);
        if(update_heapsize((long)aligned_len))
        {
            force_sig(SIGKILL);
        }
    }
    return result;
}
```

Listing 16: New Kernel code for mmap syscall in /arch/x86/kernel/sys_x86_64.c

```
SYSCALL_DEFINE1(brk, unsigned long, brk)
{
    unsigned long newbrk, oldbrk, origbrk;
    struct mm_struct *mm = current->mm;
    struct vm_area_struct *brkvma, *next =
        NULL;
    unsigned long min_brk;
    bool populate;
    bool downgraded = false;
    ...
success:
    populate = newbrk > oldbrk && (mm->
        def_flags & VM_LOCKED) != 0;
    if (downgraded)
        mmap_read_unlock(mm);
    else
        mmap_write_unlock(mm);
    userfaultfd_unmap_complete(mm, &uf);
    if (populate)
        mm_populate(oldbrk, newbrk - oldbrk);
    return brk;

out:
    mmap_write_unlock(mm);
    return origbrk;
}
```

Listing 17: Old Kernel code for brk syscall in /mm/mmap.c

```
SYSCALL_DEFINE1(brk, unsigned long, brk)
{
    unsigned long newbrk, oldbrk, origbrk;
    struct mm_struct *mm = current->mm;
    struct vm_area_struct *brkvma, *next =
        NULL;
    unsigned long min_brk;
    unsigned long delta;
    bool populate;
    bool downgraded = false;
    ...
success:
    populate = newbrk > oldbrk && (mm->
        def_flags & VM_LOCKED) != 0;
    delta = newbrk - oldbrk;
    if (downgraded)
        mmap_read_unlock(mm);
    else
        mmap_write_unlock(mm);
    userfaultfd_unmap_complete(mm, &uf);
    if (populate)
        mm_populate(oldbrk, newbrk - oldbrk);
    if(update_heapsize(delta))
    {
        force_sig(SIGKILL);
    }
    return brk;
out:
    mmap_write_unlock(mm);
    return origbrk;
}
```

Listing 18: New Kernel code for brk syscall in /mm/mmap.c

Before starting a discussion on implementation, we can look into how malloc allocates the heap. Here is statement from the documentation

Malloc Details

Normally, `malloc()` allocates memory from the heap, and adjusts the size of the heap as required, using `sbrk(2)`. When allocating blocks of memory larger than `MMAP_THRESHOLD` bytes, the glibc `malloc()` implementation allocates the memory as a private anonymous mapping using `mmap(2)`. `MMAP_THRESHOLD` is 128 kB by default, but is adjustable using `mallopt(3)`.

This explains why we need to filter private anonymous mappings while accounting for heap regions that get allocated using mmap. If we see update_heapsize is mostly similar to the update_filecount only, and there invocation in either brk/mmap syscall is mostly similar to previous section, only thing to note is we are passing PAGE_ALIGN lengths to increment our heap-size.

6 Other Details of Implementation

6.1 Modifications in 'syscall_64.tbl' & 'syscalls.h'

```
375   451 common  hello   sys_hello
376   452 common  register    sys_register
377   453 common  fetch   sys_fetch
378   454 common  deregister  sys_deregister
379   455 common  resource_cap    sys_resource_cap
380   456 common  resource_reset  sys_resource_reset
381
```

Figure 1: Modifications to syscall_64.tbl

```
298  /*
299   * These syscall function prototypes are kept in the same order as
300   * include/uapi/asm-generic/unistd.h. Architecture specific entries go below,
301   * followed by deprecated or obsolete system calls.
302   *
303   * Please note that these prototypes here are only provided for information
304   * purposes, for static analysis, and for linking from the syscall table.
305   * These functions should not be called elsewhere from kernel code.
306   *
307   * As the syscall calling convention may be different from the default
308   * for architectures overriding the syscall calling convention, do not
309   * include the prototypes if CONFIG_ARCH_HAS_SYSCALL_WRAPPER is enabled.
310   */
311  #ifndef CONFIG_ARCH_HAS_SYSCALL_WRAPPER
312  asmlinkage long sys_hello(void);
313  asmlinkage int sys_register(pid_t pid);
314  asmlinkage int sys_fetch(struct per_proc_resource __user *stats, pid_t pid);
315  asmlinkage int sys_deregister(pid_t pid);
316  asmlinkage int sys_resource_cap(pid_t pid, long heap_quota, long file_quota);
317  asmlinkage int sys_resource_reset(pid_t pid);
```

Figure 2: Modifications to syscalls.h

7 Extending the implementation to handle Process and Thread Group

The submission we have made is 'thread specific implementation' here we have discussed some ideas to extend them to 'process specific implementation'. For simplification assume that we are given input only pid of main thread of thread group whose 'tgid' is equal to 'pid'. So when we will register/fetch/deregister/set resource cap we will do it for the entire group of threads in that process/thread group.

7.1 Modification needed to Node structure

```
struct per_proc_resource {
    pid_t pid;
    unsigned long heapsize;
    unsigned long openfile_count;
    long heap_quota;
    long file_quota;
    bool quotas_defined;
    pid_t tgid;
};
```

Listing 19: Suggested Update For Node Structure

keeping track of tgid will help us enable faster deletion when pid of main thread of thread group is provided.

7.2 Modifications needed to sys_register

```
SYSCALL_DEFINE1(register, pid_t, pid)
{
    struct task_struct *task, *thread, *leader;
    struct pid_node *node;
    struct list_head *pos;
    int ret = 0;
    printk(KERN_INFO "Syscall sys_register called with pid: %d\n", pid);
    if (pid < 1) {
        printk(KERN_INFO "sys_register: PID %d is less than 1\n", pid);
        return -22;
    }
    rcu_read_lock();
    task = find_task_by_vpid(pid);
    rcu_read_unlock();
    if (!task) {
        printk(KERN_INFO "sys_register: PID %d does not exist\n", pid);
        return -3;
    }
    leader = task->group_leader;
    mutex_lock(&monitored_list_mutex);
    for_each_thread(leader, thread) {
        list_for_each(pos, &monitored_pids_head_node) {
            node = list_entry(pos, struct pid_node, next_prev_list);
            if (node->proc_resource->pid == thread->pid) {
                mutex_unlock(&monitored_list_mutex);
                printk(KERN_INFO "sys_register: Thread %d (in group %d) is already monitored\n",
                       thread->pid, leader->pid);
                return -23;
            }
        }
    }
    for_each_thread(leader, thread) {
        node = kmalloc(sizeof(struct pid_node), GFP_KERNEL);
        if (!node) {
            ret = -ENOMEM;
            goto rollback;
```

```
        }
        node->proc_resource = kmalloc(sizeof(struct per_proc_resource), GFP_KERNEL);
        if (!node->proc_resource) {
            kfree(node);
            ret = -ENOMEM;
            goto rollback;
        }
        node->proc_resource->pid = thread->pid;
        node->proc_resource->tgid = leader->pid;
        node->proc_resource->heapsize = 0;
        node->proc_resource->openfile_count = 0;
        node->proc_resource->heap_quota = -1;
        node->proc_resource->file_quota = -1;
        node->proc_resource->quotas_defined = false;
        list_add(&node->next_prev_list, &monitored_pids_head_node);
        printk(KERN_INFO "sys_register: Registered thread %d in group %d\n",
               thread->pid, leader->pid);
    }
    mutex_unlock(&monitored_list_mutex);
    return 0;
rollback:
    {
        struct list_head *q, *tmp;
        list_for_each_safe(q, tmp, &monitored_pids_head_node) {
            node = list_entry(q, struct pid_node, next_prev_list);
            if (node->proc_resource->tgid == leader->pid) {
                list_del(q);
                kfree(node->proc_resource);
                kfree(node);
            }
        }
    }
    mutex_unlock(&monitored_list_mutex);
    printk(KERN_ERR "sys_register: Memory allocation failed for thread group %d\n",
           leader->pid);
    return ret;
}
```

Listing 20: Suggested Update For sys_register

Only main diffrences includes getting task struct to leader thread, since we assume pid of leader thread will be given as input then we can expect leader=task holds further we can use for_each_thread (defined at /include/linux/sched/signal.h) to go through all the threads of that thread group and add them to the monitoring list. For the current convention we assume if some thread already we discard entire thread group and free up any allocation made for this group (here tgid stored in node helps in deletion).

7.3 Modifications needed to sys_fetch

```
SYSCALL_DEFINE2(fetch, struct per_proc_resource __user *, stats, pid_t, pid)
{
    struct pid_node *node;
    struct list_head *pos;
    struct per_proc_resource kstats = {0};
    bool found = false;
    if (!stats || !access_ok(stats, sizeof(struct per_proc_resource))) {
        return -22;
    }
    kstats.pid = pid;
    kstats.tgid = pid;
    kstats.heapsize = 0;
    kstats.openfile_count = 0;
    kstats.heap_quota = -1;
    kstats.file_quota = -1;
```

```
    kstats.quotas_defined = false;
    mutex_lock(&monitored_list_mutex);
    list_for_each(pos, &monitored_pids_head_node) {
        node = list_entry(pos, struct pid_node, next_prev_list);
        if (node->proc_resource->tgid == pid) {
            kstats.heapsize += node->proc_resource->heapsize;
            kstats.openfile_count += node->proc_resource->openfile_count;
            if (!found) {
                kstats.heap_quota = node->proc_resource->heap_quota;
                kstats.file_quota = node->proc_resource->file_quota;
                kstats.quotas_defined = node->proc_resource->quotas_defined;
                found = true;
            }
        }
    }
    mutex_unlock(&monitored_list_mutex);
    if (!found) {
        printk(KERN_INFO "sys_fetch: No monitored threads found for tgid %d\n", pid);
        return -22;
    }
    kstats.heapsize = kstats.heapsize / (1024 * 1024);
    if (copy_to_user(stats, &kstats, sizeof(struct per_proc_resource))) {
        printk(KERN_ERR "sys_fetch: copy_to_user failed for tgid %d\n", pid);
        return -22;
    }
    return 0;
}
```

Listing 21: Suggested Update For sys_fetch

Now again there is a similar update that may be needed for sys_fetch. We go through the monitoring list and check nodes where the tgid matches input pid (assuming input pid is leader thread's pid) and then accumulate the file opened and heap allocated and then copy the accumulation to the location pointed by pointer stats.

7.4 Other Concerns related to new implementation

A more or less similar approach to complete the remaining syscalls, and handling updates to file-count and heap-size is mostly same ,only thing that will change is killing condition, for which we will need to do an accumulation similar to in previous section and check against cap/quotas. But more important thing that we will need to account for is that some threads may get added to thread group beyond the point when the process got registered and some threads in thread group may finish the execution before the entire process/thread group has actually come to termination. For which we will need to modify fork(),execv() and exit() system calls, when fork() and execv() are used to create a new thread (and not a new process) we can look at it's leader's pid , if that is part of monitoring list we need to add this newly created thread to the list while when thread exits if that thread in monitoring list we need to remove that from monitoring list. So there significant part that needs to taken into account before we extend the current implementation into 'process specific implementation'.