Operating Systems Assignment 1 - HARD - ELL783

Suraj Natekar
2024EET2355

March 6, 2025

1 Introduction

This report details the implementation of a resource usage tracker and limiter for the Linux kernel version 6.1.6. The implementation monitors specific processes for heap memory usage and open file descriptors, and enforces resource quotas by terminating processes that exceed defined limits.

The system consists of two main components:

- Resource Usage Tracker: Monitors selected processes and collects data on heap memory and open file descriptors
- Resource Usage Limiter: Enforces resource quotas on monitored processes

2 Environment Setup

The implementation was developed and tested on Linux kernel version 6.1.6 running on a virtual machine. The development environment was configured according to the assignment specifications, with essential tools and libraries required for kernel modification and compilation.

3 Resource Usage Tracker Implementation

3.1 Data Structure Design

The resource usage tracker utilizes a doubly linked list to maintain information about monitored processes. The data structures are defined in the header file and used throughout the implementation.

```c
struct per_proc_resource {
    pid_t pid;
    unsigned long heapsize;       // Heap usage (in MB)
    unsigned long openfile_count; // Number of open files
};

struct pid_node {
    struct per_proc_resource *proc_resource;
    struct list_head list;
};
```

Listing 1: Header File: resource_tracker.h

The implementation initializes the global data structures in the source file:

```c
// Define the global get heap size and count open files
unsigned long get_process_heap_size(pid_t pid);
unsigned long count_open_files(struct task_struct *task);
```

Listing 2: Global Data Structures Initialization

This design uses the Linux kernel's list mechanism to manage the linked list of monitored processes. A spinlock protects access to this shared data structure to ensure thread safety when multiple system calls operate on the list concurrently.

3.2 Resource Calculation Functions

Two key utility functions calculate resource usage for monitored processes:

```c
unsigned long get_process_heap_size(pid_t pid) {
    struct task_struct *task;
    struct vm_area_struct *vma;
    unsigned long heap_size = 0;
    struct mm_struct *mm;
    MA_STATE(mas, NULL, 0, 0);
    // Find the task_struct for the given PID
    task = find_task_by_vpid(pid);
    if (!task)
        return 0;

    mm=get_task_mm(task);

    if (!mm) {
        return 0;
    }

    heap_size += task->mm->brk - task->mm->start_brk;

    // Iterate over VM areas using Maple Tree
    mas_for_each(&mas, vma, ULONG_MAX-1) {
        if (vma && vma->anon_vma != NULL && vma->vm_file ==
   NULL) {
            heap_size += vma->vm_end - vma->vm_start;
        }
    }

    mmput(mm);

    return heap_size >> 20;  // Convert bytes to MB
}
```

Listing 3: code for Heap Size Calculation

The `get_process_heap_size` function calculates the total heap memory usage of a process by examining its virtual memory areas. This function performs two critical operations:

1. It traverses the process's memory map using the Maple Tree data structure to identify anonymous memory regions (those not backed by files), which typically represent dynamically allocated memory.

2. It accounts for the traditional heap segment by calculating the difference between the current break value (`brk`) and the starting break value (`start_brk`).

The function converts the total size from bytes to megabytes by right-shifting by 20 bits, which aligns with the assignment requirements for reporting heap usage in MB.

```
unsigned long count_open_files(struct task_struct *task) {
    struct fdtable *fdt;
    unsigned long file_count = 0;

    if (!task->files || !task->files->fdt)
        return 0;

    fdt = rcu_dereference_raw(task->files->fdt);
    if (fdt && fdt->fd) {
        int i;
        for (i = 0; i < fdt->max_fds; i++) {
            if (rcu_dereference_raw(fdt->fd[i]) != NULL)
                file_count++;
        }
    }

    return file_count;
}
```
Listing 4: File Count Calculation Function

The `count_open_files` function determines the number of open file descriptors used by a process. The implementation follows these steps:

1. It first checks whether the process has a valid file descriptor table by ensuring that both `files` and `fdt` pointers are non-null.
2. The function then accesses the file descriptor table using `rcu_dereference_raw`, which ensures safe access in an RCU (Read-Copy-Update) environment, preventing race conditions.
3. It iterates over all possible file descriptors up to the process's maximum limit (`max_fds`).
4. During iteration, it checks if each file descriptor entry is non-null, indicating an open file, and increments the count accordingly.

This approach ensures that the function provides an accurate count of open file descriptors for the given process while maintaining safety in concurrent environments using RCU mechanisms.

3.3 System Call Implementations

3.3.1 sys_register System Call

The `sys_register` system call adds a process to the monitored list:

```
\begin{lstlisting}[language=C, caption=System Call for
   Registering a Process]
SYSCALL_DEFINE1(register, pid_t pid):
    if pid < 1:
        return -EINVAL  // Invalid process ID

    task = find_task_by_vpid(pid)
    if task does not exist:
        return -ESRCH  // Process not found

    increment task reference count using get_task_struct(
   task)
```

```
 allocate new_node and new_proc_resource
 if allocation fails:
     decrement task reference count with put_task_struct(
task)
     return -ENOMEM  // Memory allocation failed

 initialize new_proc_resource with:
     pid = task.pid
     heapsize = 0  // Initialized to zero
     openfile_count = 0  // Initialized to zero

 assign new_proc_resource to new_node

 lock(pid_list_lock)

 iterate through pid_list to check if pid is already
monitored
 if pid is found:
     set already_monitored flag to true

 if not already monitored:
     add new_node to pid_list

 unlock(pid_list_lock)

 decrement task reference count using put_task_struct(
task)

 if already monitored:
     free allocated new_proc_resource and new_node
     return -EEXIST  // Already registered

 return 0  // Success
```

Listing 5: Pseudocode for sys_register

The `sys_register` system call registers a process for resource tracking. It first validates the PID and locates the process using `find_task_by_vpid`. If the process is found, its reference count is incremented using `get_task_struct` to prevent premature deallocation. The system then allocates memory for tracking structures. If allocation fails, the reference count is decremented, and an error is returned.

The function initializes resource tracking structures with default values (zero heap size and open file count) and checks whether the process is already monitored by iterating through the `pid_list`. If the process is not already registered, it is added to the list. A spinlock (`pid_list_lock`) ensures safe concurrent access during this operation. Finally, the reference count is decremented using `put_task_struct`. If the process was already registered, allocated memory is freed, and an error is returned.

3.3.2 sys_fetch System Call

The `sys_fetch` system call retrieves current resource usage for a monitored process:

```
SYSCALL_DEFINE2(fetch, struct per_proc_resource __user *
    stats, pid_t pid):
     if pid < 1:
         return -EINVAL  // Invalid process ID

     iterate through pid_list to find the process:
         if the process is found:
             task = find_task_by_vpid(pid)
             if task does not exist:
                 return -ESRCH  // Process not found

             initialize temp_stats with:
                 pid = task.pid
                 heapsize = get_process_heap_size(pid)  //
    Compute heap size
                 openfile_count = count_open_files(task)  //
    Compute open file count

             copy temp_stats to user space
             if copy fails:
                 return -EFAULT  // Invalid user space memory

             return 0  // Success

     return -ESRCH  // PID not found in monitored list
```

Listing 6: System Call for Fetching Process Resource Usage

The `sys_fetch` system call retrieves the latest resource usage of a monitored process. It first verifies if the process exists in the monitored list, then retrieves the task structure. Using `get_process_heap_size(pid)` and `count_open_files(task)`, it updates the heap size and file count. The updated values are stored and safely copied to user space using `copy_to_user`, ensuring proper synchronization.

3.3.3 sys_deregister System Call

The sys_deregister system call removes a process from the monitored list:

```
SYSCALL_DEFINE1(deregister, pid_t pid):
    if pid is invalid:
        return -EINVAL  // Invalid process ID

    lock(pid_list_lock)

    iterate through pid_list safely:
        if process with matching pid is found:
            remove it from the list
            free associated process resource memory
            free node memory
            unlock(pid_list_lock)
            return 0  // Successfully deregistered

    unlock(pid_list_lock)

    return -ESRCH  // Process not found
```

Listing 7: Pseudocode for sys_deregister

The sys_deregister system call removes a process from tracking. It first validates the PID, then searches the monitored list using list_for_each_entry_safe for safe deletion. If found, it removes the entry, frees allocated memory, and returns an appropriate status code. Proper locking ensures thread safety during list modifications.

4 Resource Usage Limiter Implementation

4.1 System Call Implementations

4.1.1 sys_resource_cap System Call

The sys_resource_cap system call sets resource limits for a monitored process. It ensures that the process exists and is already registered before updating its heap and file descriptor quotas. Proper synchronization is maintained using spinlocks.

```
SYSCALL_DEFINE3(resource_cap, pid_t pid, long heap_quota,
   long file_quota):
    if pid is invalid:
        return -EINVAL  // Invalid process ID
    task = find_task_by_vpid(pid)
    if task does not exist:
        return -ESRCH  // Process not found

    increase task reference count

    lock(pid_list_lock)

    iterate through pid_list to find the process:
        if process with matching pid is found:
            mark as found
            break

    if process is not in monitored list:
        unlock(pid_list_lock)
        return -EINVAL  // PID not in monitored list

    if quotas are already set:
        unlock(pid_list_lock)
        return -EEXIST  // Quotas already defined

    set task.heap_quota = heap_quota
    set task.file_quota = file_quota

    log quota assignment

    unlock(pid_list_lock)
    decrease task reference count

    return 0  // Success
```

Listing 8: Pseudocode for sys_resource_cap

4.1.2 sys_resource_reset System Call

The `sys_resource_reset` system call removes resource limits for a monitored process, resetting quotas to unlimited (-1). It verifies that the process is registered and uses spinlocks to ensure safe modifications.

```
SYSCALL_DEFINE1(resource_reset, pid_t pid):
    if pid is invalid:
        return -EINVAL  // Invalid process ID

    task = find_task_by_vpid(pid)
    if task does not exist:
        return -ESRCH  // Process not found

    lock(pid_list_lock)

    iterate through pid_list to find the process:
        if process with matching pid is found:
            mark as found
            break

    if process is not in monitored list:
        unlock(pid_list_lock)
        return -EINVAL  // PID not in monitored list

    if task.heap_quota or task.file_quota are not -1:
        unlock(pid_list_lock)
        return -EEXIST  // Quotas are not set

    reset task.heap_quota = -1
    reset task.file_quota = -1

    unlock(pid_list_lock)

    return 0  // Success
```

Listing 9: Pseudocode for sys_resource_reset

5 Other Modification Details

5.1 System Call Declarations

The following system calls were added in `include/linux`:

- `sys_register`, `sys_fetch`, `sys_deregister`, `sys_resource_cap`,`sys_debug_traverse` and `sys_resource_reset`.

5.2 System Call Table Registration

Entries for the new system calls were added to the syscall table:

- Lines 451-456 in the syscall table define mappings for all implemented system calls.
- I have also added one syscall of of traversing the monitered list.

5.3 Process Structure Modification

- Added `heap_quota` and `file_quota` fields in `sched.h` to track resource limits per process.
- Initialized these fields to `-1` in `fork.c`, indicating no limits by default.

5.4 File Limit Enforcement

- Added a check in `open.c` to validate the task structure before proceeding.
- Implemented a mechanism to terminate processes exceeding their file descriptor limits.

5.5 Heap Quota Enforcement

- Added logic in `mmap.c` to track and compare heap usage against the process's heap quota.
- If a process exceeds its heap quota, it is deregistered from monitoring and terminated using `SIGKILL`.

6 Conclusion

This implementation of the resource usage tracker and limiter showcases the extensibility of the Linux kernel's modular architecture. By introducing new system calls, we have developed a mechanism to monitor and regulate process resource consumption, aiding system administrators and application developers in effective resource management.

The implementation ensures a balance between efficiency and correctness by leveraging kernel data structures, synchronization mechanisms like spinlocks, and proper memory management to maintain thread safety and system performance. This solution establishes a robust foundation that can be further enhanced with additional capabilities, such as dynamic quota adjustments, more granular resource tracking, or deeper integration with existing kernel subsystems.