University of Dhaka

Department of Computer Science and Engineering

CSE:3211 Operating Systems Lab

Submitted By:

Ahaj Mahin Faiak

Roll No: 01

Md. Mahmudul Hasan

Roll No: 10

Abrar Eyasir

Roll No: 12

Submitted On:

March 20,2025

Submitted To:

Dr. Mosaddek Hossain Kamal, Professor

Contents

1 Introduction . . . **3**
- 1.1 Overview . . . 3
- 1.2 Objectives . . . 3

2 Memory Management . . . **3**
- 2.1 Heap Management . . . 3
- 2.2 Heap Initialization . . . 4
  - 2.2.1 Memory Allocation Strategy . . . 4
  - 2.2.2 Free List Management . . . 4
  - 2.2.3 Memory Alignment and Optimization . . . 4
- 2.3 Stack Management . . . 5
  - 2.3.1 Stack Allocation and Initialization . . . 5
  - 2.3.2 Task Control Block (TCB) and Stack Initialization . . . 5
  - 2.3.3 Stack Execution Flow . . . 6

3 System Call . . . **6**
- 3.1 System Call Mechanism . . . 6
- 3.2 SYS_fork Implementation . . . 8
- 3.3 SYS_execv Implementation . . . 9
- 3.4 SYS_getpid Implementation . . . 10
- 3.5 SYS_exit Implementation . . . 10
- 3.6 SYS_malloc Implementation . . . 11

4 Task Scheduling . . . **11**
- 4.1 Scheduler Initialization . . . 11
- 4.2 Task States and Queue . . . 12
- 4.3 Context Switching via PendSV . . . 12
- 4.4 Scheduling Triggers . . . 14
  - 4.4.1 Voluntary Yield . . . 14
  - 4.4.2 System Timer . . . 15
  - 4.4.3 Task Termination . . . 15

5 Results . . . **16**
- 5.1 Syscall execution from Userland . . . 16
- 5.2 Initialize task . . . 17
- 5.3 Round Robin Scheduler . . . 18
- 5.4 SYS_fork example . . . 19
- 5.5 SYS_execv example . . . 20
- 5.6 System call functions example . . . 21

6 Applying the Patch to DUOS 22
6.1 Procedure for Patch Application 22
6.2 Platform-Specific Considerations 22

1 Introduction

1.1 Overview

This lab assignment focuses on designing and implementing essential features of the DUOS operating system, including system calls, thread management, and multitasking scheduling. The aim is to ensure efficient system resource management and reliable task execution. This report outlines the architecture of the syscall interface, the communication mechanisms used for task scheduling, and the memory management techniques employed to support dynamic task execution.

1.2 Objectives

The objectives of this lab are to:

- **Implement OS service calls** using `SVC` and `SVCPend`.
- **Develop and manage memory services**, including heap and stack operations through system calls such as:
  - `SYS_open`, `SYS_read`, `SYS_write`
  - `SYS_malloc`, `SYS_free`
  - `SYS_fork`, `SYS_execv`
  - `SYS_getpid`, `SYS_exit`
  - `SYS_yield`
- **Implement kernel services** for creating and managing threads and processes in RAM, along with scheduling algorithms such as:
  - **Time-sharing scheduling**
  - **First-Come, First-Served (FCFS) scheduling**
- **Compare and evaluate** the performance of different scheduling strategies.

2 Memory Management

2.1 Heap Management

Memory management is an important part of operating system design, particularly in embedded systems like DUOS. Heap is a dynamically allocated memory pool that enables efficient memory management for both kernel and user applications. DUOS provides system calls such as `SYS_malloc` and `SYS_free` to handle dynamic memory allocation and deallocation.

2.2 Heap Initialization

At system startup, DUOS initializes a contiguous heap memory region for dynamic allocation. This region is separate from stack and global/static memory sections. The heap is defined within the linker script as:

```
.heap :
{
    . = ALIGN(4);
    _sheap = .;
    . = . + 32K;
    *(.heap)
    . = ALIGN(4);
    _eheap = .;
} >SRAM
```

This defines a 32KB heap in SRAM, ensuring proper memory segmentation.

2.2.1 Memory Allocation Strategy

DUOS employs the **first-fit** strategy for memory allocation. The allocator scans the free list and assigns the first available block that meets the request size. This approach balances allocation efficiency and memory fragmentation.

2.2.2 Free List Management

To optimize memory reuse, DUOS maintains a linked list of free memory blocks. When a block is freed, it is returned to this list. Additionally, adjacent free blocks are merged periodically to reduce fragmentation and maximize contiguous memory availability.

2.2.3 Memory Alignment and Optimization

All allocated memory blocks in DUOS adhere to an **8-byte alignment** boundary. Proper alignment enhances memory access speed and ensures compatibility with the processor's instruction set.

2.3 Stack Management

Each task in DUOS requires a dedicated stack space to store execution context, function calls, and local variables. Unlike the heap, the stack follows a **Last-In, First-Out (LIFO)** structure.

2.3.1 Stack Allocation and Initialization

Upon task creation, DUOS assigns a dedicated stack region. A typical stack size is **1024 words**, ensuring sufficient memory for execution. Stack initialization includes:

- Assigning the **Process Stack Pointer (PSP)** for user tasks.
- Setting the **Main Stack Pointer (MSP)** for kernel operations.
- Preloading system registers (xPSR, PC, LR) with default values.

2.3.2 Task Control Block (TCB) and Stack Initialization

The `create_tcb()` function configures the **Task Control Block (TCB)** and stack for a new task:

```c
void create_tcb(TCB_TypeDef *tcb, void (*func_ptr)(void), uint32_t *stack_start)
{
    tcb->magic_number = MAGIC_NUMBER;
    tcb->task_id = TASK_ID;
    TASK_ID++;
    tcb->status = READY;
    tcb->execution_time = 0;
    tcb->waiting_time = 0;
    tcb->digital_signature = DIGITAL_SIGNATURE;

    // Assign process stack pointer (PSP)
    tcb->psp = stack_start;

    // Initialize stack with default register values
    *(--tcb->psp) = 0x01000000;  // xPSR
    *(--tcb->psp) = (uint32_t)func_ptr; // PC
    *(--tcb->psp) = 0xFFFFFFFD; // LR

    // Reserve space for general-purpose registers
    for (uint32_t i = 0; i < 13; i++)
    {
        *(--tcb->psp) = 0;
    }

    // Ensure memory consistency
    __ISB();
}
```

2.3.3 Stack Execution Flow

- The stack is initialized with default values.
- The program counter (PC) points to the task entry function.
- General-purpose registers are allocated stack space.
- The Instruction Synchronization Barrier (ISB) ensures memory consistency before execution.

3 System Call

System invocations enable applications to request services from the operating system kernel. In **ARM-based architectures**, the **SVC (Supervisor Call)** instruction generates an exception, transitioning the processor to **privileged mode** to securely execute kernel operations.

Frequently used system invocations include: `SYS_exit`, `SYS_getpid`, `SYS_read`, `SYS_write`, `SYS_time`, `SYS_reboot`, `SYS_yield`, `SYS_malloc`, `SYS_free`, `SYS_fork`, and `SYS_execv`. These invocations manage process control, memory allocation, I/O operations, and system administration.

3.1 System Call Mechanism

System services are facilitated through the **Supervisor Call (SVC)** instruction, which generates an exception to transfer control to the kernel.

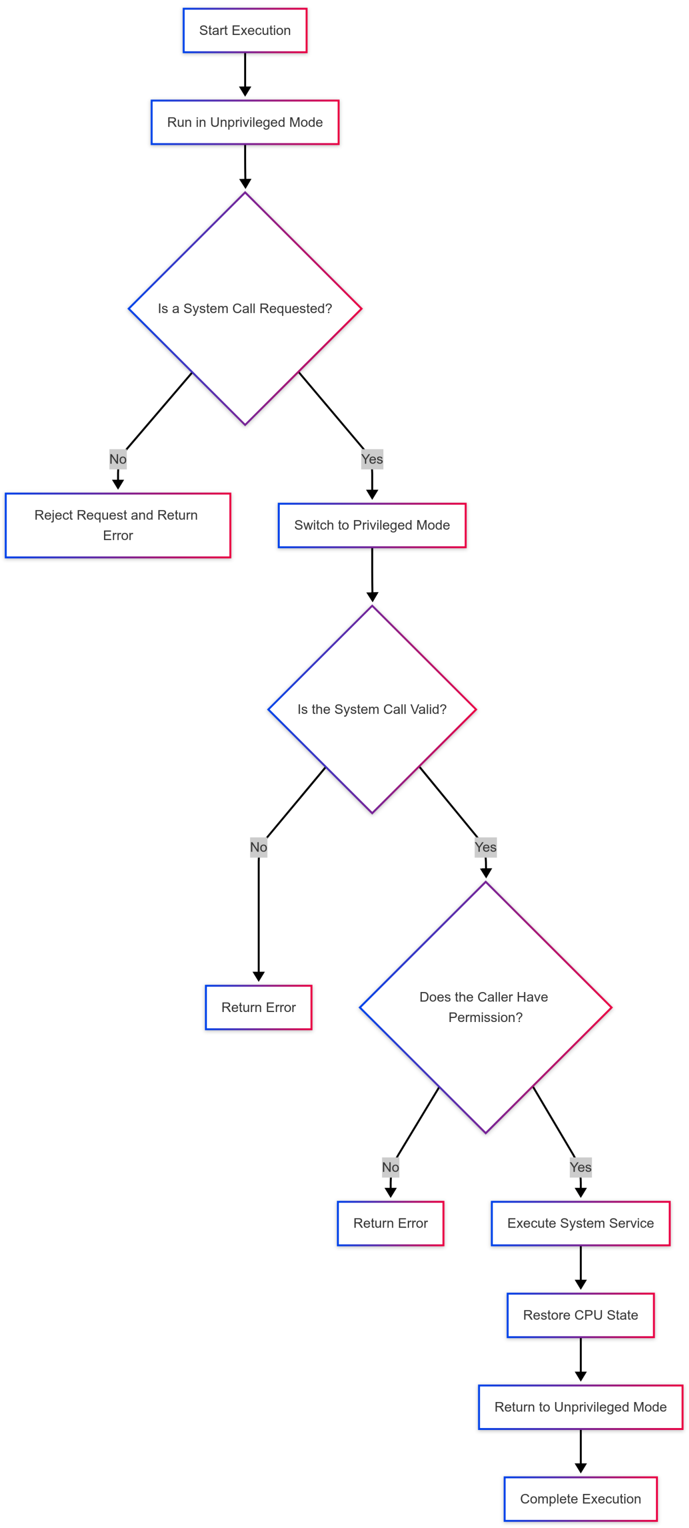

Start Execution
Run in Unprivileged Mode
Is a System Call Requested?
No
Yes
Reject Request and Return Error
Switch to Privileged Mode
Is the System Call Valid?
No
Yes
Return Error
Does the Caller Have Permission?
No
Yes
Return Error
Execute System Service
Restore CPU State
Return to Unprivileged Mode
Complete Execution

Figure 1: Flowchart of System Call Steps

The system invocation number is retrieved as follows:

```
uint32_t syscall_num = ((uint8_t *)svc_params[6])[-2];
```

The kernel handler processes the request using a `switch` statement to execute the corresponding service:

```
switch (syscall_num) {
    case SYS_read:
        // Handle read request
        break;
    case SYS_write:
        // Handle write request
        break;
    // Additional cases for other services
    default:
        // Handle invalid syscall number
        break;
}
```

This method ensures a secure transition from **user mode** to **kernel mode**, preserving system integrity and stability.

3.2 SYS_fork Implementation

The `fork` system invocation creates a new process by duplicating the currently executing process. The implementation involves the following steps:

- The parent process's stack pointer (`parent_psp`) is used to access the current execution context.
- A new stack space is allocated for the child process (`child_stack[1024]`).
- The child process's Task Control Block (TCB) is initialized with a unique task ID:

  ```
  tcb->magic_num = MAGIC_VALUE;
  tcb->task_id = CURRENT_TASK_ID;
  CURRENT_TASK_ID++;
  ```

- The parent's context, including register values and program counter, is copied to the child's stack:

```
*(--tcb->psp) = parent_psp[7];
*(--tcb->psp) = (uint32_t)parent_psp[6];
*(--tcb->psp) = parent_psp[5];
```

- Different return values are set for the parent and child:

  ```
  parent_psp[0] = CURRENT_TASK_ID + 1;
  *(--tcb->psp) = 0;
  ```

- The child process is added to the `READY` queue with status `READY`.
- The syscall handler (`syscall.c`) processes the `fork` request by calling `sys_fork()` and storing the return value in `svc_args[2]`.

3.3 SYS_execv Implementation

The `execv` system invocation replaces the current process image with a new one by performing the following steps:

- The system searches for the requested executable in the file system using the `find_file()` function:

  ```
  int find_file(char *filename) {
      for (uint32_t i = 0; i < total_files; i++) {
          if (strcomp((uint8_t *)file_catalog[i].name,
          (uint8_t *)filename) == 0) {
              return i;
          }
      }
      return -1;
  }
  ```

- The file system is an in-memory structure mapping file names to function pointers:

  ```
  file_entry_t file1;
  file1.address = (uint32_t *)program_B;
  file1.size = 1024;
  file1.mode = O_RDONLY;
  strcopy((uint8_t *)file1.name, (const uint8_t *)"PROGRAM_B");
  file_catalog[total_files++] = file1;
  ```

- The current process's stack is reconfigured to use the entry point of the new program.
- The syscall handler processes the request using `sys_execv()`:

  ```
  char *filename = (char *)svc_args[0];
  char **argv = (char **)svc_args[1];
  char **envp = (char **)svc_args[2];
  int result = __sys_execv(filename, argv, envp);
  svc_args[0] = result;
  ```

3.4 SYS_getpid Implementation

The `getpid` system invocation retrieves the task ID from the current task's TCB.

```
int pid = READY_QUEUE[CURRENT_TASK_INDEX].task_id;
svc_args[0] = pid;
```

The user-level function `getPID()` invokes the system call and returns the result:

```
asm volatile("MOV %0, R0" : "=r"(pid));
return pid;
```

3.5 SYS_exit Implementation

The `exit` system invocation terminates the current process by marking it as killed:

```
TCB_TypeDef *target_tcb = (TCB_TypeDef *)svc_args[2];
target_tcb->status = TERMINATED;
```

The user-level function `task_exit()` provides the current task's TCB to the system call:

```
TCB_TypeDef *current_tcb = READY_QUEUE + CURRENT_TASK_INDEX;
__asm volatile("MOV R2, %0\n" : : "r"(current_tcb));
```

After marking the task as terminated, the function calls `yield()` to allow the scheduler to switch tasks.

3.6 SYS_malloc Implementation

The `malloc` system invocation allocates memory from the heap:

```
__asm volatile("mov r2, %0\n" : : "r"(size));
```

The syscall handler uses `heap_malloc()` to allocate memory:

```
uint32_t alloc_size = svc_args[2];
void *memory_ptr = heap_malloc(alloc_size);
svc_args[2] = (uint32_t)memory_ptr;
```

The heap is initialized during system startup with a size of `32KB` as defined in the linker script:

```
.heap :{
    . = ALIGN(4);
    _sheap = .;
    . = . + 32K;
    *(.heap)
    . = ALIGN(4);
    _eheap = .;
} >SRAM
```

4 Task Scheduling

Task scheduling is a fundamental aspect of real-time operating systems (RTOS), ensuring that multiple tasks execute efficiently while maintaining system responsiveness. This section provides an in-depth analysis of scheduler initialization, task states, context switching, and various scheduling triggers.

4.1 Scheduler Initialization

The scheduler serves as the central mechanism responsible for managing task execution. During system initialization, the scheduler is set up to organize task priorities, manage execution queues, and allocate system resources. Initialization typically involves configuring system data structures, setting up task control blocks (TCBs), and enabling necessary hardware timers to facilitate periodic scheduling events.

4.2 Task States and Queue

Each task in the system transitions through different states, typically including *Ready*, *Running*, *Blocked*, and *Terminated*. The scheduler maintains a queue to manage these state transitions:

- **Ready State:** Tasks waiting for CPU execution reside in this state. The scheduler selects one based on the priority scheduling policy.
- **Running State:** The currently executing task occupies this state until it completes execution or is preempted.
- **Blocked State:** Tasks awaiting external events (e.g., I/O completion, synchronization primitives) transition to this state until the required condition is met.
- **Terminated State:** Once execution is completed, a task moves to the terminated state, freeing allocated resources.

```c
volatile uint32_t QUEUE_SIZE_P = 3;
volatile uint32_t CURR_TASK_P = 0;
volatile uint16_t TASK_ID = 1000;

volatile TCB_TypeDef READY_QUEUE[MAX_QUEUE_SIZE_P];
```

4.3 Context Switching via PendSV

Efficient task scheduling requires seamless context switching, which is achieved via the *PendSV* (Pendable Service Call) exception. PendSV is a low-priority interrupt used by the RTOS to trigger context switches when task execution needs to be altered. The context switch process involves saving the current task's execution state (registers, stack pointer) and restoring the state of the next scheduled task. This mechanism ensures minimal overhead and maintains task continuity without system instability.

```c
void __attribute__((naked)) PendSV_Handler(void)
{
    // Clear all pending interrupts
    SCB->ICSR |= (1 << 27);

    // kprintf("PendSV_Handler\n");

    // save current context
    if (READY_QUEUE[CURR_TASK_P].status == RUNNING)
    {
        READY_QUEUE[CURR_TASK_P].status = READY;
        asm volatile(
            "mrs r0, psp\n"
            "isb 0xf\n"
            "stmdb r0!, {r4-r11}\n");

        asm volatile("mov %0, r0\n"
                     : "=r"(READY_QUEUE[CURR_TASK_P].psp
                         )
                     :);
    }
    __DSB();
    __ISB();
    /*---------------------------------------------*/

    uint32_t chosen_task = MAX_QUEUE_SIZE_P;
    uint32_t count = 0;

    for (int i = (CURR_TASK_P + 1) % QUEUE_SIZE_P;; i =
        (i + 1) % QUEUE_SIZE_P)
    {
        if (READY_QUEUE[i].status == READY)
        {
            chosen_task = i;
            break;
        }
        count++;
        if (count >= MAX_QUEUE_SIZE_P)
        {
            break;
        }
    }

    if (chosen_task == 5)
    { // finished
        __set_pending(0);
        asm volatile("bl sleep_state");
    }
    else
    {
        CURR_TASK_P = chosen_task;
    }

    __DSB();
    __ISB();

    asm volatile(
        "mov r0, %0"
        :
        : "r"((uint32_t)READY_QUEUE[CURR_TASK_P].psp));

    READY_QUEUE[CURR_TASK_P].status = RUNNING;

    asm volatile(
        "ldmia r0!,{r4-r11}\n"
        "msr psp, r0\n"
        "isb 0xf\n"
        "bx lr\n");
}
```

4.4 Scheduling Triggers

Scheduling events can be triggered through multiple mechanisms, allowing for preemptive or cooperative multitasking. The key triggers include voluntary task yield, system timer interrupts, and task termination events.

4.4.1 Voluntary Yield

A task can voluntarily relinquish CPU control by invoking a yield function. This mechanism allows cooperative multitasking where a running task explicitly informs the scheduler that it can switch to another task of equal or higher priority. Voluntary yielding is particularly useful in situations where a task completes its critical execution phase and wishes to allow other tasks to progress.

```
void yield(void)
{
    __ISB();

    asm volatile("PUSH {r4-r11}");
    asm volatile("svc %0" : : "i"(SYS_yield));
    asm volatile("POP {r4-r11}");
    __ISB();
}
```

4.4.2 System Timer

The system timer, commonly implemented via a periodic tick interrupt (e.g., SysTick in ARM Cortex-M), acts as a fundamental scheduling trigger. Each timer tick updates the system time and invokes the scheduler to evaluate if a task switch is necessary. This mechanism enables time-sliced scheduling and ensures that time-critical tasks receive appropriate CPU attention.

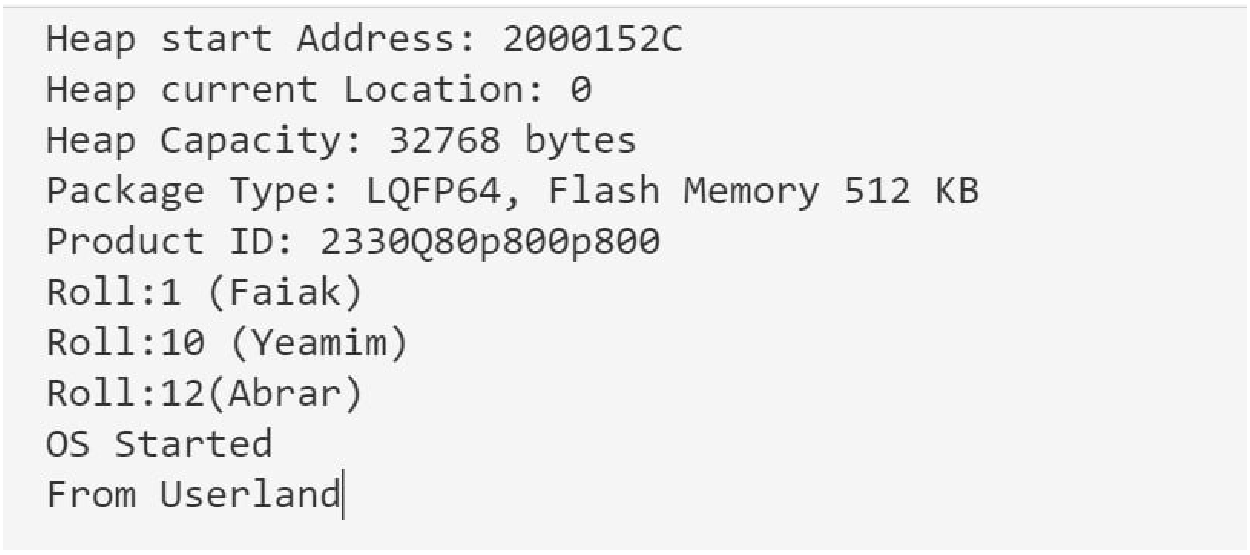
4.4.3 Task Termination

When a task completes execution, it must be properly terminated to release allocated resources. The scheduler detects task completion events and removes the task from the active scheduling queue. In dynamic task environments, terminated tasks may also trigger memory deallocation routines or notify dependent tasks, ensuring efficient resource utilization and preventing memory leaks.

```
void task_exit(void)
{
    // READY_QUEUE[CURR_TASK_P].status = KILLED;

    __ISB();

    TCB_TypeDef *tcb = READY_QUEUE + CURR_TASK_P;
    __asm volatile(
        "MOV R2, %0\n"
        :
        : "r"(tcb));
    asm volatile("PUSH {r4-r11}");
    asm volatile("svc %0" : : "i"(SYS__exit));
    asm volatile("POP {r4-r11}");

    kprintf("task exited : %d\n", READY_QUEUE[
       CURR_TASK_P].task_id);

    __DSB();
    __ISB();

    yield();
}
```

5 Results

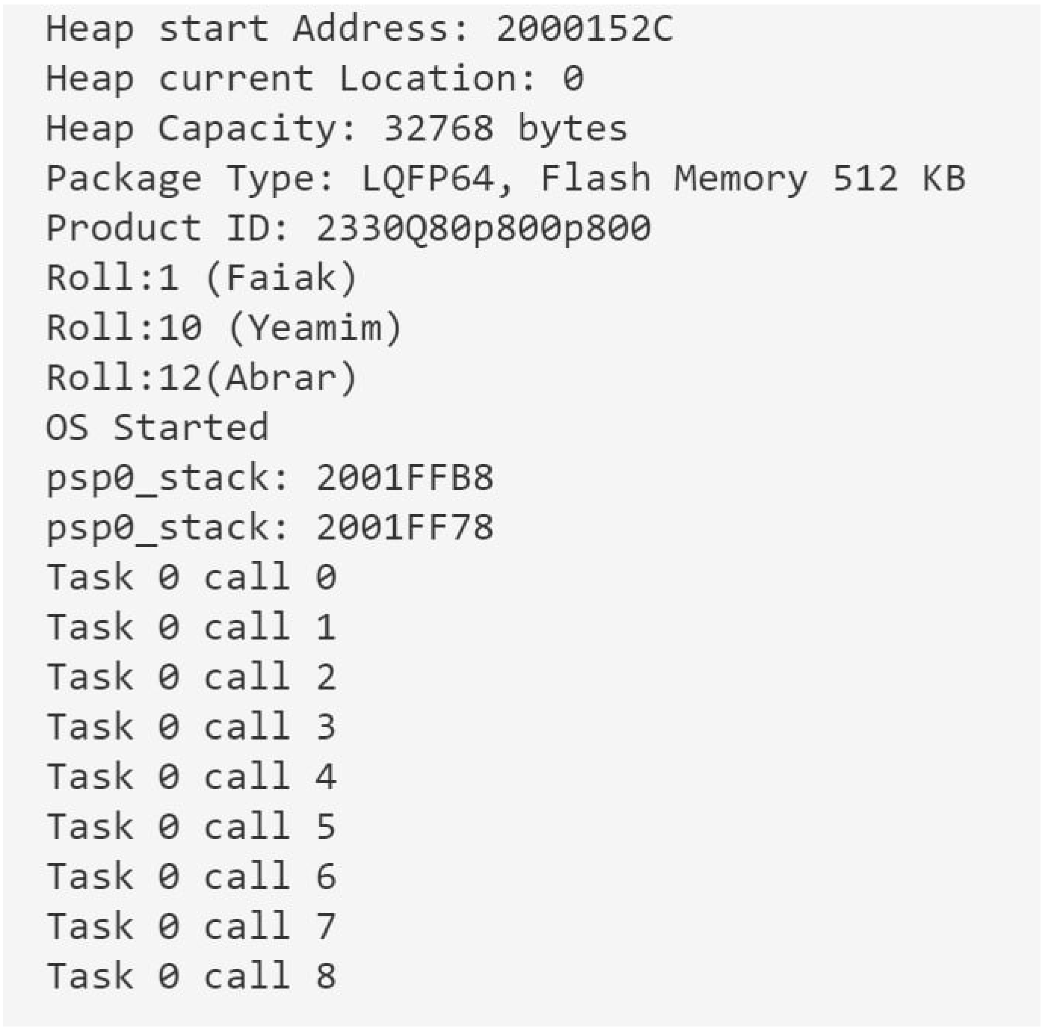
5.1 Syscall execution from Userland



```
Heap start Address: 2000152C
Heap current Location: 0
Heap Capacity: 32768 bytes
Package Type: LQFP64, Flash Memory 512 KB
Product ID: 2330Q80p800p800
Roll:1 (Faiak)
Roll:10 (Yeamim)
Roll:12(Abrar)
OS Started
From Userland
```

Figure 2: Syscall execution

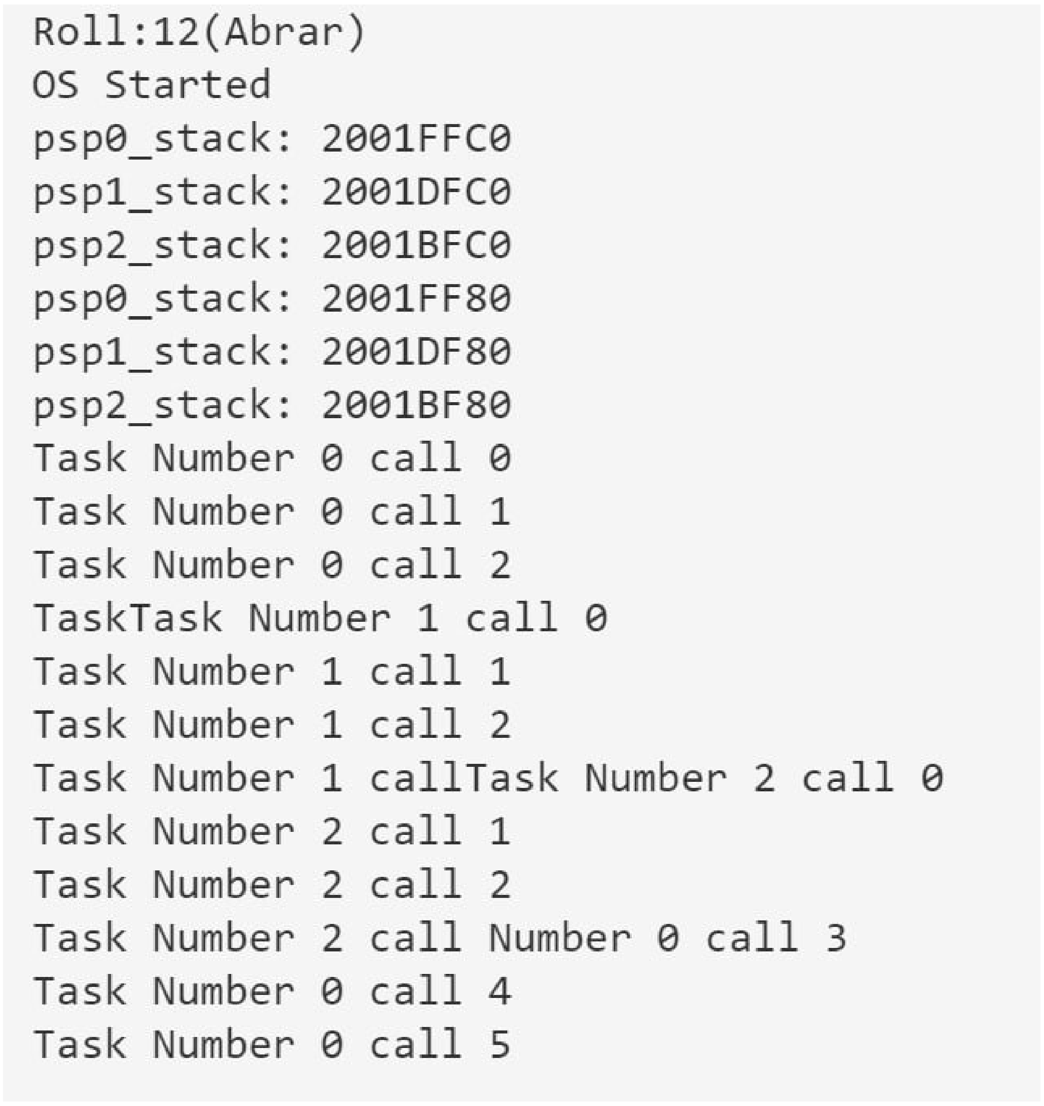
5.2 Initialize task



```
Heap start Address: 2000152C
Heap current Location: 0
Heap Capacity: 32768 bytes
Package Type: LQFP64, Flash Memory 512 KB
Product ID: 2330Q80p800p800
Roll:1 (Faiak)
Roll:10 (Yeamim)
Roll:12(Abrar)
OS Started
psp0_stack: 2001FFB8
psp0_stack: 2001FF78
Task 0 call 0
Task 0 call 1
Task 0 call 2
Task 0 call 3
Task 0 call 4
Task 0 call 5
Task 0 call 6
Task 0 call 7
Task 0 call 8
```

Figure 3: Initialize Task

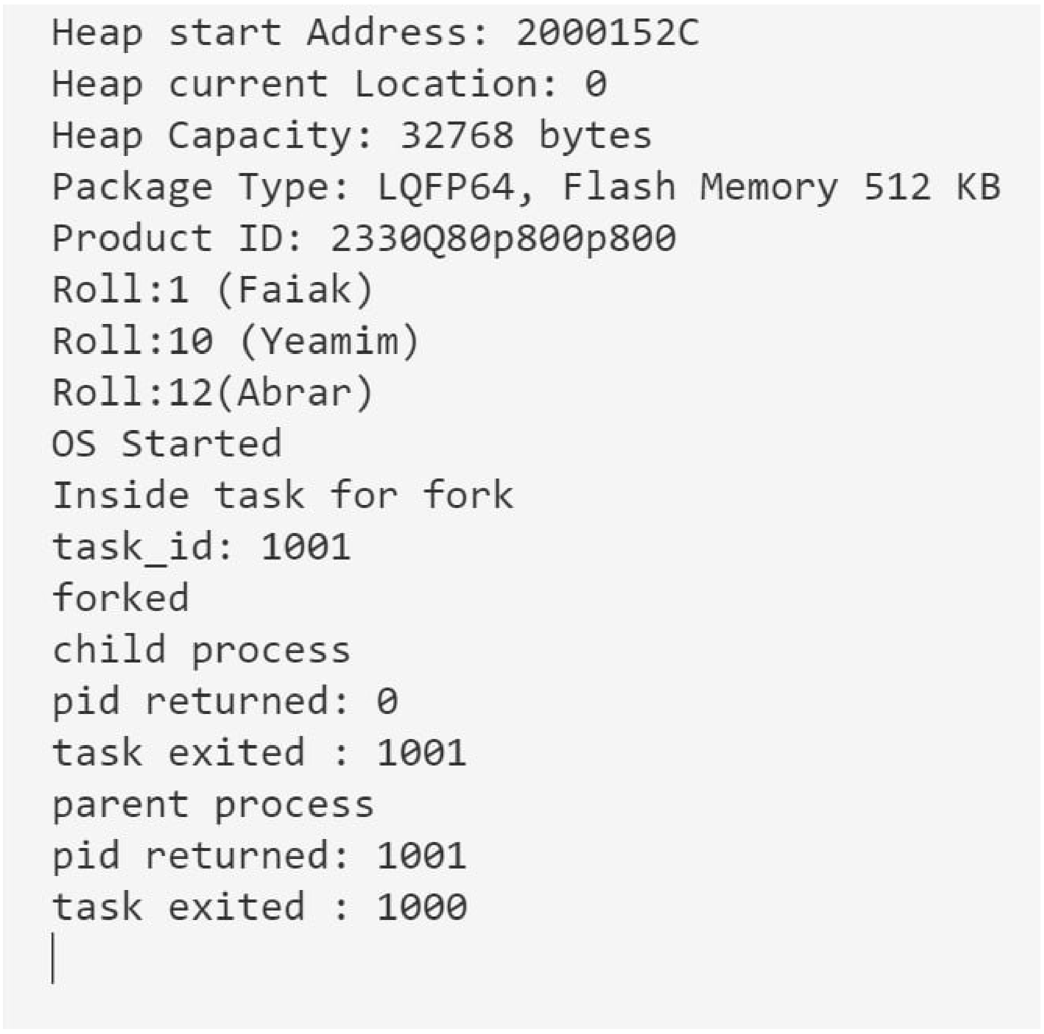
5.3 Round Robin Scheduler



```
Roll:12(Abrar)
OS Started
psp0_stack: 2001FFC0
psp1_stack: 2001DFC0
psp2_stack: 2001BFC0
psp0_stack: 2001FF80
psp1_stack: 2001DF80
psp2_stack: 2001BF80
Task Number 0 call 0
Task Number 0 call 1
Task Number 0 call 2
TaskTask Number 1 call 0
Task Number 1 call 1
Task Number 1 call 2
Task Number 1 callTask Number 2 call 0
Task Number 2 call 1
Task Number 2 call 2
Task Number 2 call Number 0 call 3
Task Number 0 call 4
Task Number 0 call 5
```

Figure 4: Round Robin Dispatcher

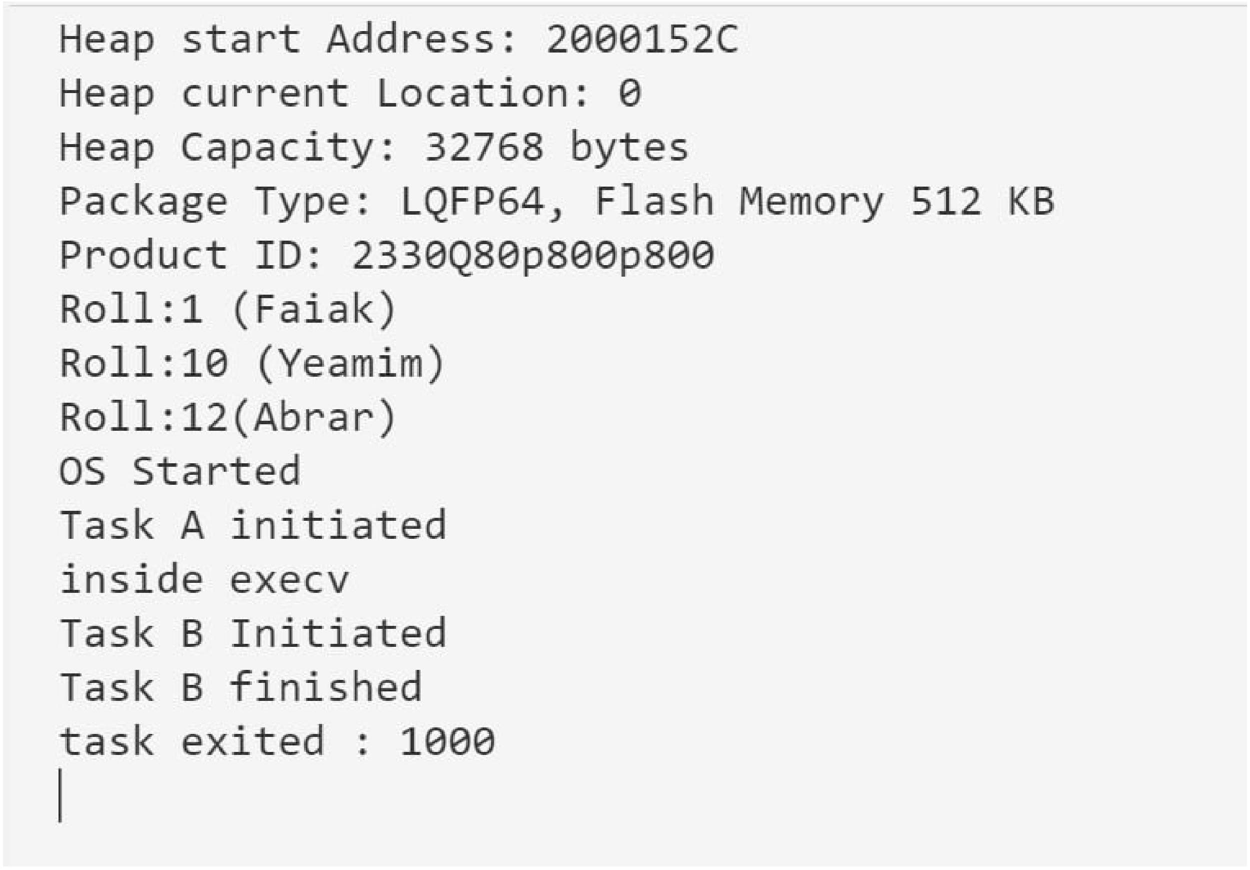
5.4 SYS_fork example



```
Heap start Address: 2000152C
Heap current Location: 0
Heap Capacity: 32768 bytes
Package Type: LQFP64, Flash Memory 512 KB
Product ID: 2330Q80p800p800
Roll:1 (Faiak)
Roll:10 (Yeamim)
Roll:12(Abrar)
OS Started
Inside task for fork
task_id: 1001
forked
child process
pid returned: 0
task exited : 1001
parent process
pid returned: 1001
task exited : 1000
```

Figure 5: Fork System Call

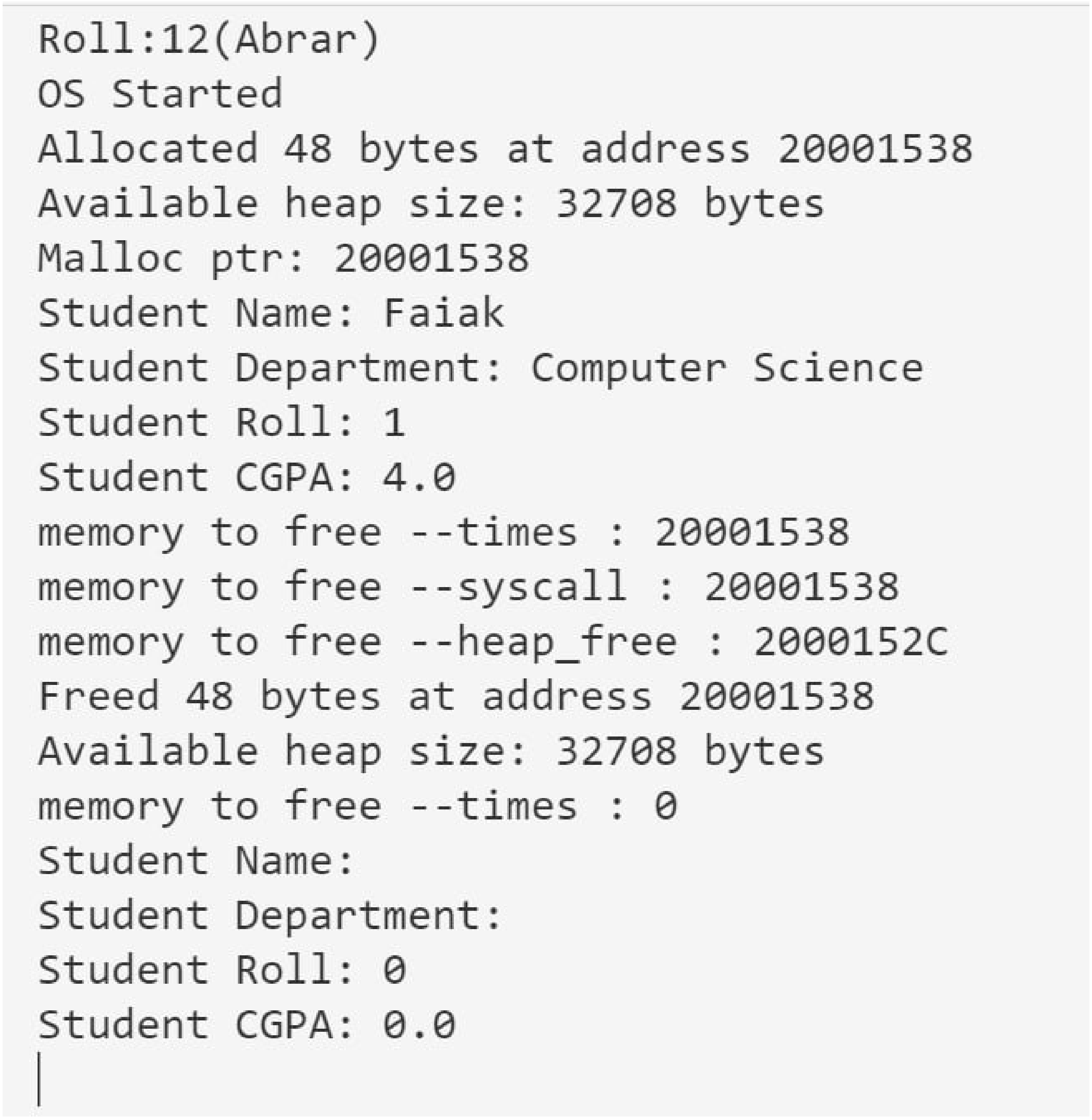
5.5 SYS_execv example



```
Heap start Address: 2000152C
Heap current Location: 0
Heap Capacity: 32768 bytes
Package Type: LQFP64, Flash Memory 512 KB
Product ID: 2330Q80p800p800
Roll:1 (Faiak)
Roll:10 (Yeamim)
Roll:12(Abrar)
OS Started
Task A initiated
inside execv
Task B Initiated
Task B finished
task exited : 1000
```

Figure 6: execv system call

5.6 System call functions example



```
Roll:12(Abrar)
OS Started
Allocated 48 bytes at address 20001538
Available heap size: 32708 bytes
Malloc ptr: 20001538
Student Name: Faiak
Student Department: Computer Science
Student Roll: 1
Student CGPA: 4.0
memory to free --times : 20001538
memory to free --syscall : 20001538
memory to free --heap_free : 2000152C
Freed 48 bytes at address 20001538
Available heap size: 32708 bytes
memory to free --times : 0
Student Name:
Student Department:
Student Roll: 0
Student CGPA: 0.0
```

Figure 7: System calls Functions

6 Applying the Patch to DUOS

6.1 Procedure for Patch Application

In order to apply the necessary modifications to the DUOS system, users are required to download the relevant DUOS repository and execute the patching command. The patch file, named `assignment.patch`, should be placed in the same directory as the DUOS source code. The patch can be applied using the following command in a Unix-based terminal or Windows-compatible environment:

```
patch -p1 -d duos24 < assignment.patch
```

It is crucial to ensure that the DUOS directory exists and is accessible before executing the command.

6.2 Platform-Specific Considerations

The provided patch file has been designed primarily for **Windows-based** environments. However, for users working on alternative operating systems, additional modifications may be required. Specifically, users must configure the correct path for the **OpenOCD** debugger in the system's `makefile`. The relevant configuration file can be found at:

```
src/compile/makefile
```

Following the successful application of the patch, users must execute the appropriate command based on their operating system:

- For **Linux** environments:
  `make run`