University of Dhaka

Department of Computer Science and Engineering

CSE 3211: Operating Systems Lab

Lab Report 3

Design and Deployment of System Calls, Thread Creation, and Multi-tasking Scheduling in DUOS

Submitted By:

Name: Md. Sakib Ur Rahman **Roll No:** 37

Name: Md. Shoriful Islam Rayhan **Roll No:** 41

Submitted On: March 18, 2025

Submitted To:

Dr. Mosaddek Hossain Kamal
Professor, Department of CSE, University of Dhaka

Contents

Chapter 1

Introduction

1.1 Overview

This lab assignment involves designing and implementing system calls, thread management, and multitasking scheduling for the DUOS operating system. Efficient resource allocation and task execution are critical for ensuring optimal performance and system stability. This report explores the architecture of the syscall interface, inter-process communication mechanisms for task scheduling, and advanced memory management techniques that support dynamic task execution.

1.2 Objectives

The goal of this lab is to,

- Implement OS service calls using SVC and SVCPend for system-level operations.
- Deploy heap and stack memory management, supporting system services: SYS_open, SYS_read, SYS_write, SYS_malloc, SYS_free, SYS_fork, SYS_execv, SYS_getpid, SYS_exit, and SYS_yield.
- Design and integrate kernel services for dynamic thread and process creation in RAM, incorporating scheduling techniques like time-sharing and FCFS, and analyze their efficiency.

Chapter 2

System call and Task Scheduling

2.1 System call

In an operating system, system call is a way that an application requests a service from the kernel of the operating system. In ARM-based architecture, Supervisor Call (SVC) instruction is used to implement system call in operating system. It traps an exception that seitches the processor from unprivileged to privileged mode, that allows the kernel to execute the requested service. The system calls in operating system are SYS_exit, SYS_getpid, SYS_read, SYS_write, SYS_time, SYS_reboot, SYS_yield, SYS_malloc, SYS_free, SYS_fork, and SYS_execv.

2.2 Task Scheduling using SysTick and PendSV

In ARM-based systems, PendSV (Pendable Service Call) is an exception mechanism specifically designed for efficient context switching in task management. It facilitates deferred interrupt processing, making it well-suited for scheduling and switching tasks in a multi-tasking environment. Typically triggered by the SysTick timer, PendSV ensures smooth task transitions by handling context switching only after higher-priority interrupts have been serviced. When a switch is required, the SysTick handler sets the PendSV exception to pend. The PendSV handler then saves the current task's state, including its stack pointer and registers, and restores the state of the next scheduled task. This process updates the Task Control Block (TCB) and the ready queue, ensuring the seamless execution of multiple tasks.

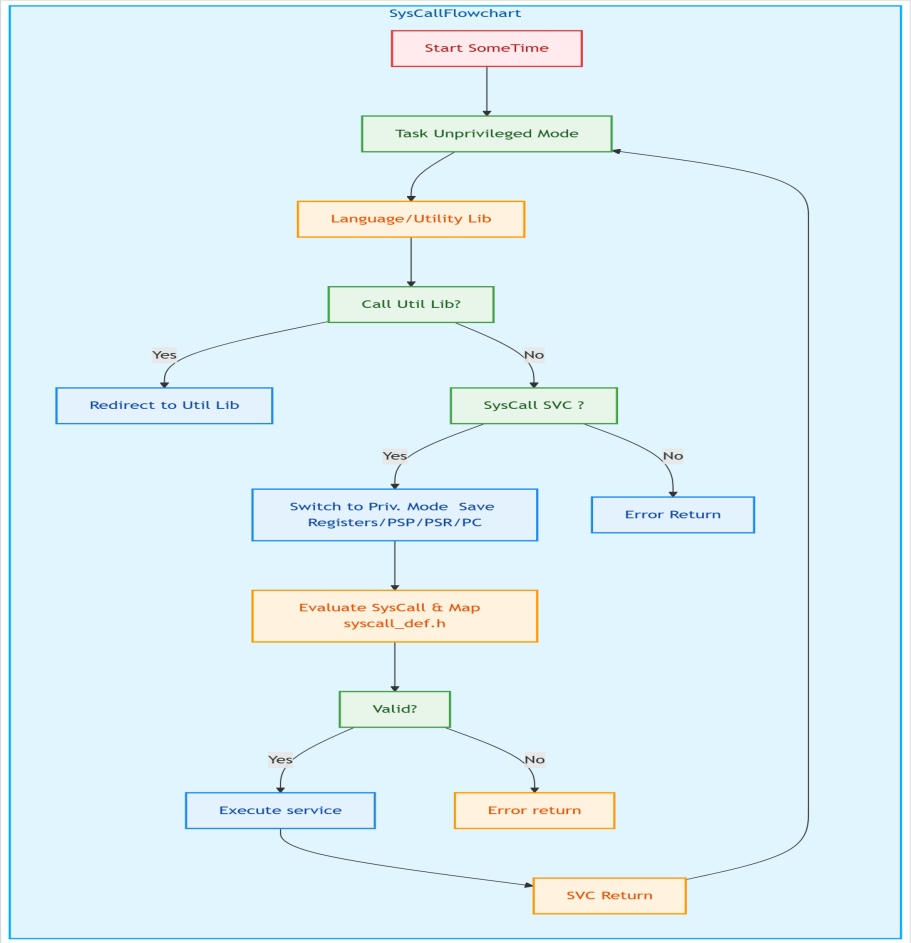


Figure 2.1: System call steps

Chapter 3

Implementation Details

3.1 Memory Management: Heap and Stack

3.1.1 Heap

The DUOS operating system employs a dynamic memory allocation system facilitated by the `SYS_malloc` and `SYS_free` system calls. The kernel manages the heap memory and provides controlled access to user applications through these system calls. This mechanism ensures efficient memory utilization while supporting dynamic allocation and deallocation.

The heap enables both the kernel and user applications to dynamically allocate memory through the following mechanisms:

1. **Heap Initialization:** The system configures the heap during startup, establishing an initial free memory block.
2. **Allocation Strategy:** A first-fit algorithm is employed to efficiently locate suitable memory blocks for allocation.
3. **Free List Management:** The system maintains a linked list of free memory blocks, optimizing reuse and minimizing fragmentation.
4. **Memory Alignment:** Allocations adhere to proper alignment constraints (e.g., 8-byte boundaries) to enhance performance and prevent access violations.

The heap structure is declared in the .heap section of the memory layout.

```
1 .heap :
2 {         . = ALIGN(4);
3           _sheap = .;
```

```
4           . = . + 32K;
5           *(.heap)
6           . = ALIGN(4);
7           _eheap = .;
8   }>SRAM
```

The heap is positioned after the .bss section in memory, which means it comes after all initialized and uninitialized global variables. This allocation ensures that the heap has a dedicated 32KB region in SRAM that doesn't overlap with other program data.

3.1.2 Stack

Each task in DUOS has its own stack space. When creating a new task, a stack is allocated and initialized with the appropriate context.

1. Stack Allocation: A dedicated stack area (typically 1024 words) is assigned to each task.

2. Stack Initialization: The stack is set up with the task's entry point and initial register values.

3. Stack Pointer Management: User tasks utilize the Process Stack Pointer (PSP), while kernel operations rely on the Main Stack Pointer (MSP).

```
1   void create_tcb(TCB_TypeDef *tcb, void (*func_ptr)(
        void), uint32_t *stack_start)
2   {
3       tcb->magic_number = MAGIC_NUMBER;
4       tcb->task_id = TASK_ID;
5       TASK_ID++;
6       tcb->status = READY;
7       tcb->execution_time = 0;
8       tcb->waiting_time = 0;
9       tcb->digital_sinature = DIGITAL_SIGNATURE;
10
11      tcb->psp = stack_start;
12      *(--tcb->psp) = 0x01000000;          // xPSR
13      *(--tcb->psp) = (uint32_t)func_ptr; // PC
14      *(--tcb->psp) = 0xFFFFFFFD;          // LR
15
16      for (uint32_t i = 0; i < 13; i++)
```

```
17      {
18          *(--tcb->psp) = 0;
19      }
20      __ISB();
21  }
```

3.2 Process Management Services

3.2.1 SYS_fork Implementation

The fork system call creates a new process by duplicating the current one. In the DUOS implementation:

- The stack pointer of the parent process (`parents_psp`) is utilized to retrieve the current execution context.
- A separate stack area (`psp_stack_for_child[1024]`) is allocated for the child process.
- The child process's Task Control Block (TCB) is initialized with a unique task ID:

```
1       tcb->magic_number = MAGIC_NUMBER;
2       tcb->task_id = TASK_ID;
3       TASK_ID++;
```

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

```
1       *(--tcb->psp) = parents_psp[7];  // xPSR
2       *(--tcb->psp) = (uint32_t)parents_psp[6]; //
            PC
3       *(--tcb->psp) = parents_psp[5];  // LR
```

- Different return values are set for parent and child:

```
1       parents_psp[0] = TASK_ID + 1;  // Parent
           returns child's PID
2       *(--tcb->psp) = 0;             // Child
           returns 0
```

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

3.2.2 SYS_execv Implementation

The execv system call replaces the current process image with a new one:

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

```
1    int find_file(char *filename){
2        for (uint32_t i = 0; i < file_count; i
             ++){
3            if (strcomp((uint8_t *)file_list[i].
                 name, (uint8_t *)filename) == 0){
4                return i;
5            }
6        }
7        return -1;
8    }
```

- The file system is a simple in-memory structure that maps file names to function pointers:

```
1    file_entry_t file1;
2    file1.address = (uint32_t *)file_B;
3    file1.size = 1024;
4    file1.mode = O_RDONLY;
5    strcopy((uint8_t *)file1.name, (const
         uint8_t *)"PRINT_B");
6    file_list[file_count++] = file1;
```

- The current process's stack is reconfigured with the entry point of the new program.
- The syscall handler processes the request by calling `__sys_execv()`:

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

```
5     svc_args[0] = val;
```

3.2.3 SYS_getpid Implementation

This straightforward service retrieves the task ID from the current task's TCB:

```
1 int pid = READY_QUEUE[CURR_TASK_P].task_id;
2 svc_args[0] = pid;
```

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

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

3.2.4 SYS_exit Implementation

The exit system call terminates the current process:

```
1 TCB_TypeDef *tcb_to_kill = (TCB_TypeDef *)svc_args
     [2];
2 tcb_to_kill->status = KILLED;
```

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

```
1 TCB_TypeDef *tcb = READY_QUEUE + CURR_TASK_P;
2 __asm volatile("MOV R2, %0\n" : : "r"(tcb));
```

After marking the task as killed, the function calls `yield()` to allow the scheduler to select a new task.

3.3 Memory Management Services

3.3.1 SYS_malloc Implementation

The malloc system call allocates memory from the heap:

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

The syscall handler allocates memory using `heap_malloc()`:

```
uint32_t size_m = svc_args[2];
void *ptr = heap_malloc(size_m);
svc_args[2] = (uint32_t)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
```

3.4 System Call Mechanism

All system services are implemented through the Supervisor Call (SVC) instruction, which generates an exception and transfers control to the kernel:

```
uint32_t callno = ((uint8_t *)svc_args[6])[-2];
```

The handler dispatches to the appropriate service function based on the call number:

```
switch (callno){
    case SYS_read:
        // Handle read
        break;
    case SYS_write:
        // Handle write
        break;
    // Other cases...
}
```

This mechanism ensures a secure transition between user mode and kernel mode, protecting system resources from unauthorized access.

3.5 Task States and Queue

DUOS maintains a ready queue for tasks:

```
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];
```

Tasks can be in different states:

- READY: Ready to run
- RUNNING: Currently executing
- WAITING: Waiting for an event
- KILLED: Terminated

3.6 Context Switching via PendSV

The PendSV handler performs context switching between tasks:

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

    // 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)
                     :);
    }

    // Find next ready task
    // ... task selection logic ...

```

```
23      // Load next task context
24      asm volatile(
25          "mov r0, %0"
26          :
27          : "r"((uint32_t)READY_QUEUE[CURR_TASK_P].psp
                ));
28
29      READY_QUEUE[CURR_TASK_P].status = RUNNING;
30
31      asm volatile(
32          "ldmia r0!,{r4-r11}\n"
33          "msr psp, r0\n"
34          "isb 0xf\n"
35          "bx lr\n");
36  }
```

This handler:

1. Saves the context of the currently running task (registers R4-R11)
2. Selects the next task to run based on a scheduling algorithm
3. Loads the context of the selected task
4. Updates the Process Stack Pointer (PSP) to point to the new task's stack
5. Returns to the new task

3.7 Scheduling Triggers

The scheduler can be triggered in several ways:

3.7.1 Voluntary Yield

A task can voluntarily give up the CPU using the `yield()` function:

```
1  void yield(void)
2  {
3      __ISB();
4      asm volatile("PUSH {r4-r11}");
5      asm volatile("svc %0" : : "i"(SYS_yield));
6      asm volatile("POP {r4-r11}");
```

```
7       __ISB();
8   }
```

3.7.2 System Timer

The SysTick timer can trigger context switches at regular intervals for preemptive multitasking.

3.7.3 Task Termination

When a task exits, the scheduler is triggered to select a new task:

```
1   void task_exit(void)
2   {
3       __ISB();
4       TCB_TypeDef *tcb = READY_QUEUE + CURR_TASK_P;
5       __asm volatile("MOV␣R2,␣%0\n" : : "r"(tcb));
6       asm volatile("PUSH␣{r4-r11}");
7       asm volatile("svc␣%0" : : "i"(SYS__exit));
8       asm volatile("POP␣{r4-r11}");
9       yield();
10  }
```

3.8 Scheduler Initialization

The scheduler is initialized in `init_scheduler()`:

```
1   void init_scheduler(void)
2   {
3       uint64_t psp0_stack[1024], psp1_stack[1024],
          psp2_stack[1024];
4
5       create_tcb(READY_QUEUE + CURR_TASK_P, (void *)
          task0, psp0_stack + 1024);
6       CURR_TASK_P = (CURR_TASK_P + 1) % QUEUE_SIZE_P;
7       create_tcb(READY_QUEUE + CURR_TASK_P, (void *)
          task1, psp1_stack + 1024);
8       CURR_TASK_P = (CURR_TASK_P + 1) % QUEUE_SIZE_P;
9       create_tcb(READY_QUEUE + CURR_TASK_P, (void *)
          task2, psp2_stack + 1024);
```

```
10      CURR_TASK_P = (CURR_TASK_P + 1) % QUEUE_SIZE_P;
11
12      set_pending(1);
13      READY_QUEUE[CURR_TASK_P].status = RUNNING;
14      start_task((uint32_t)(READY_QUEUE[CURR_TASK_P].
            psp));
15  }
```

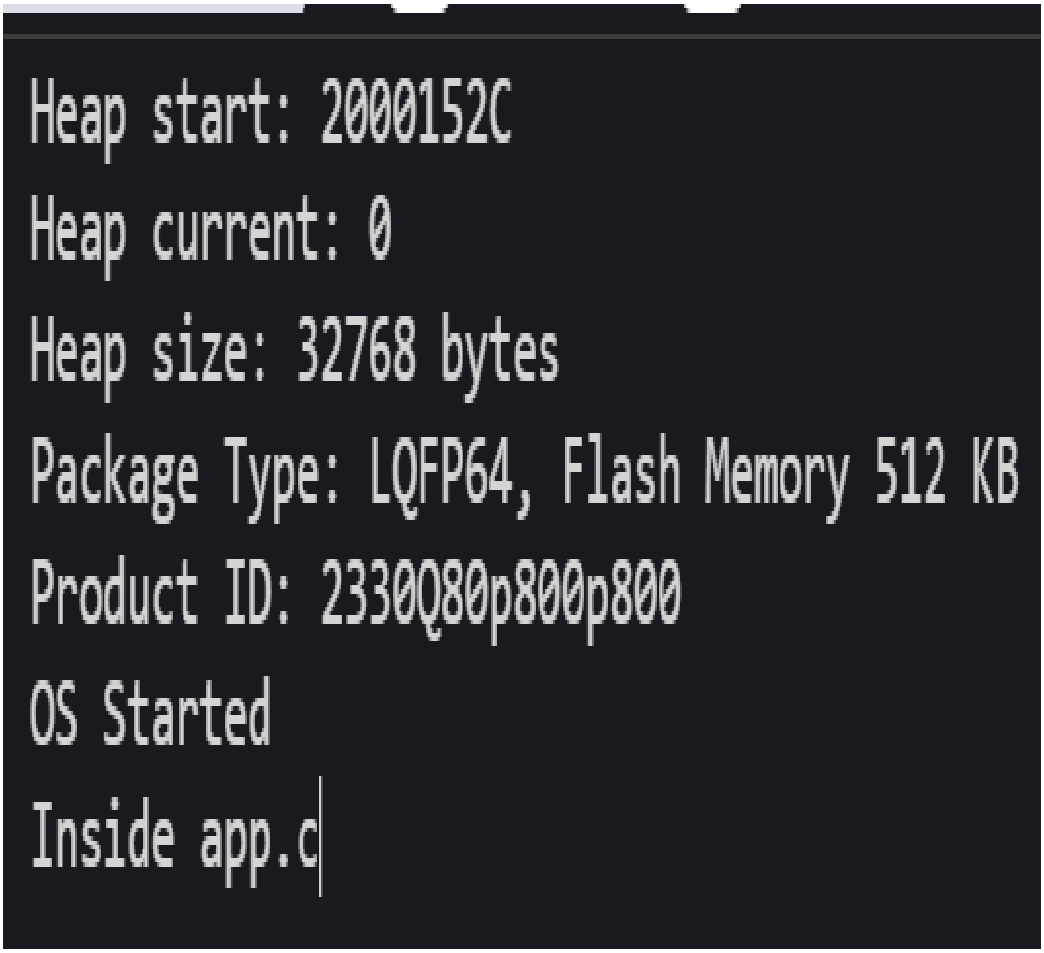
This function:

1. Allocates stacks for initial tasks
2. Creates TCBs for each task
3. Sets up the ready queue
4. Triggers the scheduler to start the first task

Chapter 4

Output

4.1 Syscall execution from App.c



4.2 Initialize task

```
Heap start: 2000152C
Heap current: 0
Heap size: 32768 bytes
Package Type: LQFP64, Flash Memory 512 KB
Product ID: 2330Q80p800p800
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
Task 0 call 9
Task 0 call 10
Task 0 call 11
Task 0 call 12
Task 0 call 13
Task 0 call 14
```

4.3 Round Robin Scheduler

```
Heap start: 2000152C
Heap current: 0
Heap size: 32768 bytes
Package Type: LQFP64, Flash Memory 512 KB
Product ID: 2330Q80p800p800
OS Started
psp0_stack: 2001FFC0
psp1_stack: 2001DFC0
psp2_stack: 2001BFC0
psp0_stack: 2001FF80
psp1_stack: 2001DF80
psp2_stack: 2001BF80
Task 0 call 0
Task 0 call 1
Task 0 call 2
Task 0 call Task 1 call 0
Task 1 call 1
Task 1 call 2
Task 1 call 3
Task 1 call 4
Task 1 callTask 2 call 0
Task 2 call 1
Task 2 call 2
```

4.4 SYS_fork example

```
Heap start: 2000152C
Heap current: 0
Heap size: 32768 bytes
Package Type: LQFP64, Flash Memory 512 KB
Product ID: 2330Q80p800p800
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
```

4.5 SYS_execv example

```
Heap start: 2000152C
Heap current: 0
Heap size: 32768 bytes
Package Type: LQFP64, Flash Memory 512 KB
Product ID: 2330Q80p800p800
OS Started
A started
inside execv
B started
B finished
task exited : 1000
```

4.6 System call functions example

```
Heap start: 2000152C
Heap current: 0
Heap size: 32768 bytes
Package Type: LQFP64, Flash Memory 512 KB
Product ID: 2330Q80p800p800
OS Started
Allocated 32 bytes at address 20001538
Available heap size: 32724 bytes
Malloc ptr: 20001538
Student ID: 1
Student CGPA: 3.500000000
memory to free --times : 20001538
memory to free --syscall : 20001538
memory to free --heap_free : 2000152C
Freed 32 bytes at address 20001538
Available heap size: 32724 bytes
memory to free --times : 0
Student ID: 0
Student CGPA: 0.0
Allocated 32 bytes at address 20001538 from freelist
Available heap size: 32700 bytes
Malloc ptr: 20001538
ptr: 20001538
```

Chapter 5

User operational guidelines

5.1 How to apply the patch

The user needs to download the trivial DUOS and apply the provided patch using the following command:

```
patch -p1 -d duos24 < 37_41_lab-3_duos24_patch.diff
```

Both the DUOS folder and the patch file should be in the same directory.

Important Note

The provided patch file is designed for Windows operating systems. For other operating systems, users must specify the path of OpenOCD according to their system in \src\compile\makefile.
After applying the patch, users should run: `make run` for **Linux** operating systems, or `make run_w` for **Windows** operating systems.

Bibliography

[1] STMicroelectronics, *STM32F4xx Reference Manual*.