



UNIVERSITY OF DHAKA

Department of Computer Science and Engineering

CSE:3211 Operating Systems Lab

Lab Report : Design and Deploy Syscall, Thread
Creation in RAM and Multi-tasking Schedule for DUOS

Submitted By:

Name : Sarowar Jahan Rafin

Roll No : 09

Name : Abdullah Al Mahmud

Roll No : 15

Name : Md. Sadman Sakib

Roll No : 49

Submitted On:

March 19, 2025

Submitted To:

Dr. Mosaddek Hossain Kamal, Professor

Contents

1	Introduction	3
1.1	Overview	3
1.2	Objectives	3
2	System call and Task Scheduling	4
2.1	System call	4
2.2	Task Scheduling using SysTick and PendSV	4
3	Implementation Details	6
3.1	Memory Management: Heap and Stack	6
3.1.1	Heap	6
3.1.2	Stack	7
3.2	Process Management Services	8
3.2.1	SYS_fork Implementation	8
3.2.2	SYS_execv Implementation	9
3.2.3	SYS_getpid Implementation	10
3.2.4	SYS_exit Implementation	10
3.3	Memory Management Services	10
3.3.1	SYS_malloc Implementation	10
3.4	System Call Mechanism	11
3.5	Task States and Queue	11
3.6	Context Switching via PendSV	12
3.7	Scheduling Triggers	13
3.7.1	Voluntary Yield	13
3.7.2	System Timer	14
3.7.3	Task Termination	14
3.8	Scheduler Initialization	14
4	Output	16
4.1	Syscall execution from App.c	16
4.2	Initialize task	17

4.3	Round Robin Scheduler	18
4.4	SYS_fork example	19
4.5	SYS_execv example	20
4.6	System call functions example	21
5	User operational guidelines	22
5.1	How to apply the patch	22

Chapter 1

Introduction

1.1 Overview

This task centers on creating and implementing system calls, thread initiation, and multitasking scheduling within the DUOS operating system. Effective management of system resources and task execution is crucial to ensure optimal performance and dependability. The report outlines the syscall interface's structure, the communication methods for task scheduling, and the memory management techniques employed to facilitate dynamic task execution.

1.2 Objectives

The objective of this lab is to,

- Implement OS service calls utilizing SVC and SVCPend
- Configure Heap and Stack memory alongside services such as: SYS_open, SYS_read, SYS_write, SYS_malloc, SYS_free, SYS_fork, SYS_execv, SYS_getpid, SYS_exit, SYS_yield, and SYS_yield
- Design and develop the mentioned kernel services to create new threads and processes in RAM, and establish scheduling mechanisms like time sharing and FCFS for comparison

Chapter 2

System call and Task Scheduling

2.1 System call

In an operating system, a system call serves as a mechanism for an application to request a service from the operating system's kernel. Within ARM-based architectures, the Supervisor Call (SVC) instruction is employed to implement system calls. This instruction triggers an exception that transitions the processor from unprivileged mode to privileged mode, enabling the kernel to perform the requested operation. Examples of system calls in an operating system include `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

PendSV (Pendable Service Call) serves as an exception mechanism in ARM-based systems and is utilized for context switching in task management. It is specifically designed for deferred interrupt handling, making it well-suited for managing task scheduling and transitions in a multi-tasking environment. The SysTick timer, which generates regular interrupts, typically initiates PendSV. During a required context switch, the SysTick handler triggers the PendSV exception. Once all higher-priority interrupts are processed, the PendSV handler is executed. This handler preserves the current task's state, including its stack pointer and registers, and restores the state of the next task to be executed. The process updates the Task Control Block (TCB) and the ready queue to reflect the context of the newly selected task.

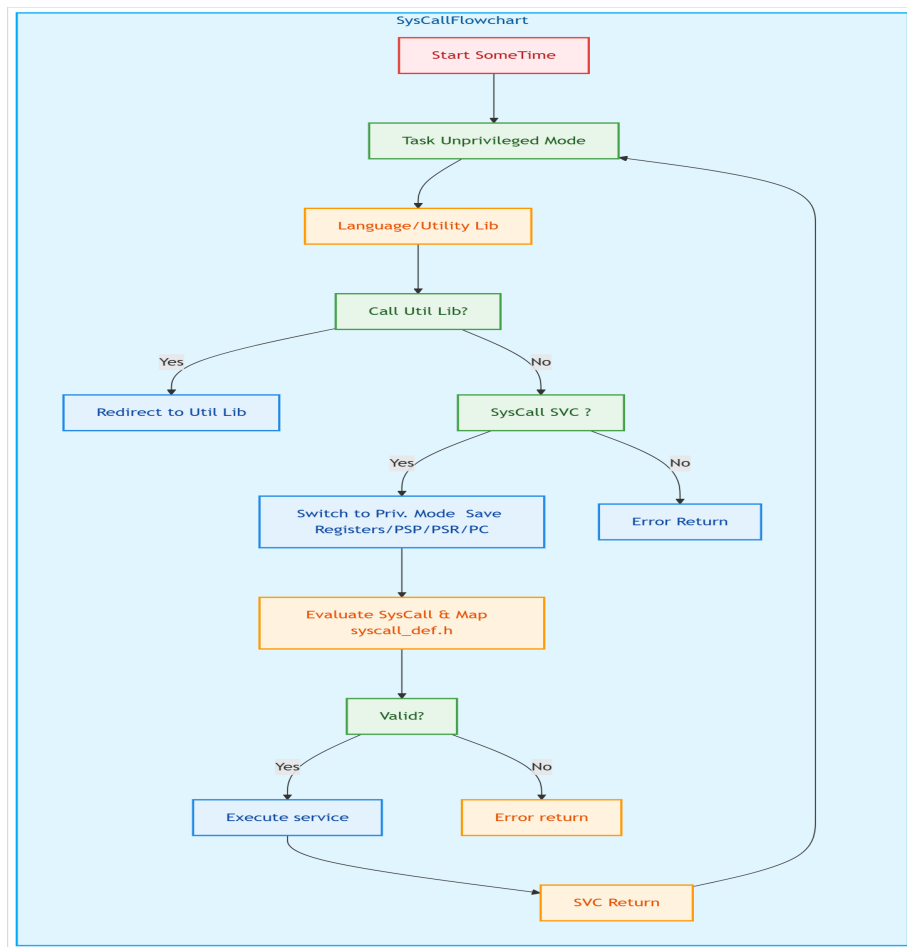


Figure 2.1: System call steps

Chapter 3

Implementation Details

3.1 Memory Management: Heap and Stack

3.1.1 Heap

The DUOS operating system supports dynamic memory allocation through the `SYS_malloc` and `SYS_free` system calls. These system calls enable the kernel to manage heap memory, which is made accessible to user applications.

The heap offers dynamic memory allocation for both the kernel and user applications with the following features:

1. **Heap Initialization:** The system configures the heap during startup by establishing the initial free memory block.
2. **Allocation Strategy:** Implements a first-fit algorithm to identify appropriate memory blocks for allocation.
3. **Free List Management:** Keeps track of available memory blocks to optimize reuse.
4. **Memory Alignment:** Ensures memory allocations adhere to proper alignment (8-byte boundaries).

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

```
1 .heap :  
2 {  
3     . = ALIGN(4);  
4     _sheap = .;  
5     . = . + 32K;  
6     *(.heap)  
7     . = ALIGN(4);
```

```

7 |         _ehheap = .;
8 | }>SRAM

```

The heap is located immediately after the .bss section in memory, following both initialized and uninitialized global variables. This arrangement ensures that a 32KB region in SRAM is exclusively allocated to the heap, preventing any overlap with other program data.

3.1.2 Stack

In DUOS, each task is provided with its own stack space. During the creation of a new task, the system allocates and initializes a stack with the relevant 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 states.
3. Stack Pointer Management: The Process Stack Pointer (PSP) is utilized for user tasks, while the Main Stack Pointer (MSP) is reserved for kernel-level operations.

```

1 | void create_tcb(TCB_TypeDef *tcb, void (*func_ptr)(
2 | void), uint32_t *stack_start)
3 | {
4 |     tcb->magic_number = MAGIC_NUMBER;
5 |     tcb->task_id = TASK_ID;
6 |     TASK_ID++;
7 |     tcb->status = READY;
8 |     tcb->execution_time = 0;
9 |     tcb->waiting_time = 0;
10 |    tcb->digital_signature = DIGITAL_SIGNATURE;
11 |
12 |    tcb->psp = stack_start;
13 |    *(&tcb->psp) = 0x01000000;           // xPSR
14 |    *(&tcb->psp) = (uint32_t)func_ptr;   // PC
15 |    *(&tcb->psp) = 0xFFFFFFF0;           // LR
16 |
17 |    for (uint32_t i = 0; i < 13; i++)
18 |    {

```



```

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 parent process's stack pointer (`parents_psp`) is used to access the current execution context.
- A new stack area is allocated for the child process (`psp_stack_for_child[1024]`).
- 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 (strcmp((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 = 0_RDONLY;
5  strcpy((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 R0, 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, R0\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, R0\n" : : "r"(size));
```

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

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

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

```
1 .heap :{  
2     . = ALIGN(4);  
3     _sheap = .;  
4     . = . + 32K;  
5     *(.heap)  
6     . = ALIGN(4);  
7     _eheap = .;  
8 } >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:

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

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

```
1 switch (callno){  
2     case SYS_read:  
3         // Handle read  
4         break;  
5     case SYS_write:  
6         // Handle write  
7         break;  
8     // Other cases...  
9 }
```

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:

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

```
4 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:

```
1 void __attribute__((naked)) PendSV_Handler(void)
2 {
3     // Clear all pending interrupts
4     SCB->ICSR |= (1 << 27);
5
6     // Save current context
7     if (READY_QUEUE[Curr_Task_P].status == RUNNING)
8     {
9         READY_QUEUE[Curr_Task_P].status = READY;
10        asm volatile(
11            "mrs r0, _psp\n"
12            "isb 0xf\n"
13            "stmdb r0!, {r4-r11}\n");
14
15        asm volatile("mov %0, r0\n"
16                    : "=r" (READY_QUEUE[Curr_Task_P]
17                        .psp)
18                    :);
19    }
20
21    // Find next ready task
22    // ... task selection logic ...
23
24    // Load next task context
25    asm volatile(
        "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 pre-emptive 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],
4         psp2_stack[1024];
5     create_tcb(READY_QUEUE + CURR_TASK_P, (void *)
6         task0, psp0_stack + 1024);
7     CURR_TASK_P = (CURR_TASK_P + 1) % QUEUE_SIZE_P;
8     create_tcb(READY_QUEUE + CURR_TASK_P, (void *)
9         task1, psp1_stack + 1024);
10    CURR_TASK_P = (CURR_TASK_P + 1) % QUEUE_SIZE_P;
11    create_tcb(READY_QUEUE + CURR_TASK_P, (void *)
12        task2, psp2_stack + 1024);
13    CURR_TASK_P = (CURR_TASK_P + 1) % QUEUE_SIZE_P;
14    set_pending(1);
15    READY_QUEUE[CURR_TASK_P].status = RUNNING;
```

```
14 |     start_task((uint32_t)(READY_QUEUE[Curr_Task_P].  
15 |         psp));  
    }
```

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

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

User need to download the trivial DUOS and apply the provided patch by `patch -p1 -d duos24 < assignment.patch`. Both should be in the same directory.

Important Note

Provided patch file is for **Windows** operating system. For different operating system, users have to write the path of **OpenOCD** of their respected system in `\src\compile\makefile`. After applying the patch user needs to write `make run` for **LINUX** operating system or `make run_w` for **WINDOWS** operating system.

Bibliography

- [1] STMicroelectronics, *STM32F4xx Reference Manual*.