Operating System Lab Assignment 02: Design and Deploy Syscall, use cases for System Call Test.

Dr. Mosaddek Tushar, Professor
Computer Science and Engineering, University of Dhaka,
Version 1.0
Demo Due Date: The following two weeks max (syscall).

October 08, 2025

Contents

1 Objectives and Policies

1.1 General Objectives

The objectives of the lab assignment are to understand and have hands-on training to understand

1.2 Assessment Policy

The assignment has three level objectives (i) primary objectives, (ii) advanced objectives, and (iii) optional boost objectives. Every student must complete the primary objective; however, they can attempt advanced and optional Boost-up objectives. The advanced objective will be a primary objective in the subsequent assignment. Further, you can achieve five (5) marks for completing the optional objectives and add these marks at the end of the semester with your total lab marks. However, you cannot get more than 100% assigned for the labs. The current lab does not contain advanced or optional boost objectives.

2 What to do?

- Deploy OS service call using SVC

  (a) **Explore the SVCall (Supervisor Call) Exception:** Investigate the operation of the `SVC` exception in ARM Cortex-M or similar architectures. Understand the mechanism by which the CPU switches from user mode to privileged kernel mode when an SVC instruction is executed. Examine how the processor automatically saves the current execution context—including registers, program counter (PC), and status registers—onto the stack, and how it uses the `EXC_RETURN` value to return to the exact instruction after the exception is serviced. This exploration helps in understanding the low-level mechanics of system call handling, exception prioritization, and context switching.

  (b) **Determine the System Call Number from the SVC Instruction:** Learn how the immediate value in the `SVC` instruction (for example, `SVC #25`) encodes the system call identifier. Trace how the exception handler extracts this value from the stack frame to determine which kernel service to invoke. Understanding this helps in mapping user-level function calls to the corresponding kernel routines, forming the basis of syscall dispatching.

  (c) **Understand User-to-Kernel Communication:** Study how arguments are passed from user applications to the kernel during a system call. Typically, registers or the stack are used to pass input parameters, and return values are delivered back to user space once the kernel service completes. Analyze the full path of execution, including entry into the SVC handler, service routine execution, and return to user mode. This step provides insight into how the operating system enforces security and controlled access to hardware resources while allowing applications to request services.

  (d) **Gain Hands-on Experience with OS Modes:** Observe the differences between user mode and kernel mode. Learn how user mode restricts access to privileged instructions and protected memory regions, and how kernel mode allows full access to system

resources. Practice switching between modes using SVC, and see firsthand how the CPU enforces protection boundaries, which is essential for building secure and stable operating systems.

(e) **Resource Management for System Calls:** Some system calls, such as `SYS__fork` or `SYS__open`, require careful management of kernel resources. You will need to create and maintain resource tables that track process control blocks (PCBs), open file descriptors, and memory allocations. Understand how heap memory is allocated and managed in kernel space to ensure that new processes and opened files do not overwrite existing kernel data. This task reinforces concepts of memory management, process lifecycle, and resource tracking.

(f) **Synchronization and Atomic Operations:** System calls like `SYS__sem_*` (semaphore operations) or mutex-related calls require atomic execution to prevent race conditions. You will implement or utilize kernel-level mechanisms to ensure that critical sections are executed without interruption. This may include disabling interrupts temporarily or using atomic CPU instructions. Understanding these concepts is crucial for developing concurrent, multi-tasking systems where shared resources must be protected to ensure data consistency and system stability.

(g) **Integrate and Test System Calls:** After implementing each system call, perform tests to verify correct argument passing, execution, and return values. For example, test `SYS__read` and `SYS__write` for terminal input/output, `SYS__fork` for process creation, and semaphore calls for proper synchronization. Observing the behavior in real scenarios helps consolidate understanding of how user-level requests translate into kernel actions and back.

(h) **Optional Extensions:** Advanced students may explore additional mechanisms, such as:

- Implementing system call tables and dynamic dispatching.
- Monitoring and logging SVC invocations for debugging.
- Exploring nested exceptions, prioritization, and preemption.
- Studying the interaction between SVC and other exceptions such as PendSV or SysTick.

2.1 Implementation of System Call Mechanism using Syscall Exception (SVC)

The following description envisions implementing SVC to enable unprivileged user applications to access kernel services. You must use the class lecture and programming manual to complete the assignment, and note that you must comprehend the underlying notion of the SVC.

2.1.1 Syscall using SVC

SVC is an integral part of the ARM processor and exception, enabling the user-to-kernel interface to deliver essential kernel services to the user program. Usually, user programs running in unprivileged mode cannot directly access the connected hardware devices, drivers, and many other processor resources. Therefore, the user program needs a way to get access to available services from the kernel in privileged mode. Almost all microprocessors have the features to facilitate access to kernel functions. ARM processors provide a particular instruction, 'SVC,' to manipulate an exception to

get into it. Generally, it is known as SysCall. However, you cannot set any interrupt or exception from other handlers or exceptions; in that case, it generates a HardFault. In this part of the assignment, you must implement syscalls to create an interface between the user program and kernel. Test it from the unprivileged user program.

2.1.2 Task Scheduling using SysTick and PendSV

Next, you must implement the PendSV services for switching between tasks. Every 10ms, the DUOS resumes or executes a task that remains in the task queue. The task queue preserves the task header or TCB containing the last addresses of the active task or task currently awaiting execution. In the operating system, the queue is called the ready queue. We will use the ready queue later in the scheduling assignment. For this part of the lab, you use it for switching between tasks. Therefore you require a kernel stack that will hold the ready queue with the most updated stack addresses of the tasks. The kernel acquires the stackframe address from the ready queue, redirects to the task stack, loads the registers' value, and jumps to the code to execute or resume the execution of a task. For the switching interval, you must use the SysTick exception handler configured to interrupt every 10ms. The SysTick handler (at the tail) initiates the PendSV for a context switch. Usually, we should keep the priority level low for SysTick and PendSV for the smooth operation of the other interrupt ISR. Note that the ISR routine, including SVC and PendSV, must be simple to avoid heavy-weight functions like 'kprintf' or 'kscanf.' The PendSV must perform the context switching to execute or resume the next task.

2.1.3 Syscalls and kernel services

You must implement the syscall using following three files

- 'syscall_def.h' contains the unique number for each syscall.
- 'syscall.h' contains all prototype or function definitions for the kernel privileged services
- 'syscall.c' includes implementing the function call defined in the 'syscall.h'. Moreover, the 'syscall.c' file consists of the syscall_def.h to get the requested syscall number. Nonetheless, the kernel incorporates various auxiliary files for defining and detailing the actual function tasks.

The 'syscall_def.h' in Appendix B contains the following syscall with unique number. You do not need to implement all the syscall listed here. However, you must implements (i) SYS__exit, (ii) SYS_getpid, (iii) SYS_read, (iv) SYS_write, (v) SYS___time, (vi) SYS_reboot, and (vii) SYS_yield. See the description of the kernel functions below.

i) SYS__exit: terminate a process and call a SYS_yield() function. Whenever an exit() calls, it will do two tasks (i) change the process state to 'terminated' and call a 'yield()' function. The 'yield' function activates the PendSV exception to run or resume the next task listed in the ready queue. No argument required in the exit function. However, 'exit' function must call the SVC with the appropriate kernel service ID listed in syscall_def.h.

ii) SYS_getpid: return task_id (given in TCB) of the current task. The application layer function or library function is 'getpid()' returns process or task ID by invoke SVC with the approriate 'SYS_getpid' service ID.

iii) SYS_read: takes exactly three arguments (i) file descriptor, (ii) input buffer, and (iii) size. The kunitstd.h (Appendix A) contains the default 'STDIN_FILENO' file descriptor when user application kscanf calls the SYS_read function. The size parameter defines the maximum input size in bytes. However, the termination character, such as '\n,' determines the input size. If there is no termination character, then the maximum input limit is 'size.' You may define the largest input size as 256 bytes. Beyond this, the read function will ignore the characters and takes precisely 256 bytes as the input. In the current setting SYS_read with the 'STDIN_FILENO' descriptor, use _USART_READ to read from the character terminal. The application layer library is 'read(fd,buff,size)'. The application library function call SVC with the argument listed before and the 'SYS_read' syscall ID given in syscall_def.h. Note that the first argument should be the service ID.

iv) SYS_write: This function takes exactly three arguments to write bytes to the targeted file. The arguments are (i) file descriptor, (ii) out buffer, and (iii) size. When print use this system to display the characters in the terminal (character terminal or display monitor), then the kprintf passes 'STDOUT_FILENO' as the file descriptor. In the current setting SYS_write with the 'STDOUT_FILENO' descriptor, use _USART_WRITE to send a string to the character terminal. The application calls the application/library layer function 'write(fd, buff, size)'; however, the write function uses the SVC to call kernel services 'SYS_write' argument beginning with a service ID and three given function arguments.

v) SYS_time: Returns the current elapsed time of systick in milli-second. The application layer function name is 'getSysTickTime()'. The getSysTickTime() service provides the elapsed time of the SysTick timer in milliseconds and is implemented using an SVC (Supervisor Call). When the application layer calls getSysTickTime(), it executes an SVC instruction with a designated service number, causing the processor to enter the SVC handler in privileged mode. The handler reads the SysTick counter and the overflow count, then calculates the elapsed time in milliseconds based on the SysTick reload value and system clock frequency. This approach allows unprivileged application code to safely access system timing information without directly manipulating hardware registers. The service is useful for task scheduling, timeouts, profiling, and logging, providing a controlled and secure method to obtain system time.

vi) SYS_reboot: This service call to reboot or restart the microcontroller. The application layer function name is 'reboot.' The 'reoobt' service can be implemented using an SVC (Supervisor Call) to transition from the application layer to privileged system mode for performing a controlled software reset. When the application calls reboot(), it executes an SVC instruction with a specific service number, causing the processor to enter the SVC handler in privileged mode. Inside the handler, interrupts are disabled, and the system reset is triggered via the SCB→AIRCR register by writing the key 0x5FA and setting the SYSRESETREQ bit. This safely restarts the MCU, resetting CPU and peripheral registers to default values while preserving Flash memory. The approach ensures that unprivileged application code can request a reset securely, providing a clean mechanism for error recovery, configuration updates, or firmware reloads.

vii) SYS_yield and yield The system must make the yield() function available for user-level execution. The user application may voluntarily call of yield() function to pass the CPU to the next task. In this case, the yield() directly syscall the SYS_yield. The yield function calls

SVC with the service ID. The the yield() function allows user-level applications to voluntarily relinquish the CPU to the next ready task, facilitating cooperative multitasking. When a user application calls yield(), it does not perform the context switch directly; instead, it executes a Supervisor Call (SVC) with a designated service ID corresponding to SYS_yield. The SVC instruction triggers the processor to enter the privileged SVC handler, where the kernel performs the actual task scheduling and context switch. This mechanism ensures that unprivileged code can request a CPU yield safely without directly accessing scheduler internals or hardware registers. The service enables cooperative multitasking, allows fair CPU sharing among tasks, and maintains system stability by controlling context switching through a secure, privileged interface.

viii) `SYS_sleep`: Suspends the execution of the current task for a specified duration in milliseconds. The argument is the delay time, and the task is placed into the blocked state until the timeout expires. The user-level function is `sleep(ms)`. The sleep(ms) function allows a task to suspend its execution for a specified duration in milliseconds. When a user-level application calls sleep(ms), the current task is moved into a blocked state and will not be scheduled until the specified timeout expires. Internally, this can be implemented via a Supervisor Call (SVC) to a system service that manages the task's delay. The SVC handler records the task's wake-up time based on the system tick (SysTick) or a timer, updates the task control block with the blocked status, and triggers the scheduler to switch execution to the next ready task. Once the timeout expires, the task is returned to the ready queue and resumes execution. This mechanism ensures precise timing delays, efficient CPU utilization, and safe task suspension without busy-waiting, maintaining system responsiveness and fairness among tasks.

ix) `SYS_fork`: Creates a new process by duplicating the calling process's context. The child process receives a unique `task_id` and inherits the parent's resources, except for certain registers. The fork() system call creates a new process by duplicating the calling process's context. When a user program invokes fork(), it triggers a Supervisor Call (SVC) to switch from user mode to kernel mode, where the kernel allocates a new Task Control Block (TCB) for the child process. Most of the parent's resources—such as program counter, stack pointer, general-purpose registers, memory mappings, and open file descriptors—are copied or shared, while certain fields like the task ID and the fork() return value are set uniquely for the child. The child is placed in the scheduler's ready queue, and both parent and child resume execution immediately after the fork() call, with the parent receiving the child's task ID and the child receiving zero. This mechanism ensures that the child process is an almost identical, yet independent, execution entity, while SVC safely manages the transition to privileged kernel operations for resource allocation and process initialization.

x) `SYS_wait`: Suspends the parent process until one of its child processes terminates. It returns the `task_id` of the completed child. The corresponding user function is `wait()`. The wait() system call allows a parent process to suspend its execution until one of its child processes terminates. When a user program calls wait(), it invokes a Supervisor Call (SVC) to switch from user mode to kernel mode, where the kernel examines the parent's child process list. If no child has terminated yet, the kernel marks the parent process as blocked and removes it from the ready queue, preventing it from being scheduled. Once a child process exits, the kernel updates its exit status and notifies the parent by placing it back into the ready queue. The kernel then returns the task ID of the terminated child to the parent and, optionally, the child's exit code. During this operation, the kernel also performs resource cleanup for the

child, including deallocating its memory, closing file descriptors if not shared, and removing the child's TCB. This mechanism ensures proper parent-child synchronization, prevents zombie processes, and allows the parent to retrieve the child's termination status safely, while SVC guarantees privileged access for managing kernel data structures and scheduling.

xi) SYS_open: Opens a file or device and returns a file descriptor. The arguments typically include the filename and access mode (read, write, or append). The user-level function is `open(path, mode)`. The open() system call is used to open a file or device and obtain a file descriptor that uniquely identifies the opened resource within a process. When a user program calls open(path, mode), it triggers a Supervisor Call (SVC) to switch from user mode to kernel mode, allowing the kernel to safely access filesystem and device management structures. The kernel first validates the pathname and access mode, checks permissions, and searches the filesystem or device table for the specified resource. If the file or device exists and access is permitted, the kernel allocates an entry in the process's file descriptor table, pointing to an open file object that tracks the file's metadata, current offset, access mode, and reference count. The kernel then returns the integer file descriptor to the user, which the process can use in subsequent system calls such as read(), write(), or close(). If the resource cannot be opened, the kernel returns an error code, ensuring controlled access and proper management of system resources.

xii) SYS_close: Closes a previously opened file descriptor and releases any associated resources. The user-level interface is `close(fd)`. The close() system call terminates access to a previously opened file descriptor and releases associated kernel resources. When a user program invokes close(fd), it triggers a Supervisor Call (SVC) to switch from user mode to kernel mode, giving the kernel privileged access to manipulate file and process data structures. The kernel first validates the file descriptor to ensure it is currently open and belongs to the calling process. It then decrements the reference count of the corresponding open file object; if the count reaches zero, the kernel frees the object, updates file metadata (such as offsets), flushes any pending buffers to storage, and releases memory or device-specific resources. Finally, the kernel removes the file descriptor entry from the process's file descriptor table, preventing further access. The close() call returns success to the user process or an error code if the descriptor was invalid, ensuring proper resource cleanup, data integrity, and system stability.

xiii) SYS_sem_wait: Decrements a semaphore. If the semaphore value becomes negative, the calling task is blocked until the semaphore is available. The user-level interface is `sem_wait(semid)`. The sem˙wait(semid) system call is used to decrement the value of a semaphore and implement synchronization between tasks. When a user task calls sem˙wait(semid), it triggers a Supervisor Call (SVC) to switch from user mode to kernel mode, allowing the kernel to safely access semaphore data structures. The kernel first validates the semaphore ID and then decrements its value atomically. If the resulting value is non-negative, the calling task continues execution immediately. However, if the value becomes negative, the kernel marks the calling task as blocked, places it in the semaphore's waiting queue, and removes it from the scheduler's ready queue. When another task signals the semaphore via sem˙post(), the kernel unblocks one of the waiting tasks, allowing it to resume execution. This mechanism ensures mutual exclusion, proper synchronization, and avoidance of race conditions, while atomic operations and kernel-managed queues guarantee that concurrent access to the semaphore remains consistent and safe.

xiv) SYS_sem_post: Increments a semaphore and, if any task is waiting, unblocks one of them. The

user-level function is `sem_post(semid)`. The sem˙post(semid) system call is used to increment the value of a semaphore and potentially unblock a waiting task, facilitating synchronization between concurrent processes or threads. When a user task calls sem˙post(semid), it triggers a Supervisor Call (SVC) to switch from user mode to kernel mode, granting privileged access to the kernel's semaphore data structures. The kernel first validates the semaphore ID and then atomically increments its value. If the new value is less than or equal to zero, this indicates that one or more tasks are blocked waiting on the semaphore, so the kernel selects a task from the semaphore's waiting queue, marks it as ready, and places it back into the scheduler's ready queue. If no tasks are waiting, the increment simply updates the semaphore count. This mechanism ensures mutual exclusion and correct task synchronization, allowing multiple tasks to coordinate access to shared resources without race conditions while maintaining system integrity through atomic operations.

xv) SYS_mutex_lock: Locks a mutex to ensure exclusive access to shared resources. If the mutex is already locked, the task is blocked. The user-level function is `mutex_lock(mutexid)`. The mutex˙lock(mutexid) system call is used to acquire a mutex to ensure exclusive access to shared resources in a multitasking system. When a user task calls mutex˙lock(mutexid), it triggers a Supervisor Call (SVC) to switch from user mode to kernel mode, allowing the kernel to safely manipulate the mutex data structure. The kernel first validates the mutex ID and checks its current state. If the mutex is unlocked, the kernel atomically sets it to locked and records the calling task as the owner, allowing it to proceed. If the mutex is already locked by another task, the kernel places the calling task into the mutex's waiting queue and marks it as blocked, removing it from the scheduler's ready queue. Once the mutex becomes available, the kernel unblocks one of the waiting tasks, granting it ownership. This mechanism ensures mutual exclusion, prevents race conditions, and maintains consistent access to shared resources, with atomic operations guaranteeing safe concurrent access even under high contention.

xvi) SYS_mutex_unlock: Releases a previously locked mutex, enabling other tasks to acquire it. The user-level function is `mutex_unlock(mutexid)`. The mutex˙unlock(mutexid) system call is used to release a previously locked mutex, allowing other tasks to acquire it and ensuring proper synchronization of shared resources. When a user task calls mutex˙unlock(mutexid), it triggers a Supervisor Call (SVC) to switch from user mode to kernel mode, giving the kernel privileged access to the mutex data structures. The kernel first validates the mutex ID and verifies that the calling task is the current owner of the mutex. It then atomically sets the mutex state to unlocked. If one or more tasks are blocked waiting on the mutex, the kernel selects a task from the mutex's waiting queue, marks it as ready, and assigns it ownership of the mutex, placing it into the scheduler's ready queue. This mechanism ensures mutual exclusion, prevents race conditions, and maintains system stability by coordinating access to shared resources, with atomic operations guaranteeing that concurrent attempts to lock or unlock the mutex are handled safely.

xvii) SYS_signal: Sends a signal or event to a target task or process to notify an asynchronous event. The user-level interface is `signal(taskid, sig)`.The signal(taskid, sig) system call is used to send a signal or event to a target task or process, allowing asynchronous notification of events such as interrupts, exceptions, or inter-process communication. When a user task invokes signal(taskid, sig), it triggers a Supervisor Call (SVC) to switch from user mode to kernel mode, giving the kernel privileged access to process management structures. The kernel first validates the target task ID and the signal number, then queues the signal in the

target task's signal or event table. If the target task is blocked waiting for that signal, the kernel immediately marks it as ready and places it back into the scheduler's ready queue. Otherwise, the signal remains pending until the task checks or handles it. This mechanism ensures asynchronous communication, proper task synchronization, and safe delivery of events, while the kernel enforces atomic operations to prevent race conditions when multiple tasks send or receive signals concurrently.

**Instructions and Recommendations:

You are encouraged to consult references, read documentation, and acquire additional knowledge to improve your solution. Apply what you learn to implement robust and efficient system calls, task management, and synchronization. Test your implementation thoroughly to gain practical experience with both user-level and kernel-level operations.

2.2 How does the syscall and context switch work from top to bottom?

After a reset or power-on, DUOS performs system initialization. This includes loading and initializing the required drivers, setting up necessary interrupts and exceptions, allocating task stacks and stack frames, and creating Task Control Block (TCB) entries in the ready queue. Each new task is marked with the status *new*. Once initialization is complete, DUOS switches the processor to user mode and starts executing the current task in the ready queue.

When a task requires kernel services, such as reading from input or writing to output, it uses the `SVC` (Supervisor Call) instruction to switch from user mode to privileged kernel mode. This allows the task to request service from the operating system safely. For example, if a user application calls the function `kprintf`, the internal `duprintf` routine first processes the data by converting it into a suitable string or character format. After processing, `duprintf` calls the library function `write`, passing the prepared arguments. In this case, the file descriptor (`fd`) is `STDOUT_FILENO`, which represents the standard output (terminal).

The `write` function then executes an `SVC` instruction to request the kernel service. In the SVC handler, the kernel saves any necessary registers and dispatches the request to the appropriate syscall function. The syscall function, defined in `syscall.h`, identifies the specific kernel routine to execute based on the syscall ID. The first argument of the `SVC` instruction is always the service ID (or syscall ID), which is specified in `syscall_def.h`.

Figure 1 illustrates these steps in the SVC/system call process. In this example, the `write` library function uses `SVC` to access the `SYS__write` kernel service, allowing the user application to output data via the kernel safely.

For example: 'duprintf'. For SVC follow slides in lecture 05 for detail

```
function duprintf(args ... ){
        string s = convertToSTR(args ...) //see your kprintf implementation
        return_code=call write(s);
}

function write(File_descriptor fd, char* s,size_t){
        //stacked the arguments
        //first argument must be SYS_write service ID
        SVC call for SYS_write
}

function syscall(){
```

```
        1. use switch case to determine the actual function
        based on SVC service ID (first argumant) and call the function
        2. Return with exec_return code
}
function SYS_write(args ..){
        do th actual job with the USART driver
}
```

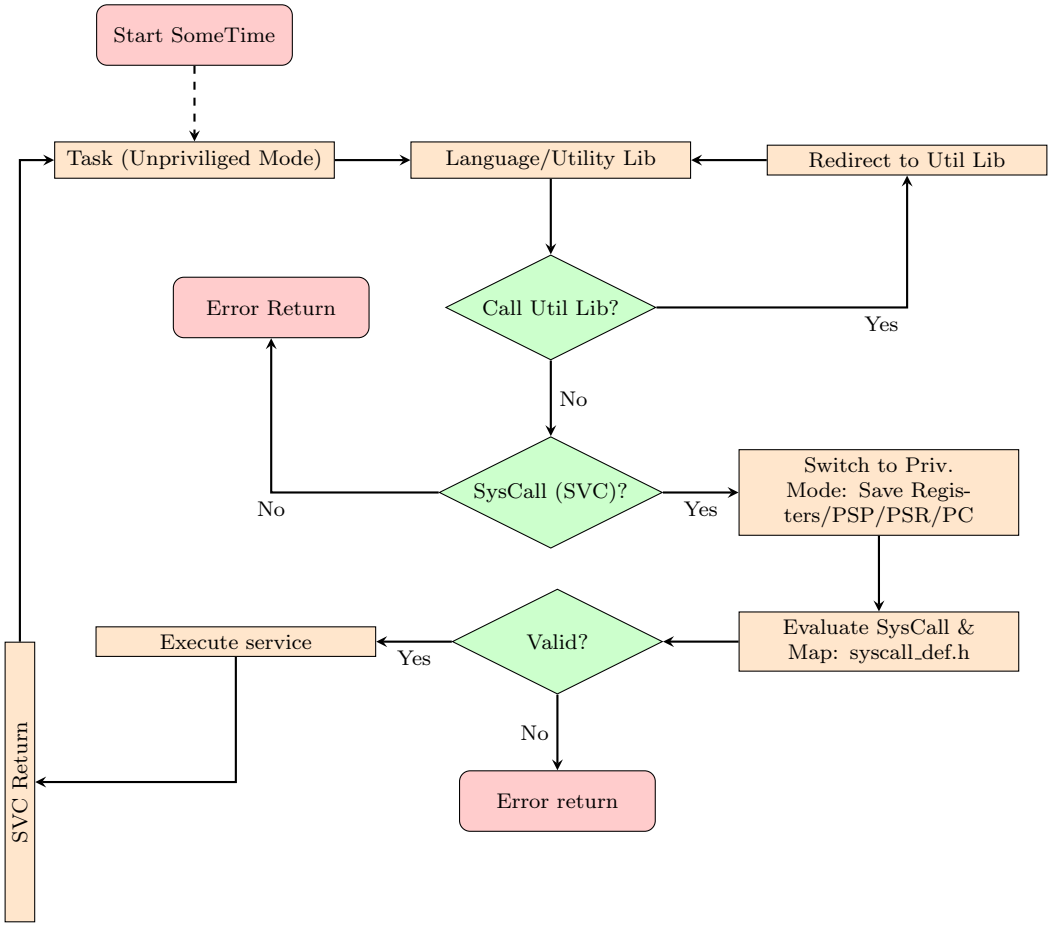


Figure 1: Syscall Steps in short

Exceptions, including the SVC (Supervisor Call), follow a specific hardware mechanism that saves the processor's previous operational context and return address before the exception is accepted. This mechanism ensures that the processor can accurately restore the execution state once the exception service routine is completed. The detailed operation of this mechanism is described in the following section.

2.3 Exception Mechanism

In ARM Cortex-M microcontrollers (such as those used in STM32 devices), the value `EXC_RETURN` is a special constant loaded into the Program Counter (`PC`) when returning from an exception or interrupt service routine (ISR). Unlike normal instructions, this value acts as a *token* that instructs the processor on how to restore the previous context and which stack and mode to return to.

When an exception occurs, the processor automatically saves a predefined set of registers on the stack and places a specific `EXC_RETURN` code into the Link Register (`LR`). Upon executing `BX LR`, the CPU interprets this value to decide how to return to the previous context.

2.3.1 Mechanism of Operation

During exception entry:

- The processor automatically saves the registers `R0--R3`, `R12`, `LR`, `PC`, and `xPSR` onto the current stack.
- The `LR` is set to a special value known as `EXC_RETURN`.

When the exception handler completes and executes the instruction

```
BX LR
```

the processor interprets the contents of `LR` as an `EXC_RETURN` code. Based on this code, it restores the correct registers and returns to either *Thread mode* or *Handler mode*, using either the *Main Stack Pointer (MSP)* or the *Process Stack Pointer (PSP)*.

2.3.2 Common `EXC_RETURN` Values

The table below summarizes the most commonly encountered `EXC_RETURN` codes.

Value	Meaning
`0xFFFFFFF1`	Return to Handler mode using MSP
`0xFFFFFFF9`	Return to Thread mode using MSP
`0xFFFFFFFD`	Return to Thread mode using PSP
`0xFFFFFFE1/F9/FD`	Same as above but with floating-point context (FPU present)

2.3.3 Bit-Level Interpretation

The `EXC_RETURN` value is always of the form

$$\texttt{EXC\_RETURN} = 0xFFFFFF00\,|\,\text{bits}$$

For ARM Cortex-M4 (and similar cores), the low byte contains the following relevant bits:

Bit	Name	Meaning
4	Return mode	1: Thread mode, 0: Handler mode
3	Stack selection	1: PSP, 0: MSP
2	FPU context	1: FPU state not saved, 0: FPU state saved
0	Reserved	Always 1

As an example, consider:

$$\texttt{0xFFFFFFFD} = 11111111\,11111111\,11111111\,11111101_2$$

From this, we can decode:

- Bit[4] = 1 ⇒ Return to Thread mode
- Bit[3] = 1 ⇒ Use PSP
- Bit[2] = 1 ⇒ FPU state not saved

2.3.4 Decoding in C

The following C function demonstrates how to interpret an `EXC_RETURN` value for debugging or analysis.

Listing 1: Decoding EXC_RETURN in C

```
void decode_exc_return(uint32_t exc_return)
{
        printf("EXC_RETURN = 0x%08lX\n", exc_return);
        printf("Return to: %s\n", (exc_return & (1<<4)) ? "Thread mode" : "Handler mode");
        printf("Stack used: %s\n", (exc_return & (1<<2)) ? "PSP" : "MSP");
        printf("FPU state saved: %s\n", (exc_return & (1<<3)) ? "No" : "Yes");
}
```

For example, calling `decode_exc_return(0xFFFFFFFD)` produces:

```
EXC_RETURN = 0xFFFFFFFD
Return to: Thread mode
Stack used: PSP
FPU state saved: No
```

2.3.5 Determining the Active Stack in an ISR

In many RTOS implementations, the stack used by the interrupted context is determined by testing bit 2 of `EXC_RETURN`. The following code is typically placed inside a naked ISR (such as `SVC_Handler`):

Listing 2: Detecting Active Stack Pointer in an Exception

```
__attribute__((naked)) void SVC_Handler(void)
        {
        __asm volatile (
        "TST lr, #4\n"    // Test bit 2 of EXC_RETURN
        "ITE EQ\n"
        "MRSEQ r0, MSP\n"    // If bit2 == 0 -> MSP
        "MRSNE r0, PSP\n"    // If bit2 == 1 -> PSP
        "B SVC_Handler_C\n"
        );
        }

void SVC_Handler_C(uint32_t *stack_frame)
        {
        uint32_t stacked_r0 = stack_frame[0];
        uint32_t stacked_pc = stack_frame[6];
        uint32_t stacked_psr = stack_frame[7];
        // Analyze or modify the caller's context if necessary
        }
```

2.3.6 Manual Context Restoration

The following assembly code demonstrates a simplified context switch routine, as used by a real-time operating system (RTOS). The `PendSV_Handler` saves the context of the current task, loads the context of the next task, and uses `EXC_RETURN` to resume normal execution.

Listing 3: Manual Context Restore Using EXC_RETURN

```
        PendSV_Handler:
        MRS   r0, PSP              ; Get process stack pointer of current task
        STMDB r0!, {r4-r11}        ; Save callee-saved registers
        LDR   r1, =current_tcb
        STR   r0, [r1]             ; Store updated PSP in task control block

        LDR   r1, =next_tcb
```

```
LDR   r0, [r1]
LDMIA r0!, {r4-r11}          ; Restore registers of next task
MSR   PSP, r0                ; Update PSP for next task

LDR   lr, =0xFFFFFFFD        ; EXC_RETURN: Thread mode, use PSP
BX    lr                     ; Exit exception and restore task context
```

Here, the value `0xFFFFFFFD` ensures that upon executing `BX LR`, the processor returns to Thread mode and restores registers from the Process Stack Pointer, effectively resuming the next task.

2.3.7 Practical Significance

The `EXC_RETURN` mechanism is a key feature enabling efficient context switching in Cortex-M processors. It removes the need for software-based context restoration by allowing the processor to automatically restore all registers when returning from an exception. This design significantly simplifies RTOS kernels, context switching, and interrupt nesting.

For instance, `FreeRTOS` and similar systems rely on:

```
LDR lr, =0xFFFFFFFD
BX  lr
```

to return from the `PendSV` exception directly into the next task's context.

The `EXC_RETURN` value is an elegant and essential feature of the ARM Cortex-M architecture. Understanding its bit-level encoding and how it directs the processor during exception return is vital for low-level system programming, RTOS development, and debugging advanced embedded applications.

2.4 What to submit

The assignment envisions creating a set of functions and policies to carry out the service call and task management with Round-Robin (timesharing) policies. The work contains two parts, (a) syscall and (b) task management, given below.

2.4.1 Syscall

Implement high-level test code to evaluate the SVC and test all SYS_*. The implementation must contain (i) high-level functions (such as 'duprintf', convert to string and call write utility function with an appropriate file descriptor and generated string), (ii) application layer utility functions, such as write, (iii) syscall function (suschas SYS_write) in syscall.c and finally, kernel service functions. The exec_return code either returns a success or an error code. The list of functions you need to implement in different level of system access modes are:

Full Spectrum Syscall Implementation			
Unprivileged Mode		Privileged Mode	
User Function	**Utility Library**(Optional)	**Service Name**	**Driver or service**
printf(···)	write(fd,data,size_t)	SYS_write	UART (if fd==STDOUT_FILENO[1])
scanf(···)	read(fd,data,size_t)	SYS_read	STDIN˙FILENO
reboot()	sys_reboot())	SYS_reboot	NMI (reset)
exit()	sys_exit())	SYS_exit	Terminate (TCB status)
getpid()	sys_getpid())	SYS_getpid	TCB/PCB – task_id
gettime()	sys_gettime())	SYS_time	Systick Time
yield()	–	SYS_yield	PendSV for reschedule

2.4.2 Task Management

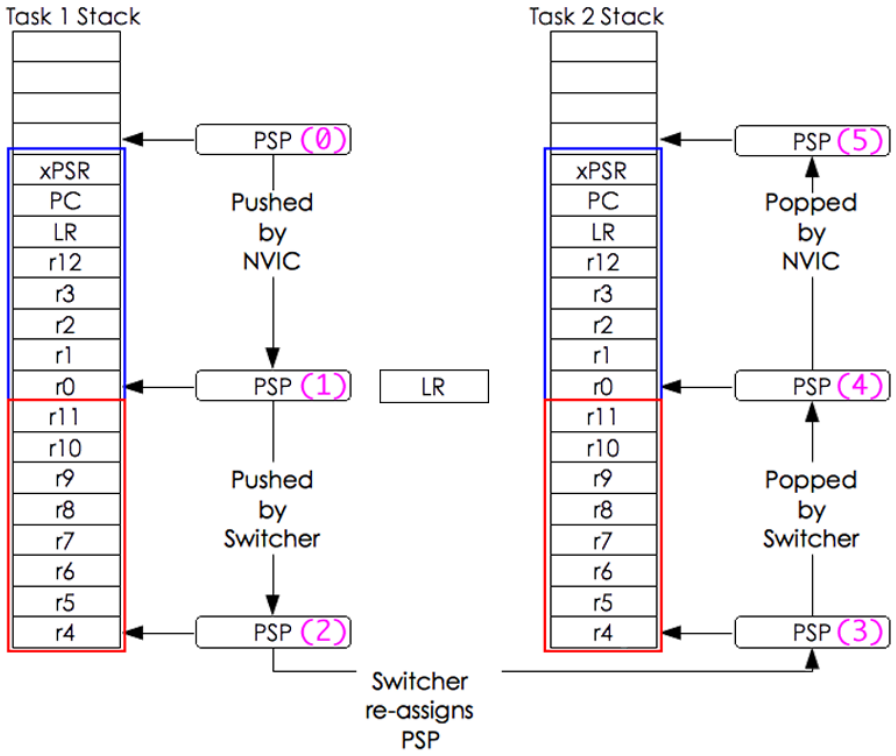
2.5 Prototyping, data structure and C code

Appendix C shows the directory tree structure of the DUOS.

2.5.1 Data Structure for TCB

You must impelement the data struture for the task as,

```
typedef struct t_task_tcb {
        uint32_t magic_number;        // Magic number for stack/structure integrity (e.g., 0xFECABAA0)
        uint16_t task_id;             // Unique 16-bit task ID starting from 1000
        void    *psp;                 // Task stack pointer (points to the task's stack frame)
        uint16_t status;              // Task status: running, waiting, ready, killed, or terminated
        uint8_t  priority;            // Task priority (lower value = higher priority)
        uint16_t parent_id;           // Task ID of parent process (if any)
        uint32_t execution_time;      // Total execution time in milliseconds
        uint32_t waiting_time;        // Total waiting time in milliseconds
        uint32_t digital_signature;   // Digital signature or integrity check (e.g., 0x00000001)
        uint32_t w_chld[16];          // Array to store suspended child task IDs or signals from parent
        uint32_t *heap_mem_start;     // Pointer to dynamically allocated heap memory for the task
        uint32_t heap_mem_size;       // Size of allocated heap memory in bytes
        uint32_t *open_resources[8];  // Pointers to resources (files, devices) opened by the task
        uint8_t  sem_waiting_count;   // Number of semaphores the task is currently waiting on
        uint8_t  mutex_locked_count;  // Number of mutexes the task currently holds
        uint32_t last_wakeup_time;    // Timestamp of last wakeup (for sleep or delay)
} TCB_TypeDef;
```



To be continued ..... Next Assignment
(Scheduling, sychronization, deadlock and so on)

Appendices

A kunistd.h

```
#ifndef _KERN_UNISTD_H_
#define _KERN_UNISTD_H_

/* Constants for read/write/etc: special file handles */
#define STDIN_FILENO  0      /* Standard input */
#define STDOUT_FILENO 1      /* Standard output */
#define STDERR_FILENO 2      /* Standard error */

#endif /* _KERN_UNISTD_H_ */
```

B syscall_def.h

```
#ifndef _SYSCALL_DEF_H_
#define _SYSCALL_DEF_H_

#define SYS_fork         0
#define SYS_vfork        1
#define SYS_execv        2
#define SYS__exit        3
#define SYS_waitpid      4
#define SYS_getpid       5
#define SYS_getppid      6
//                             (virtual memory)
#define SYS_sbrk         7
#define SYS_mmap         8
#define SYS_munmap       9
#define SYS_mprotect     10
//                             (security/credentials)
#define SYS_umask        17
#define SYS_issetugid    18
#define SYS_getresuid    19
#define SYS_setresuid    20
#define SYS_getresgid    21
#define SYS_setresgid    22
#define SYS_getgroups    23
#define SYS_setgroups    24
#define SYS___getlogin   25
#define SYS___setlogin   26
//                             (signals)
#define SYS_kill         27
#define SYS_sigaction    28
#define SYS_sigpending   29
```

```
#define SYS_sigprocmask  30
#define SYS_sigsuspend   31
#define SYS_sigreturn    32
//                             -- File-handle-related --
#define SYS_open         45
#define SYS_pipe         46
#define SYS_dup          47
#define SYS_dup2         48
#define SYS_close        49
#define SYS_read         50
#define SYS_pread        51

#define SYS_getdirentry  54
#define SYS_write        55
#define SYS_pwrite       56

#define SYS_lseek        59
#define SYS_flock        60
#define SYS_ftruncate    61
#define SYS_fsync        62
#define SYS_fcntl        63
#define SYS_ioctl        64
#define SYS_select       65
#define SYS_poll         66

//                             -- Pathname-related --
#define SYS_link         67
#define SYS_remove       68
#define SYS_mkdir        69
#define SYS_rmdir        70
#define SYS_mkfifo       71
#define SYS_rename       72
#define SYS_access       73
//                             (current directory)
#define SYS_chdir        74
#define SYS_fchdir       75
#define SYS___getcwd     76
//                             (symbolic links)
#define SYS_symlink      77
#define SYS_readlink     78
//                             (mount)
#define SYS_mount        79
#define SYS_unmount      80

//                             -- Any-file-related --
#define SYS_stat         81
#define SYS_fstat        82
#define SYS_lstat        83
```

```
//                              (timestamps)
#define SYS_utimes       84
#define SYS_futimes      85
#define SYS_lutimes      86
//                              (security/permissions)
#define SYS_chmod        87
#define SYS_chown        88
#define SYS_fchmod       89
#define SYS_fchown       90
#define SYS_lchmod       91
#define SYS_lchown       92
//                              -- Sockets and networking --
#define SYS_socket       98
#define SYS_bind         99
#define SYS_connect      100
#define SYS_listen       101
#define SYS_accept       102
//#define SYS_socketpair 103
#define SYS_shutdown     104
#define SYS_getsockname  105
#define SYS_getpeername  106
#define SYS_getsockopt   107
#define SYS_setsockopt   108

//                              -- Time-related --
#define SYS___time       113
#define SYS___settime    114
#define SYS_nanosleep    115

//                              -- Other --
#define SYS_sync         118
#define SYS_reboot       119
#define SYS_yield        120
/*CALLEND*/

#endif
```

C DUOS Directory Structure – Tree

Use 'tree' command in linux for detail DUOS tree. (Following a tiny detail)

```
duos ..... Root of DUOS
└── src ..... Source Tree
    ├── compile ..... Contains compile and binary download logic
    │   ├── Makefile ..... Make rules
    │   ├── mapfile ..... contain map file(s)
    │   ├── object ..... Object code
    │   └── target ..... Binary/Executable
    ├── doc ..... Documentation
    │   └── Readme.txt
    └── kern ..... Kernel Components
        ├── arch
        ├── dev
        ├── include
        ├── kmain
        ├── lib
        ├── proc
        ├── syscall
        ├── thread
        └── vfs
```

Alert: You can discuss with your classmates, however, do not copy code from others.