System call f = read(fd, buffer, nbytes);

User Space (Steps 1–6)

- **Push nbytes:**

  The user program pushes the number of bytes (nbytes) to be read onto the stack. This specifies how much data the program wants to read from the file.
- **Push &buffer:**

  The user program pushes the memory address of the buffer (&buffer) where the read data will be stored. This buffer is used to transfer the data from kernel space back to user space.
- **Push fd:**

  The program pushes the file descriptor (fd) onto the stack. This identifies the file or resource to be read.
- **Call read:**

  The program calls the read function from the library. This is a wrapper provided in the user space to simplify system calls for the programmer.
- **Put system call code in register:**

  The library procedure writes the system call number for read into a specific CPU register (e.g., eax in x86). This tells the kernel which service (read in this case) the program is requesting.
- **Trap to kernel:**

  The library procedure triggers a software interrupt (e.g., using int 0x80 in x86 or svc in ARM) to transition from user mode to kernel mode. This is called a trap, and it allows the CPU to switch to executing privileged kernel code.

Kernel Space (Steps 7–9)

- **Dispatch to appropriate handler:**

  The trap handler in the kernel examines the system call code in the register and dispatches the request to the appropriate system call handler (in this case, the handler for the read system call).
- **Execute system call handler:**

  The kernel system call handler processes the request. For read, this may involve checking the validity of the file descriptor, verifying permissions, accessing the file system, and reading the requested data.
- **Return data to user space:**

  After completing the operation, the system call handler places the result (e.g., the number of bytes read) in a designated register (e.g., eax) and prepares to return control to the user program.

User Space Return (Steps 10–11)

- **Return from kernel:**

Control is returned from kernel mode to user mode. The CPU resumes executing instructions in the user program after the system call.

- **Increment stack pointer (SP):**

  The library function cleans up the stack by incrementing the stack pointer to remove the parameters (nbytes, &buffer, and fd) that were previously pushed. Afterward, it returns control to the user program.

Stack Pointer (SP) and Stack Frame Overview

Stack Pointer (SP) is used to manage the stack for the current task or interrupt context.

MSP (Main Stack Pointer) is used in Handler Mode (interrupts/exceptions) and also in Thread Mode when SPSEL = 0.
PSP (Process Stack Pointer) is used in Thread Mode when SPSEL = 1, often to separate task stack management.

Stack Frame is a part of the stack that contains data necessary for function calls, including:

The return address (where to return after a function or interrupt handler finishes).
The saved register values (such as R0-R12, LR, PC, xPSR) so the task or interrupt handler can restore the state of the processor when returning.

Operating System Task Management with SP

1. **Task Initialization (Thread Mode)**
   - When an OS is setting up a new task, it begins by handling the Supervisor Call (SVC) exception, which is commonly used to create tasks and manage system-level functions like memory allocation and task creation.
   - SVC Handler is triggered to set up the task.
     - The SVC handler creates a new task stack frame in the Process Stack Pointer (PSP), which will hold the context (registers, return address) for the new task.
     - The task's stack frame is set up with default values, which include:
       - Saved registers (R0-R12, LR, PC, xPSR).
       - The exception return address for where the task should continue when it's scheduled to run again.

2. **Context Switching and Task Scheduling**

- SysTick Handler: In an RTOS, the SysTick Timer is typically used to trigger periodic context switches (e.g., for preemption).
  - The SysTick handler saves the current task’s context and loads the task’s stack from the PSP.
  - This involves saving the current status (registers, etc.) of the running task, and saving the current PSP value.

3 Context Switch Steps

- Save the current context: Before switching tasks, the OS saves the current task's state (register values and stack pointer) into its stack frame.
    - The current task's stack frame (on the PSP) contains the state of registers (R0-R12, LR, etc.), as well as the current PSP value.
    - Save current PSP value: The value of the PSP register (the stack pointer for the current task) is saved into the task's stack frame.
    - Set the PSP for the next task: The OS sets the PSP to point to the task's stack frame of the next task to run.
    - Restore the context of the next task: The OS restores the registers and the PSP value for the next task.
        - This includes setting up the Program Counter (PC) and Link Register (LR) so the next task will resume execution from where it left off.

4 **Task Switch and Return**

- Once the context for the next task has been restored, the exception return mechanism is used to switch back to the next task.
- Exception Return involves using the BX LR (branch to link register) instruction, which ensures that the Program Counter (PC) points to the appropriate location in the task's code (using the restored stack frame).

NVIC vs. SVC

- NVIC (Nested Vectored Interrupt Controller) and SVC (Supervisor Call) are both important components in ARM Cortex-M microcontrollers, but they serve different purposes:NVIC is responsible for managing interrupts and exceptions. It handles the prioritization and nesting of interrupts, which ensures that higher-priority interrupts can preempt lower-priority ones.

- SVC, on the other hand, is a mechanism that allows the execution of system-level functions. It provides a way for user-level applications to request services from the OS by triggering a software interrupt to perform operations like task switching or privileged operations.

Feature
NVIC
SVC
Purpose
Manages interrupts and exceptions in ARM Cortex-M.

Allows user code to invoke system services in privileged mode.

Functionality
Handles interrupt prioritization, nesting, and exception management.

Invokes a software interrupt to perform tasks such as context switching, task creation, etc.

Control Mode
Handles interrupts in both privileged and unprivileged modes.

Works in unprivileged mode to switch to privileged mode.

Trigger Method
Automatically triggered by hardware (e.g., external interrupts, timers).

Triggered explicitly by the program using the \texttt{SVC} instruction.
Interrupt/Exception
Supports hardware-driven interrupts and exceptions.
Uses software-based interrupts (via the \texttt{SVC} instruction).
Context Switching

Can trigger a context switch by handling interrupts and exceptions.
Triggers a context switch to privileged mode when invoked.

SVC (Supervisor Call)

- The SVC (Supervisor Call) is a software interrupt mechanism in ARM Cortex-M processors. It allows an application running in an unprivileged mode to request services from the operating system or the kernel, which operates in privileged mode. This transition from unprivileged to privileged mode allows the operating system to perform system-level tasks such as process management, I/O operations, or hardware access.

- **Usage:** SVC is typically used for making system calls in an OS, such as task switching, memory allocation, and other OS services.

- **Trigger:** The SVC is explicitly triggered by the SVC instruction in the program. This triggers an exception that can then be handled by the OS.

- **Exception Handler:** When an SVC is triggered, the processor enters an exception handler, and the OS can perform the necessary privileged operations. Afterward, control is returned to the user application.

- **Independence from OS Implementation:** The application does not need to know the exact address of the OS service function or the details of the hardware-level execution. It only needs to know the service number (the function to call) and the parameters required for that service.

- **SVC Instruction:** The application triggers an SVC call using the SVC instruction, along with an immediate value that indicates which OS service is being requested, and possibly the parameters. The format is:

- SVC #0x3 (Call service function 3)
- SVC 0x3 (Traditional syntax accepted in ARM)

PendSVC (Pending Supervisor Call)

- The PendSVC is a state where a Supervisor Call (SVC) request has been issued but has not yet been executed. This state occurs when the processor is in the middle of handling an interrupt or exception, and another SVC request is made during this time. The request is "pending" and will be handled as soon as the processor finishes the current exception handling and returns to the main program flow.

- **Usage:** PendSVC is useful in scenarios where multiple SVC requests might be generated in a short amount of time. For instance, when the OS needs to handle multiple tasks or context switches and does not want to immediately interrupt the current exception handling process.

- **Trigger:** The PendSVC state is triggered when an SVC request is made during an ongoing exception, causing the SVC handler to be "queued" until the processor finishes handling the current exception.

- **Execution:** Once the processor finishes the current exception or interrupt handling, the pending SVC is executed.

- **Context Switching:** PendSV is typically triggered by the OS when a context switch is required (e.g., when a higher-priority task needs to run). It is a "pending" call that gets executed when the processor is ready to handle it, usually after finishing the current interrupt or exception.

- **Efficient Task Management:** Since PendSV is often used for tasks like task switching, it ensures that the system can efficiently manage multiple tasks without unnecessary interruption. It works by signaling that a context switch should be done, and the exception will occur when the processor is ready.

SVC (Supervisor Call) System Call in Embedded Systems
In ARM Cortex-M microcontrollers, the SVC (Supervisor Call) instruction allows user-level code to invoke system services provided by the kernel (OS) in privileged mode. The **SVC handler** is responsible for handling these requests. Here's a detailed breakdown of how this works, including Keil C, GCC, and the steps to manage stack pointers (PSP or MSP) in the SVC handler.

In keil c:
```
svc 0x03;  // Generate an SVC instruction, with service number 3
```

In gcc:

```
__asm("svc #0x3");  // Generate an SVC call with service number 3
```

SVC Handler Execution
When the SVC instruction is executed by the program, the processor switches to Handler Mode (exception mode) and begins executing the SVC handler. The handler will process the request and provide the appropriate service (e.g., task switching, memory management).

Immediate Data (SVC Number):

- The SVC handler can determine which service is being requested by reading the immediate value encoded in the SVC instruction. This value is the SVC number (the service number), which tells the handler which function to execute.

- The SVC number is embedded in the instruction, and you can access it by examining the stacked Program Counter (PC) in the exception frame.

Stack Management: MSP vs. PSP
The processor uses two stack pointers:

- **MSP (Main Stack Pointer):** Used in Handler Mode (exceptions and interrupts) and sometimes in Thread Mode when SPSEL = 0.
- **PSP (Process Stack Pointer):** Used in Thread Mode when SPSEL = 1 for task management, especially in an RTOS.

Determining which Stack (MSP or PSP) is in Use:

- When the SVC handler is called, you need to figure out whether the stack pointer is pointing to PSP or MSP. This is important because the stacked context (registers and PC) is different depending on which stack was used during the interrupt or exception.

Link Register (LR) is used to determine this:

- If the EXC_RETURN bit is set to 0, then MSP is being used.
- If the EXC_RETURN bit is set to 1, then PSP is being used.

The EXC_RETURN is stored in the LR (Link Register) during context switches.

Code
SVC_Handler:

- **TST LR, #4** ; Test bit 2 of EXC_RETURN (from LR)
- **ITE EQ** ; If-Then-Else (ITE) instruction for conditional execution
- **MRSEQ R0, MSP** ; If the test was equal (bit 2 = 0), MSP is used, so copy it to R0
- **MRSNE R0, PSP** **;** If the test was not equal (bit 2 = 1), PSP is used, so copy it to R0
- **LDR R0, [R0, #24]** **;** Load the stacked PC value from the exception frame (stacked PC = **the** address of the instruction after the SVC)
- **LDRB R0, [R0, #-2];** Load the first byte of the SVC instruction (the immediate value) **from** the instruction address (now in R0)
  ; Now, the SVC number is in R0

```
void SVC_Handler_C(unsigned int * svc_args)
 {
uint8_t svc_number;
uint32_t stacked_r0, stacked_r1, stacked_r2, stacked_r3;
svc_number = ((char *) svc_args[6])[-2]; //Memory[(Stacked PC)-2]
stacked_r0 = svc_args[0];
stacked_r1 = svc_args[1];
stacked_r2 = svc_args[2];
stacked_r3 = svc_args[3];
// ... other processing
...
// Return result (e.g. sum of first two arguments)
svc_args[0] = stacked_r0 + stacked_r1;
return;
}
```

```
#include <stdio.h>
// Define SVC functions
int __svc(0x00) svc_service_add(int x, int y); // Service #0 : Add
int __svc(0x01) svc_service_sub(int x, int y); // Service #1 : Sub
int __svc(0x02) svc_service_incr(int x); // Service #2 : Incr
void SVC_Handler_main(unsigned int * svc_args);
// Function declarations
int main(void)
{
int x, y, z;
x = 3; y = 5;
z = svc_service_add(x, y);
printf ("3+5 = %d \n", z);
x = 9; y = 2;
z = svc_service_sub(x, y);
printf ("9-2 = %d \n", z);
x = 3;
z = svc_service_incr(x);
printf ("3++ = %d \n", z);
while(1);
}
```

```
// SVC handler - Assembly wrapper to extract
//stack frame starting address
__asm void SVC_Handler(void)
{
TST LR, #4 ; Test bit 2 of EXC_RETURN
ITE EQ
MRSEQ R0, MSP; if 0, stacking used MSP, copy to R0
MRSNE R0, PSP; if 1, stacking used PSP, copy to R0
B __cpp(SVC_Handler_C)
ALIGN 4
}
// SVC handler - main code to handle processing
// Input parameter is stack frame starting address
// obtained from assembly wrapper.
void SVC_Handler_main(unsigned int * svc_args)
{
// Stack frame contains:
// r0, r1, r2, r3, r12, r14, the return address and xPSR
// - Stacked R0 = svc_args[0];
// - Stacked R1 = svc_args[1];
// - Stacked R2 = svc_args[2];
// - Stacked R3 = svc_args[3];
```

```
// - Stacked R12 = svc_args[4]
// - Stacked LR = svc_args[5]
// - Stacked PC = svc_args[6]
// - Stacked xPSR = svc_args[7]
unsigned int svc_number;
svc_number = ((char *)svc_args[6])[-2];
switch(svc_number)
{
case 0: svc_args[0] = svc_args[0] + svc_args[1];
break;
case 1: svc_args[0] = svc_args[0] - svc_args[1];
break;
case 2: svc_args[0] = svc_args[0] + 1;
break;
default: // Unknown SVC request
break;
}
return;
}
```

Bits	Function
Bits 31:3	Reserved
Bit 2	**FPCA:** Indicates whether floating-point context currently active: 0: No floating-point context active 1: Floating-point context active. The Cortex-M4 uses this bit to determine whether to preserve floating-point state when processing an exception.
Bit 1	**SPSEL:** Active stack pointer selection. Selects the current stack: 0: MSP is the current stack pointer 1: PSP is the current stack pointer. In Handler mode this bit reads as zero and ignores writes. The Cortex-M4 updates this bit automatically on exception return.
Bit 0	**nPRIV:** Thread mode privilege level. Defines the Thread mode privilege level. 0: Privileged 1: Unprivileged.

PendSV context switching

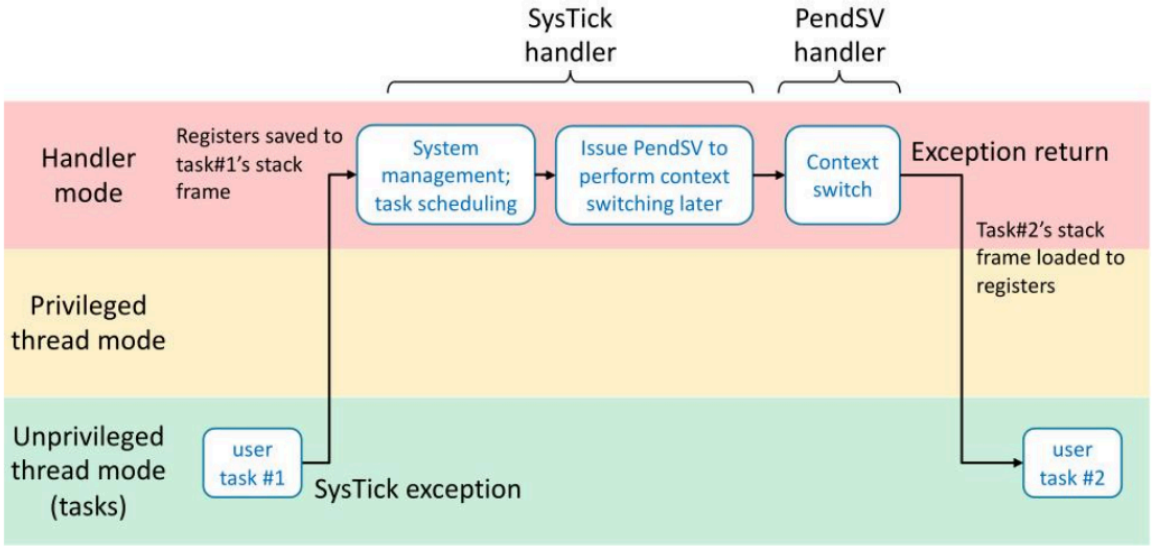


Figure 2: PendSV in action for context switching steps

```
#include "stm32f4xx.h"
```

```c
// Define task functions
void task0(void);  // Blinks green LED every 500ms
void task1(void);  // Blinks yellow LED every 1000ms
void task2(void);  // Blinks red LED every 1500ms

// Task stack arrays for each task
uint32_t task0_stack[128];
uint32_t task1_stack[128];
uint32_t task2_stack[128];

// Task pointers
void (*current_task)(void); // Pointer to current task

// SysTick Timer for context switching (simple time slice)
void SysTick_Handler(void)
{
    // Trigger PendSV to perform context switch
    SCB->ICSR |= SCB_ICSR_PENDSVSET_Msk;
}

// PendSV Handler for context switching
void PendSV_Handler(void)
{
    // Save context of the current task
    __asm("MRS R0, PSP");               // Get current PSP (task stack pointer)
    __asm("STMDB R0!, {R4-R11}");       // Save registers R4 to R11 to the stack

    // Select next task based on the current task
    if (current_task == task0) {
        current_task = task1;
    } else if (current_task == task1) {
        current_task = task2;
    } else {
        current_task = task0;
    }

    // Restore context of the next task
    __asm("LDMIA R0!, {R4-R11}");       // Restore registers R4 to R11 from the
stack
    __asm("MSR PSP, R0");                // Update PSP with the restored value
    __asm("BX LR");                      // Return to the next task
}

// Task0: Blinks the green LED every 500ms
void task0(void)
{
```

```
    while(1) {
        // Turn on Green LED
        GPIOA->ODR |= (1 << 5);   // Assuming green LED is on pin 5 of GPIOA
        delay_ms(500);
        GPIOA->ODR &= ~(1 << 5);  // Turn off Green LED
        delay_ms(500);
    }
}

// Task1: Blinks the yellow LED every 1000ms
void task1(void)
{
    while(1) {
        // Turn on Yellow LED
        GPIOA->ODR |= (1 << 6);   // Assuming yellow LED is on pin 6 of GPIOA
        delay_ms(1000);
        GPIOA->ODR &= ~(1 << 6);  // Turn off Yellow LED
        delay_ms(1000);
    }
}

// Task2: Blinks the red LED every 1500ms
void task2(void)
{
    while(1) {
        // Turn on Red LED
        GPIOA->ODR |= (1 << 7);   // Assuming red LED is on pin 7 of GPIOA
        delay_ms(1500);
        GPIOA->ODR &= ~(1 << 7);  // Turn off Red LED
        delay_ms(1500);
    }
}

// Delay function
void delay_ms(uint32_t ms)
{
    uint32_t count = ms * 1000;
    while (count--) {
        __NOP();  // No operation - busy wait
    }
}

// Main function to initialize the OS and start tasks
int main(void)
{
    // Initialize LEDs (assume GPIOA configured for LEDs)
    GPIOA->MODER |= (1 << 10) | (1 << 12);  // Set PA5, PA6, PA7 as output
```

```
    // Initialize task pointers
    current_task = task0;

    // Initialize SysTick for periodic interrupts
    SysTick_Config(SystemCoreClock / 1000);  // 1ms tick

    // Initialize PendSV for context switching
    SCB->SHP[10] = 0xFF;  // Set PendSV priority to the lowest

    // Start the multitasking system (context switch will occur in SysTick
interrupt)
    while (1)
    {
        // Run the current task
        current_task();
    }
}
```

interrupt Service Routines (ISRs) are special functions designed to handle hardware interrupts. They execute quickly and respond to immediate needs, like handling an input signal from a sensor or processing communication events. However, there are strict guidelines about what can and cannot be done inside an ISR to ensure system stability, reliability, and performance.

The key point is that ISRs must execute as quickly and efficiently as possible to prevent interrupting other critical operations or causing system instability. Let's go through the specifics of what should not be done in an ISR:

- **No Blocking Code in ISRs**

Blocking code refers to code that causes the program to wait for something to happen before continuing, effectively preventing further execution. This can delay the ISR, block other interrupts from being handled, or cause a system to become unresponsive.

- **Wait functions (wait()):** Functions that cause the system to pause for a certain period (such as waiting for a signal or waiting for some time to pass). These functions are blocking in nature and prevent the ISR from completing quickly.

- **Infinite Loops:** Writing an infinite loop (e.g., while(1) {}) inside an ISR will make the ISR never return, which is dangerous because it will stop further processing of interrupts and cause the system to "hang". ISRs should always exit quickly after performing their task.

- **Other blocking calls:** Any function that internally waits for some condition or input before proceeding (such as waiting for hardware events, user input, or system resources) is not suitable for an ISR because it will block the ISR from finishing promptly.

- **No printf(), malloc(), or new in ISRs**
- **printf() in ISR:** Printing to the console or any form of output in the ISR is a bad practice because it involves significant overhead. The printf() function is not designed to be fast and may involve buffer management, string formatting, and even accessing external peripherals (like UART or

serial ports) — all of which can take an unpredictable amount of time and can introduce delays. Additionally, if printf() is used in the ISR, it may not work as expected or lead to undefined behavior because it might depend on other system resources (like heap memory) that could be in an inconsistent state during the ISR.

- **Memory Allocation (malloc() or new):** Memory allocation functions like malloc() or new are not safe to use in ISRs because they involve dynamic memory management that can involve complex operations (e.g., searching for free memory, updating the heap, etc.). These functions can block the program, cause memory fragmentation, or lead to undefined behavior when used in an ISR context. They may also interfere with other critical sections of code that rely on the heap, especially in a real-time system where memory management needs to be predictable.

- **Why malloc()/new are unsafe:** These functions are generally not re-entrant and may require locks or other mechanisms to ensure that multiple threads or interrupts don't interfere with each other. Since an ISR can preempt other code at any point, calling these functions can lead to race conditions, memory corruption, or system crashes.

- **ISRs should be as fast as possible to prevent blocking other interrupts.**
- **Blocking functions like wait(), printf(), and malloc() should never be used in ISRs.**
- **Avoid non-re-entrant functions that rely on shared resources or global state, as they can lead to data corruption or system crashes.**
- **The primary purpose of an ISR is to handle urgent, time-sensitive tasks. Complex or resource-heavy tasks should be deferred to the main loop or background tasks.**

What is a Process?
A process is an independent program in execution. It has its own memory space, system resources, and execution context, including a program counter, registers, and stack. A process is the basic unit of execution in most operating systems, and it represents a running application or service.

Key Features of a Process:
- **Memory Isolation:** Each process has its own memory space. It cannot directly access the memory of other processes, providing isolation and security between applications.
- **Independent Execution:** A process runs independently from others, with its own resources. It cannot directly communicate with other processes unless mechanisms like inter-process communication (IPC) are used.
- **Multiple Processes:** A system can run multiple processes concurrently. These processes can be in different states: running, waiting, ready, etc.
- **Resource Management:** Each process has a set of resources allocated to it, including CPU time, memory, and I/O devices.
- **Heavyweight:** Processes are considered heavyweight because creating and managing them involves more overhead, such as memory allocation, context switching, and inter-process communication.

Example:
When you run a web browser, text editor, or game on your computer, each of these applications runs as a separate process, each with its own allocated memory and execution context.

What is a Thread?

A thread is a smaller unit of execution within a process. A thread, sometimes referred to as a "lightweight" process, is the basic unit of CPU scheduling within a process. Multiple threads can exist within a single process and share the same memory space, allowing them to communicate with each other more efficiently than if they were separate processes.

Key Features of a Thread:

- **Shared Memory:** Threads within the same process share the same memory space, including global variables and heap memory. This allows for easier and faster communication between threads.
- **Lightweight:** Threads are lightweight because they don't require a new memory space to be created. They simply share the existing memory and resources of their parent process.
- **Concurrent Execution**: Threads can run concurrently, performing different tasks within the same process. For example, a web browser can run one thread for handling user input and another for fetching web pages.
- **Faster Context Switching:** Since threads share memory, switching between threads (context switching) is much faster compared to switching between processes.
- **Resource Sharing:** Threads within a process share system resources like file handles, memory, and data structures, which makes them more efficient for certain types of concurrent processing.
- **Synchronization:** Since threads share memory, synchronization mechanisms (e.g., mutexes, semaphores) are often required to prevent data corruption when multiple threads try to access the same resources simultaneously.

Example:
In a modern web browser, one thread might handle user interface updates, another might download files, and a third might render web pages. All of these threads share the same memory space but perform different tasks concurrently.

When to Use Processes vs. Threads
Use Processes when:

- You need strong isolation between tasks (e.g., running separate applications).
- You need to prevent interference between different tasks for security or stability reasons.
- You want the tasks to be completely independent (e.g., different applications running on the system).
- You require more control over resource management and memory allocation.

Use Threads when:

- You need to perform multiple tasks within the same application that share resources (e.g., multiple background tasks in a game).
- You want to improve the performance of a program through parallel execution (e.g., multi-core CPU usage).
- You need efficient communication between tasks running in the same process (e.g., threads processing different parts of a file concurrently).
- You need low overhead in terms of resource management and creation.

Process Creation:

- **System Call:** A process is created when the OS receives a system call (e.g., fork() or CreateProcess).
- **Process Control Block (PCB):** The OS creates a PCB to store information about the process (PID, state, memory info, etc.).
- **Memory Allocation:** The OS allocates memory for the process, including code, data, heap, and stack.
- **Process Scheduling:** The new process is placed in the ready queue and awaits CPU time to execute.

Process Termination:

- **Exit System Call:** A process ends when it calls an exit system call (e.g., exit() in Unix, ExitProcess() in Windows).
- **Release Resources:** The OS deallocates memory and resources (e.g., file handles, CPU time).
- **Update PCB:** The process's PCB is updated to reflect its terminated state.
- **Process Reaping:** In some systems (e.g., Unix), the parent process must clean up any remaining resources, a process known as "reaping."

Process Creation

When a user or the operating system requests the execution of a program, a process must be created. The OS handles the creation of processes using a system call (such as fork() in Unix-like operating systems) and allocates resources for the process to execute. Here's how the process creation works in more detail:

System Call for Process Creation:

A process is created when the OS receives a system call from an application. In Unix-like systems, this call is typically fork() or exec(). On Windows, functions like CreateProcess are used to initiate a new process.

Allocating Process Control Block (PCB):

When the OS creates a new process, it allocates a Process Control Block (PCB). The PCB is a data structure that contains important information about the process, such as:

Process ID (PID)
Program counter (PC)
CPU registers (e.g., general-purpose registers, stack pointer)
Memory management information (e.g., pointers to process memory, page tables)
Process state (e.g., running, ready, blocked)
I/O status information (e.g., file descriptors)

Assigning Memory Space:

The OS allocates memory space for the new process. This typically includes:

Code Segment: The program's executable code.
Data Segment: Variables and constants.
Heap: Dynamic memory allocation (e.g., malloc() in C).
Stack: The function call stack, which holds function parameters, local variables, and return addresses.

Creating Address Space:

The OS creates an address space for the new process, which is the range of memory addresses it can access. The address space is typically isolated from other processes for security and stability.

Loading the Program:

If the process is being started from an executable program (e.g., a binary file), the OS loads the program code into the allocated memory. This might involve reading the program from disk, placing it into the

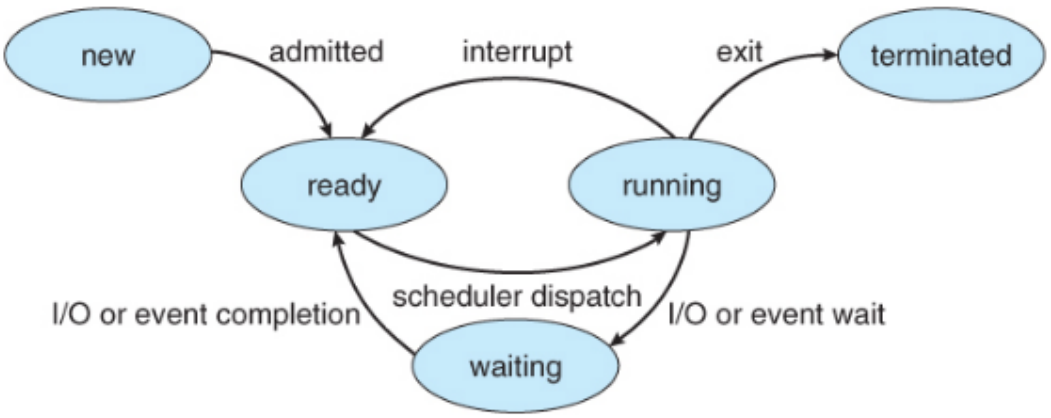
process's address space, and resolving any dependencies (e.g., shared libraries or dynamic link libraries, DLLs).

Setting the Process State to Ready:

Once the new process is created and its memory space is initialized, the OS places the process in the ready state. The process is now ready to be scheduled and executed by the CPU.

Process Scheduling:

The OS scheduler selects the process for execution based on its scheduling algorithm. The process will eventually be moved from the ready queue to the running state, where the CPU executes the process.

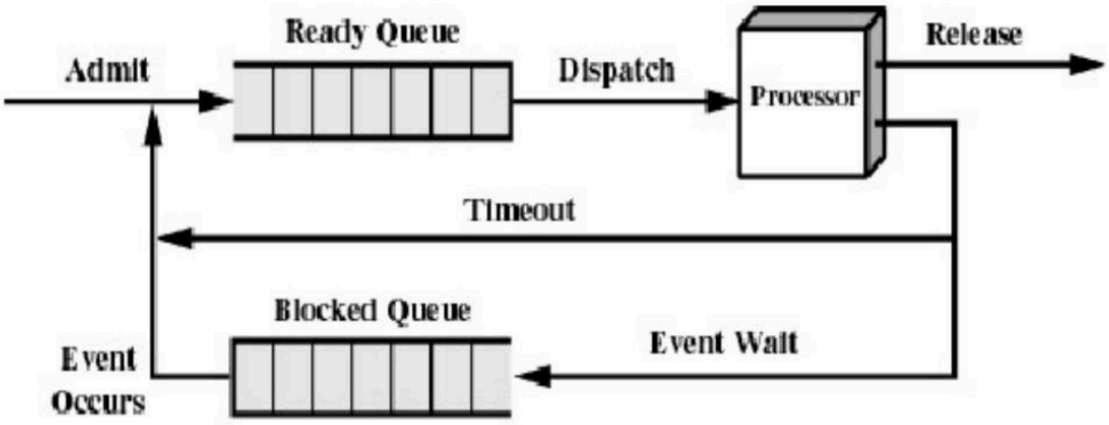


The dispatcher is a part of the operating system responsible for giving control of the CPU to a process that is ready to execute. It performs context switching by saving the state of the currently running process and loading the state of the next process to run.

Key Functions of a Dispatcher:

- **Context Switching:** Saves the state (e.g., registers, program counter) of the currently running process and loads the state of the next process.
- **Process Scheduling:** It determines which process in the ready queue gets the CPU next.
- **Dispatch Time:** It takes very little time to perform the context switch. Ideally, it should be fast to avoid unnecessary delays.

Using Two Queues



(a) Single blocked queue

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Figure 1: Example fileds of a process table – PCB

TCB – Thread Components/attributes

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure 3: OS Thread fileds (Col1) Item shared by all thread, (Col2) Item private to each thread

a process is composed of the following essential elements:

- **Process Control Block (PCB):** Contains metadata for process management.

- **Program Code:** The executable instructions of the program.
- **Process Stack:** Stores local variables, function parameters, and return addresses.
- **Heap:** Used for dynamic memory allocation.
- **Data Segment:** Contains global and static variables.
- **Open File Table:** Tracks open files.
- **Execution Context:** Stores CPU state information.

Threads within a process have both per-thread elements (specific to each thread) and shared elements (common among all threads in the same process). Here's a breakdown:

1. Per-Thread Elements (Thread-Specific)
These are unique to each thread and necessary for its execution:

- **Thread Control Block (TCB):** Stores information like thread ID, state, scheduling priority, and program counter.
- **Program Counter (PC):** Points to the next instruction to execute for that thread.
- **CPU Registers:** Includes general-purpose registers, stack pointer, and other registers needed for thread execution.
- **Thread Stack:** Contains function call information, local variables, and return addresses specific to the thread.

2. Shared Elements (Process-Specific)
These elements are shared among all threads in the same process:

- **Memory Space:** Threads share the same address space, including the code, heap, and data segments of the process.
- **Heap:** Dynamic memory allocated during program execution is shared among all threads of the process.
- **Global Variables:** All threads in a process can access and modify global variables.
- **File Descriptors:** Threads share file handles and other I/O resources within the process.

Threads are used for concurrent execution within a process to improve efficiency and performance. Key reasons include:

- **Parallelism**: Threads allow multiple tasks to run concurrently on multiple CPU cores, speeding up the execution of tasks.

- **Responsiveness:** Threads enable more responsive applications, especially in scenarios like UI updates or background processing (e.g., downloading, file I/O).

- **Resource Sharing:** Threads within the same process share resources like memory, file descriptors, and global variables, reducing overhead compared to separate processes.

- **Better CPU Utilization:** Threads help utilize idle CPU time during blocking operations (e.g., waiting for I/O), keeping the system more efficient.

- **Simplified Program Design**: Threads allow better management of tasks that can be executed independently, leading to cleaner and more modular code.

In summary, threads are used to optimize CPU usage, make programs more responsive, and share resources efficiently within the same process.

User-Level Threads (ULT)

- **Managed by User**: Thread management is done entirely in user space (not by the OS).
- **Kernel Unaware:** The OS kernel treats the process as a single thread.
- **Scheduling:** Handled by the user-space thread library, not by the OS scheduler.
- **Blocking:** If one thread performs a blocking operation (e.g., I/O), the entire process is blocked.
- **Advantages:** Faster to create/manage, no kernel overhead, and more flexibility.
- **Disadvantages:** Limited by the kernel (can't take full advantage of multiple CPUs), and blocking issues affect all threads in the process.

Kernel-Level Threads (KLT)

- **Managed by Kernel:** The OS kernel manages the creation, scheduling, and synchronization of threads.
- **Kernel Aware:** The kernel knows about and schedules individual threads.
- **Scheduling:** Handled by the OS kernel, allowing for better CPU utilization.
- **Blocking:** If one thread blocks (e.g., I/O), other threads in the process can still run.
- **Advantages:** Can run on multiple CPUs, independent scheduling, better handling of blocking operations.
- **Disadvantages:** More overhead due to kernel management and slower creation/management of threads compared to ULT.

Key Difference:

- **ULT:** Threads managed in user space, faster but less efficient.
- **KLT:** Threads managed by the OS kernel, more robust but with additional overhead.

Context switching is the process of saving the state (context) of a running process or thread, so that it can be paused and resumed later, and the state of another process or thread can be loaded and executed. It is a core part of multitasking in modern operating systems, enabling multiple processes or threads to share CPU time.

Key Steps in Context Switching

Saving the Context of the Current Process/Thread:

Save Registers: The values of the CPU registers (e.g., general-purpose registers, program counter, stack pointer) of the currently running process or thread are saved.
Save CPU State: The state of the current process (e.g., program counter, status flags, etc.) is saved in its Process Control Block (PCB) or Thread Control Block (TCB).
Loading the Context of the Next Process/Thread:

Load Registers: The registers and other state information of the next process/thread to be run are loaded from its PCB/TCB.
Update Program Counter: The program counter is updated to point to the next instruction of the next process/thread.
Transfer Control:

Switch Execution: The CPU starts executing the new process or thread from where it was paused, using the loaded context.

Key Components Involved in Context Switching

Process Control Block (PCB): A data structure that contains all the information about a process, such as its state, program counter, stack pointer, memory allocation, etc.

Thread Control Block (TCB): Similar to PCB but used for thread management. It stores thread-specific information, including the thread's state and context.

Stack Pointer (SP): Keeps track of the top of the stack for each thread. During context switching, the SP is saved and restored to maintain the execution context of the thread.

Program Counter (PC): Points to the next instruction to be executed. The PC is saved and restored during context switching.

Types of Context Switching

User-Level Context Switching:

Occurs when switching between user-level threads within the same process.
Faster: No kernel intervention is required, as the user-level thread library handles it.
Disadvantages: Does not take advantage of kernel-level features like multiprocessor support or I/O blocking handling.
Kernel-Level Context Switching:

Occurs when switching between processes or kernel threads.
Slower: Involves saving and restoring more information, including the entire process state (e.g., memory management, process information).
Advantages: Can fully utilize system resources, including multiple CPU cores, and handles blocking operations better (e.g., I/O).

When Does Context Switching Occur?

Interrupts: An interrupt from hardware or a system event (e.g., timer interrupt) causes the OS to save the state of the running process/thread and switch to another task.

System Calls: A process may request the OS to perform a system service (e.g., I/O), which may involve context switching to another process or thread.

Time Quantum Expiry: In preemptive multitasking, when a running process/thread's time slice (quantum) expires, the OS performs a context switch to allow another process/thread to run.

Blocking Operations: If a process/thread is blocked (e.g., waiting for I/O), the OS may perform a context switch to allow another process/thread to execute.

Context Switching Costs

Time Overhead: Saving and restoring context takes time, which can reduce CPU efficiency, especially in systems with many context switches.

Memory Overhead: Storing context information (e.g., registers, stack state) for each process/thread consumes memory.

CPU Cache Loss: Switching between tasks often leads to a loss of data in CPU caches, which can cause performance degradation due to cache misses.

Context Switching Example:
Process A is running.
An interrupt occurs, or time quantum expires, causing the OS to preempt Process A.
Context of Process A (e.g., PC, registers, stack pointer) is saved in its PCB.
Process B is selected by the scheduler.
The context of Process B is loaded from its PCB (e.g., PC, registers, stack pointer).
Process B begins executing from where it was previously suspended.

Advantages of Context Switching
Multitasking: Allows multiple processes or threads to share CPU time, enabling concurrent execution.

Responsiveness: Improves the responsiveness of the system by ensuring that high-priority tasks can preempt lower-priority tasks.

Fairness: Ensures that each process/thread gets a fair share of the CPU time, especially in time-sharing systems.

Disadvantages of Context Switching
Performance Overhead: Context switching takes time and system resources. Excessive context switching can lead to performance degradation.

Complexity: The OS must manage the context of many processes and threads, making multitasking and resource management more complex.

Summary:
Context switching is essential for multitasking systems, allowing the CPU to switch between multiple processes or threads efficiently. It involves saving and restoring the state of a process or thread, but it comes with performance costs (e.g., time and memory overhead). Effective scheduling and minimizing unnecessary context switches are crucial for optimizing system performance.

Context switching is the process of saving and restoring the state of a process or thread so that multiple processes or threads can share a single CPU.

Steps in Context Switching:

- **Save the Current Context:**

Save the current process/thread's register values, stack pointer, program counter, etc.

- **Load the Next Context:**

Load the state of the next process/thread to be executed from its Process Control Block (PCB) or Thread Control Block (TCB).

- **Transfer Control:**

The CPU executes the next process/thread from where it was paused.

Key Components:

- **PCB (Process Control Block):** Contains the state and information of the process.
- **TCB (Thread Control Block):** Stores thread-specific information.
- **Program Counter (PC):** Holds the address of the next instruction to be executed.
- **Stack Pointer (SP):** Points to the top of the stack for a thread.

Importance:

- **Multitasking:** Allows the CPU to switch between multiple processes/threads, giving the illusion of simultaneous execution.
- **Efficiency:** Enables better CPU utilization and responsiveness in a multitasking system.

CPU Scheduling is the process by which the operating system decides which process or thread should be executed by the CPU at any given time. The main goal of CPU scheduling is to maximize CPU utilization, system throughput, and responsiveness while ensuring fairness among processes.

Key Concepts in CPU Scheduling:

- **Processes and Scheduling:**
    - When multiple processes are ready to run, the operating system must decide the order in which they should be executed.
    - The scheduler handles this decision and allocates CPU time to processes based on scheduling algorithms.

Schedulers:

- Long-term scheduler: Decides which processes should be admitted into the ready queue (controls the degree of multiprogramming).
- Short-term scheduler: Selects which process from the ready queue should be executed next on the CPU.
- Medium-term scheduler: Moves processes in and out of the memory (swapping), balancing between CPU-bound and I/O-bound processes.

Types of Scheduling Queues:

- Ready Queue: Contains processes that are ready to execute but waiting for CPU time.
- Waiting Queue: Processes waiting for I/O or other resources.
- Running: The currently executing process.
- Blocked: Processes waiting for resources, like I/O.

1. **Arrival Time (AT)**:
    - Time when the process arrives in the ready queue.
2. **Burst Time (BT)**:
    - CPU time required by a process for execution.
3. **Completion Time (CT)**:
    - Time when the process finishes execution.
    - **Formula**:

$$\text{CT} = \text{Finish time of the process}$$

4. **Turnaround Time (TAT)**:
    - Total time spent in the system (from arrival to completion).
    - **Formula**:

$$\text{TAT} = \text{CT} - \text{AT}$$

5. **Waiting Time (WT)**:
    - Time spent waiting in the ready queue (not executing).
    - **Formula**:

$$\text{WT} = \text{TAT} - \text{BT}$$

6. **Response Time (RT)**:
    - Time from when the process arrives to the first execution by CPU.
    - **Formula**:

$$\text{RT} = \text{First execution time} - \text{AT}$$

When to Use Which Algorithm:

FCFS:

Simple, easy to implement.
Best for batch processing with uniform workloads.

Can lead to convoy effect (long processes delay short ones).

Round Robin (RR):

Best for interactive systems.
Ensures fair CPU allocation among processes.
Suitable when response time is critical.

Shortest Job First (SJF):

Optimal in terms of minimizing waiting time and turnaround time.
Ideal for environments where burst times are predictable.

Priority Scheduling:

Best when processes have different levels of importance.
Suitable for systems where priority levels are important (e.g., real-time systems).
Can cause starvation (low-priority tasks may never get executed).

Multilevel Queue Scheduling:

Best for systems with distinct process types (e.g., interactive, batch).
Segregates processes into different queues based on their priority and characteristics.

Multilevel Feedback Queue Scheduling:

Best for dynamic systems where processes change behavior during execution (e.g., interactive and CPU-bound tasks).
Adjusts the priority of processes based on their behavior to prevent starvation.

How to Select the Best Scheduling Algorithm
CPU Utilization:

Goal: Maximize CPU usage.
Best Algorithm: Round Robin (RR), Shortest Job Next (SJN).

Throughput:

Goal: Maximize the number of processes completed in a given time.
Best Algorithm: FCFS (First-Come, First-Served), SJF (Shortest Job First).

Turnaround Time (TAT):

Goal: Minimize the total time a process spends in the system.
Best Algorithm: SJF (Shortest Job First), SRTF (Shortest Remaining Time First).

Waiting Time (WT):

Goal: Minimize the time a process waits in the ready queue.
Best Algorithm: SJF (Shortest Job First).

Response Time (RT):

Goal: Minimize the time before a process gets CPU time (important for interactive processes).
Best Algorithm: Round Robin (RR).

Fairness:

Goal: Ensure all processes get a fair share of CPU time.
Best Algorithm: Round Robin (RR).

FCFS Scheduling

- Pros:

  Simple to implement.
  Fair, as processes are executed in the order of arrival.
  Non-preemptive, avoiding process interruption.
- Cons:

  Can cause long waiting times, especially if a long process arrives first (convoy effect).
  Poor for interactive systems where fast response is needed.
  Not efficient for mixed workloads (long processes delay short ones).

Shortest Job First (SJF) Scheduling

- Pros:

  Minimizes Average Waiting Time: Shorter jobs are executed first, reducing the waiting time for other processes.
  Optimal for Batch Systems: Works well for non-interactive systems where job lengths are known.
  Efficient for Short Tasks: Best when most tasks have relatively short burst times.
- Cons:

  Non-Preemptive: Once a process starts, it cannot be preempted, which can lead to issues with long jobs.
  Requires Knowledge of Burst Time: Accurate burst time estimation is difficult, especially for unknown jobs.
  Starvation: Long processes may never get executed if short processes keep arriving.

Shortest Remaining Time First (SRTF) Scheduling

- Pros:

  Minimizes Waiting Time: Preemptively schedules the process with the shortest remaining burst time.
  Optimal for Batch Systems: Works well when job lengths are known or predictable.
  Preemptive: Allows preemption of longer jobs by shorter jobs, improving system responsiveness.
- Cons:

  Starvation: Long processes can be indefinitely postponed if short processes keep arriving.
  Requires Burst Time Estimation: Like SJF, it requires knowledge of remaining burst times, which may be difficult to predict.
  Complex Implementation: Preemptive nature makes SRTF harder to implement compared to non-preemptive algorithms like FCFS.

Round Robin (RR) Scheduling

- **Pros:**

  Fair: Each process gets an equal share of CPU time (time quantum), preventing any process from monopolizing the CPU.
  Preemptive: Suitable for interactive systems where quick responses are important.
  Simple to Implement: Easy to understand and implement.
- **Cons:**

  Context Switching Overhead: Frequent context switches can reduce system efficiency, especially if the time quantum is too small.
  Not Optimal for Short Jobs: Short processes may have to wait for a full time quantum, increasing their waiting time.
  Increased Turnaround Time: High turnaround time for processes with varying burst times.

Priority Scheduling

- **Pros:**

  Flexible: Allows prioritization of important tasks, ensuring critical processes get CPU time first.
  Can Be Preemptive or Non-Preemptive: Can be configured to either preempt lower-priority processes or allow them to run to completion.
  Optimized for Priority-Based Systems: Suitable for real-time or critical systems where higher priority tasks need immediate execution.
- **Cons:**

  Starvation: Low-priority processes may never get CPU time if higher priority processes keep arriving.
  Requires Prioritization: Deciding the priority of processes can be complex and subjective.

Unfair for Low-Priority Tasks: If priorities are not managed well, it can lead to a lack of fairness, as low-priority processes might be indefinitely delayed.

A two-level scheduler manages processes in two stages:

- High-Level Scheduler (Job Scheduler):

  Decides which processes should be moved between memory and disk. It manages the ready queue in memory and the waiting queue on disk.
- Low-Level Scheduler (CPU Scheduler):

  Schedules processes to run on the CPU using a multi-level queue structure.
  Processes in each queue are scheduled using Round Robin within each priority level.
  The highest priority (lower number) is selected first.
- Priority Calculation:

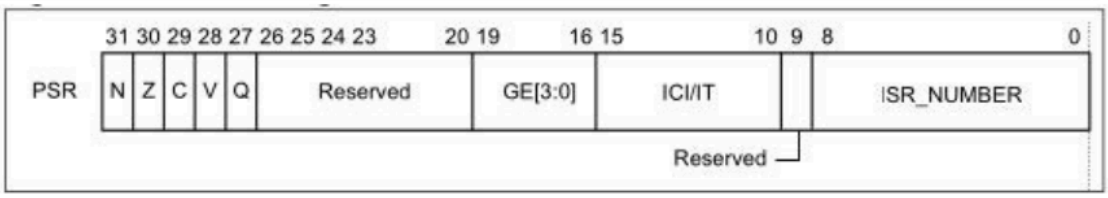
  Priority = CPU usage + nice value + base priority.
  CPU usage is based on clock ticks, decaying over time to avoid penalizing processes permanently.
  The nice value is set by the user to adjust process priority, and base priority boosts I/O-bound processes.
- Goal: To avoid starvation of low-priority processes and penalize CPU-bound processes.

Issues with Priorities in Scheduling

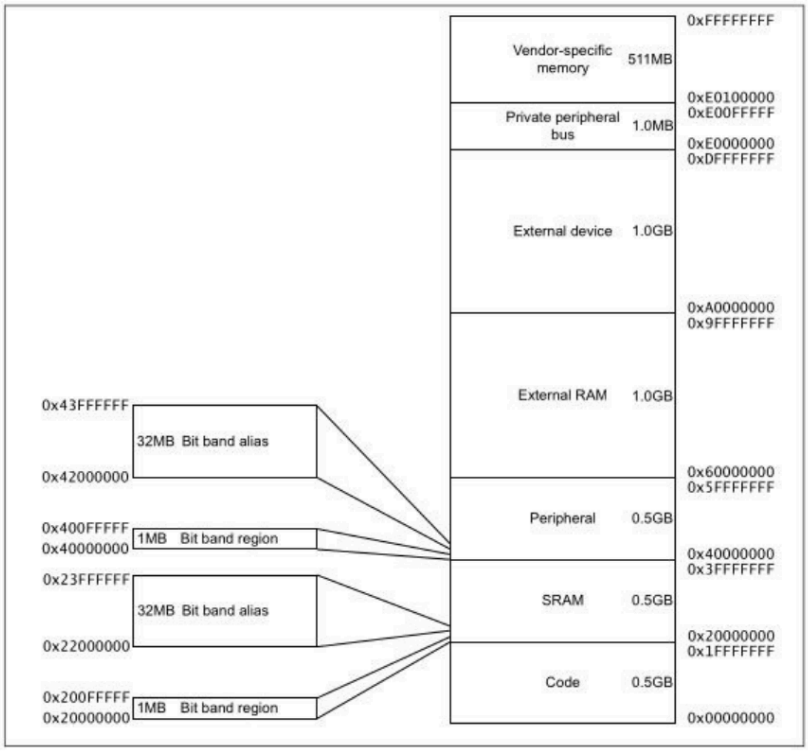
- Starvation: Low-priority processes may never get executed if high-priority tasks keep arriving.
- Priority Adaptation: Priority adjustment over time is often ad-hoc, leading to unpredictable and gradual behavior, which can be unresponsive to changes in system load.
- Unpredictable CPU Share: It's difficult to guarantee that a process will receive its desired share of CPU time due to dynamic priority changes.
- Lack of Flexibility: Applications cannot easily trade CPU time, which may lead to high-priority tasks monopolizing the CPU while low-priority ones starve.

Fair-Share Scheduling ensures that CPU time is distributed fairly among users or groups based on their allocated shares. This prevents any one user or group from consuming too much CPU time. It is particularly useful in multi-user environments but may introduce complexity and potential starvation if share management is not balanced properly.

Lottery Scheduling offers an innovative, probabilistic approach to process scheduling but lacks the predictability and guarantees provided by other algorithms like Round Robin or Priority Scheduling.



Register	Type	Combination
PSR	read-write[1], [2]	APSR, EPSR, and IPSR
IEPSR	read-only	EPSR and IPSR
IAPSR	read-write[1]	APSR and IPSR
EAPSR	read-write[2]	APSR and EPSR



- 4GB of fixed memory address
- Memory-Mapped I/O
- Bit-Banding provides automatic operation to bit data

Address range	Memory region	Memory type	XN	Description
0x00000000-0x1FFFFFFF	Code	Normal(1)	-	Executable region for program code. Can also put data here.
0x20000000-0x3FFFFFFF	SRAM	Normal(1)	-	Executable region for data. Can also put code here. This region includes bit band and bit band alias areas, see *Table 14 on page 31*.
0x40000000-0x5FFFFFFF	Peripheral	Device(1)	XN(1)	This region includes bit band and bit band alias areas, see *Table 15 on page 31*.
0x60000000-0x9FFFFFFF	External RAM	Normal(1)	-	Executable region for data.
0xA0000000-0xDFFFFFFF	External device	Device(1)	XN(1)	External Device memory
0xED000000-0xED0FFFFF	Private Peripheral Bus	Strongly-ordered(1)	XN(1)	This region includes the NVIC, System timer, and system control block.
0xED100000-0xFFFFFFFF	Memory mapped peripherals	Device(1)	XN(1)	This region includes all the STM32 standard peripherals.

The SysTick Timer is a standard timer provided by the ARM Cortex-M processors (including ARMv7-M and ARMv6-M architecture-based cores like Cortex-M4, Cortex-M3, and Cortex-M0). It is designed for real-time operating systems (RTOS) or simple timing tasks. Here's a detailed overview of the SysTick Timer in ARM Cortex-M4:

Overview of SysTick Timer
Purpose:

- Provides a basic timer for generating periodic interrupts.
- Used for RTOS task scheduling or for creating time delays in bare-metal applications.

Integration:

- Built directly into the NVIC (Nested Vectored Interrupt Controller) and processor core.
- Minimal overhead compared to external timers.

Key Features:

- 24-bit down counter (counts down from a set value to zero).
- Configurable clock source: Core clock or an external clock (divided by 8).
- Generates an interrupt when it counts down to zero.
- Automatically reloads with a preset value to continue periodic interrupts.

SysTick Timer Registers
The SysTick Timer has four main registers:

- **SysTick Control and Status Register (STK_CTRL):**

  Used to enable/disable the timer, configure the clock source, and check the count flag.
  Key bits:
  - ENABLE (bit 0): Enables or disables the timer.
  - TICKINT (bit 1): Enables or disables SysTick interrupt generation.
  - CLKSOURCE (bit 2): Selects the clock source (0 = external, 1 = core clock).
  - COUNTFLAG (bit 16): Indicates when the timer reaches zero.

SysTick Reload Value Register (STK_LOAD):

- Sets the value from which the counter starts counting down.
- Max value: 2 ^ 24 - 1

SysTick Current Value Register (STK_VAL):

- Holds the current counter value.
- Can be read to check the remaining count or written to clear the counter.

SysTick Calibration Value Register (STK_CALIB):

- Provides the calibration value for 10ms intervals (if provided by the hardware).
- Contains a NOREF flag (if the reference clock is not implemented).

How SysTick Timer Works

- Configure the reload value in STK_LOAD based on the desired time interval.
- Enable the timer and interrupt in STK_CTRL.
- The counter starts from the reload value and decrements with each clock cycle.
- When the counter reaches 0:
- The COUNTFLAG is set.
- An interrupt is generated (if enabled).
- The counter reloads the value from STK_LOAD.

Programming the SysTick Timer

Below is an example of configuring the SysTick Timer to generate a periodic interrupt:

Setup Example: 1ms Timer

Assume a system clock of 72 MHz.

1. **Calculate the Reload Value**:

$$\text{Reload Value} = \frac{\text{Timer Interval (in seconds)} \times \text{Clock Frequency}}{\text{Clock Division Factor}}$$

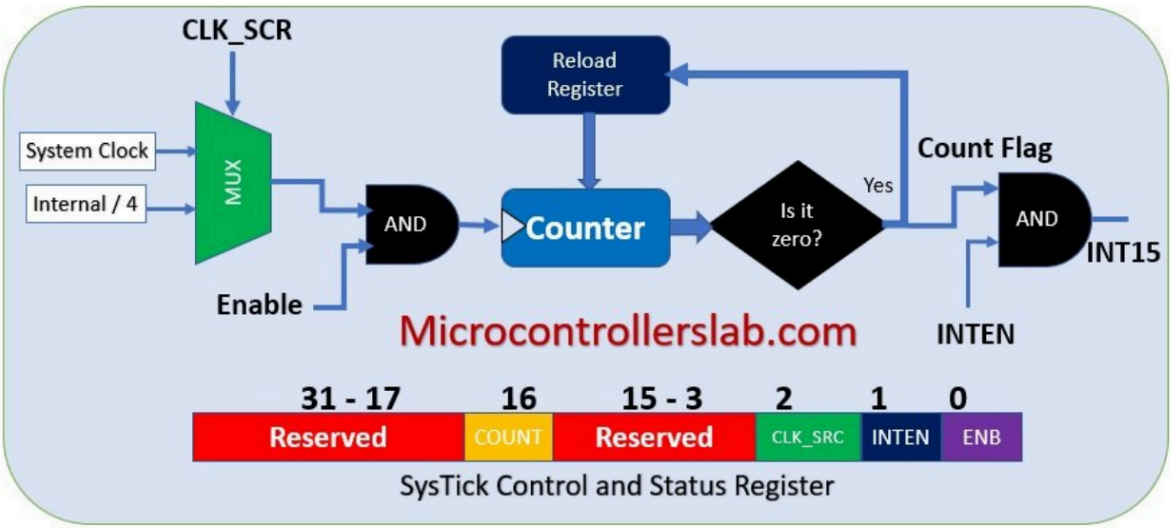
For a 1ms timer:

$$\text{Reload Value} = \frac{0.001 \times 72,000,000}{1} = 72,000$$

```
#define SYSTICK_LOAD_VAL 72000  // For 1ms interrupts at 72 MHz

void SysTick_Init(void) {
    // Load the reload value
    SysTick->LOAD = SYSTICK_LOAD_VAL - 1;
    // Clear the current value register
    SysTick->VAL = 0;
    // Enable SysTick, interrupt, and select processor clock
    SysTick->CTRL = SysTick_CTRL_ENABLE_Msk | SysTick_CTRL_TICKINT_Msk |
SysTick_CTRL_CLKSOURCE_Msk;
}

// SysTick interrupt handler
void SysTick_Handler(void) {
    // Add your periodic task here
}
```



Code to Enable an Interrupt

```c
#include "stm32f4xx.h"  // Include your microcontroller's header file

// Step 1: Set Interrupt Priority
// Set the priority of the USART2 interrupt (IRQn). The priority_level can range from 0 (highest priority) to the max value allowed.
void enable_interrupt(void) {
    NVIC_SetPriority(USART2_IRQn, 2);  // Set priority of USART2 interrupt to 2 (example priority level)
}

// Step 2: Enable Interrupt Generation in the Peripheral (USART2)
void enable_usart2_interrupt(void) {
    USART2->CR1 |= (1 << 7);  // Set the 7th bit in USART2 CR1 register to 1 to enable interrupt
}

// Step 3: Enable the Interrupt in the NVIC
void enable_irq_in_nvic(void) {
    NVIC_EnableIRQ(USART2_IRQn);  // Enable USART2 IRQ in the Nested Vector Interrupt Controller (NVIC)
}

// Step 4: Define the Interrupt Service Routine (ISR) for USART2
// This function will be called when the USART2 interrupt is triggered
void USART2_IRQHandler(void) {
    if (USART2->SR & USART_SR_RXNE) {  // Check if the receive buffer is not empty
        char received_data = USART2->DR;  // Read received data from USART2 Data
```

```
Register
        // Process the received data here (e.g., storing it, processing, etc.)
    }

    // You may need to clear other interrupt flags if required by your application.
    // USART2->SR &= ~USART_SR_RXNE;  // Clear the RXNE flag (if needed)
}

int main(void) {
    // Enable USART2 interrupt
    enable_interrupt();
    enable_usart2_interrupt();
    enable_irq_in_nvic();

    while (1) {
        // Main loop
        // The processor will enter low-power mode until an interrupt occurs
    }

    return 0;
}
```

Table 7.9 Summary of the Registers in NVIC for Interrupt Control

Address	Register	CMSIS-Core Symbol	Function
0xE000E100 to 0xE000E11C	Interrupt Set Enable Registers	NVIC->ISER [0] to NVIC->ISER [7]	Write 1 to set enable
0xE000E180 to 0xE000E19C	Interrupt Clear Enable Registers	NVIC->ICER [0] to NVIC->ICER [7]	Write 1 to clear enable
0xE000E200 to 0xE000E21C	Interrupt Set Pending Registers	NVIC->ISPR [0] to NVIC->ISPR [7]	Write 1 to set pending status
0xE000E280 to 0xE000E29C	Interrupt Clear Pending Registers	NVIC->ICPR [0] to NVIC->ICPR [7]	Write 1 to clear pending status
0xE000E300 to 0xE000E31C	Interrupt Active Bit Registers	NVIC->IABR [0] to NVIC->IABR [7]	Active status bit. Read only.
0xE000E400 to 0xE000E4EF	Interrupt-Priority Registers	NVIC->IP [0] to NVIC->IR [239]	Interrupt-Priority Level (8-bit wide) for each interrupt
0xE000EF00	Software Trigger Interrupt Register	NVIC->STIR	Write an interrupt number to set its pending status

Interrupt state

- Inactive: The exception is not active or inactive. If a higher priority interrupt preempts a lower priority active interrupt
- Pending: When an interrupt waiting to finish a higher priority or the same group interrupts in an active state
- Active: Currently serving by the processor
- Active and Pending: The MCU serves the exception by the processor, and there is a pending exception from the same source

> For an IRQ, the registers listed earlier are stacking. However, if the ISR overwrites or uses other registers, the developer should take care.

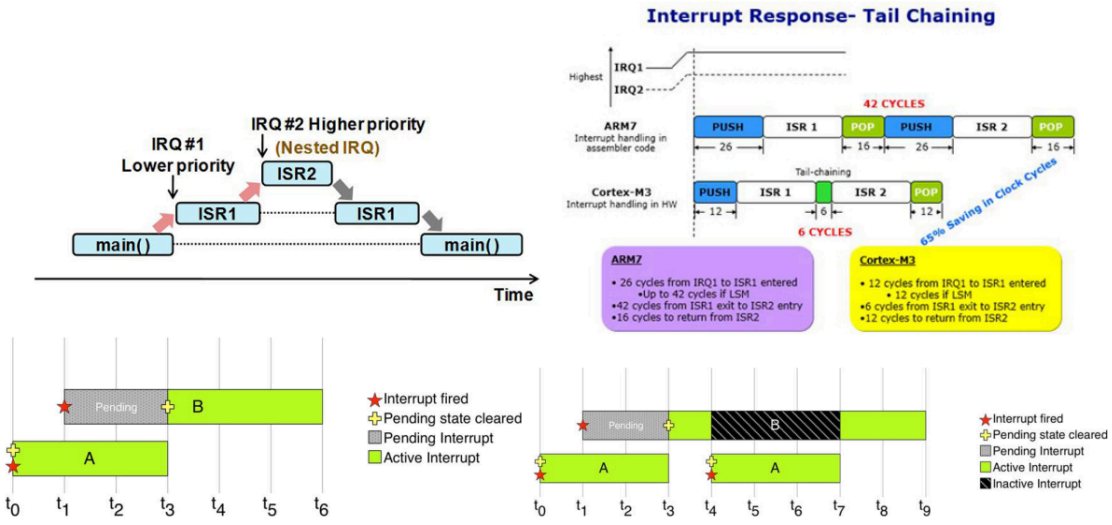


Figure 11: IRQ Life Cycle

Concurrency in an operating system refers to the ability to execute multiple processes or threads simultaneously, improving resource utilization and system efficiency. It allows several tasks to be in progress at the same time, either by running on separate processors or through context switching on a single processor. Concurrency is essential in modern OS design to handle multitasking, increase system responsiveness, and optimize performance for users and applications.

There are several motivations for allowing concurrent execution:

- Physical resource Sharing: Multiuser environment since hardware resources are limited
- Logical resource Sharing: Shared file (same piece of information)
- Computation Speedup: Parallel execution
- Modularity: Divide system functions into separation processes

Issues of Concurrency

- Non-atomic: Operations that are non-atomic but interruptible by multiple processes can cause problems.
- Race conditions: A race condition occurs of the outcome depends on which of several processes gets to a point first.

- Blocking: Processes can block waiting for resources. A process could be blocked for long period of time waiting for input from a terminal. If the process is required to periodically update some data, this would be very undesirable.
- Starvation: Starvation occurs when a process does not obtain service to progress.
- Deadlock: Deadlock occurs when two processes are blocked and hence neither can proceed to execute.

Paging is a memory management scheme that eliminates the need for a contiguous allocation of physical memory. The process of retrieving processes in the form of pages from the secondary storage into the main memory is known as paging. The basic purpose of paging is to separate each procedure into pages. Additionally, frames will be used to split the main memory. This scheme permits the physical address space of a process to be non – contiguous.

In paging, the physical memory is divided into fixed-size blocks called page frames, which are the same size as the pages used by the process. The process's logical address space is also divided into fixed-size blocks called pages, which are the same size as the page frames. When a process requests memory, the operating system allocates one or more page frames to the process and maps the process's logical pages to the physical page frames.

The mapping between logical pages and physical page frames is maintained by the page table, which is used by the memory management unit to translate logical addresses into physical addresses. The page table maps each logical page number to a physical page frame number.

The layered approach

a. Processor allocation and multiprogramming
b. Memory Management
c. Devices
d. File system
e. Users

Each layer depends on the the inner layers

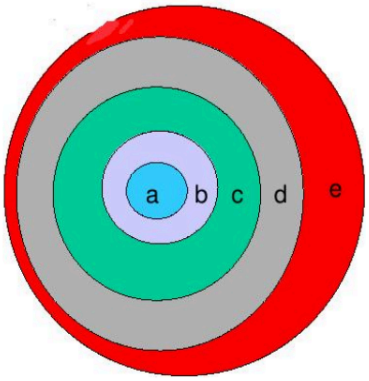
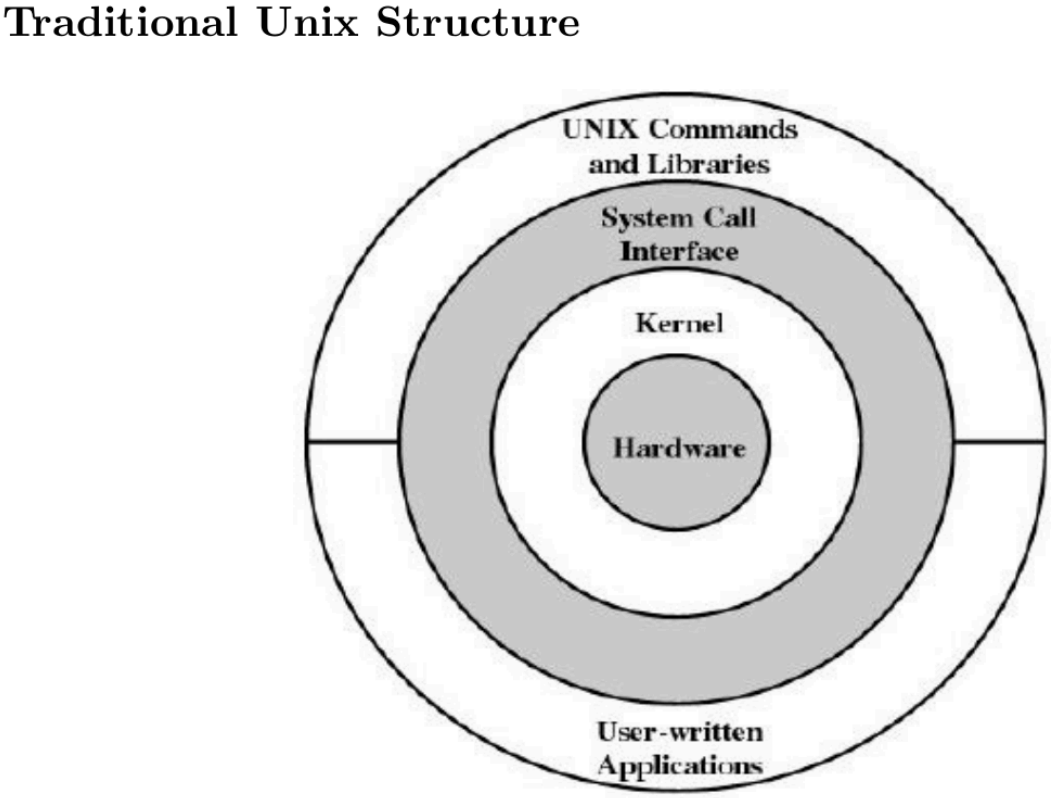


Figure 6

A monolithic kernel is an operating system kernel in which all the operating system services run in kernel space, meaning they all share the same memory space. This type of kernel is characterized by its tight integration of system services and its high performance.



A microkernel is a type of operating system kernel in which only the most basic services run in kernel space, with other services running in user space. This type of kernel is characterized by its modularity, simplicity, and ability to run multiple operating systems on the same hardware.

Monolithic Kernel
Definition: A monolithic kernel is an OS architecture where the entire kernel (system calls, device drivers, file systems, memory management) is compiled into a single large program that runs in kernel space.

Key Features:

- Single Address Space: All components share the same memory space.
- High Performance: Direct communication between kernel components without overhead.
- Large and Integrated: Includes device drivers, process management, etc.
- No Separation: Kernel and user-space applications interact directly, which can be risky if there's a bug.

Advantages:

- Faster: Fewer context switches and less overhead for system calls.
- Efficient: Direct access to hardware and resources.

Disadvantages:

- Complex and Risky: All components run in kernel mode, so bugs can crash the entire system.
- Less Flexibility: Harder to add or remove components dynamically.

Microkernel
Definition: A microkernel is an OS architecture where only the most basic functionalities (like communication, memory management) run in kernel mode. Non-essential components (e.g., device drivers, file systems) run in user space as separate processes.

Key Features:

- Minimal Kernel: Only essential functions like IPC (Inter-Process Communication) and memory management run in kernel mode.
- Modular Design: Other services like device drivers and file systems run as user-space processes.
- Isolation: Communication between kernel and user-space is done via message passing.

Advantages:

- Stability: Faults in user-space services do not crash the whole system.
- Modular: Easier to add/remove components.

Disadvantages:

- Performance Overhead: Message-passing between user-space and kernel adds latency.
- Complexity: More complex to implement due to the need for inter-process communication.

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Figure 17: Unix/Win32 Syscalls

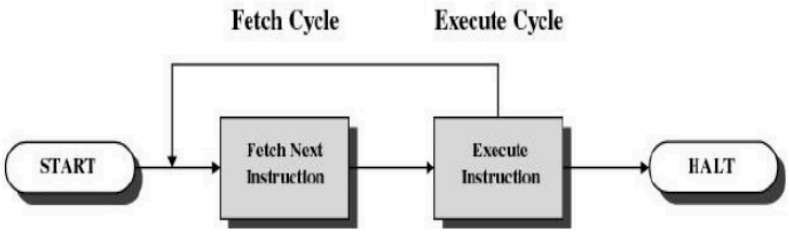


Figure 4: Simple Fetch Cycle

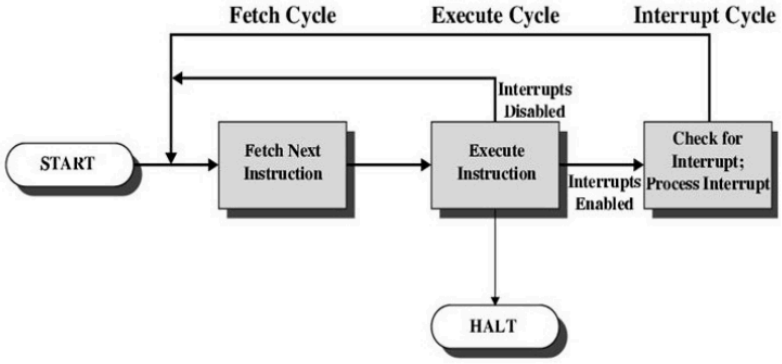


Figure 5: Simple Fetch Cycle with Interrupt

The accessibility of addresses within an address space changes depending on operating mode

- To protect kernel code and data

Mode	Mode Identifier
User	usr
*Fast Interrupt	fiq
*Supervisor	svc
*Abort	abt
*System	sys
*undefined	und

Table 1: Arm Processor Operating Mode

*Similar to kernel mode or handled by the kernel

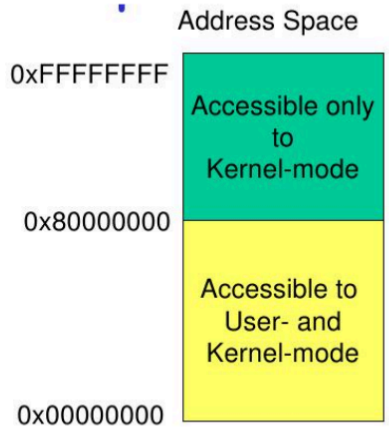


Figure 10: Microprocessor address space access

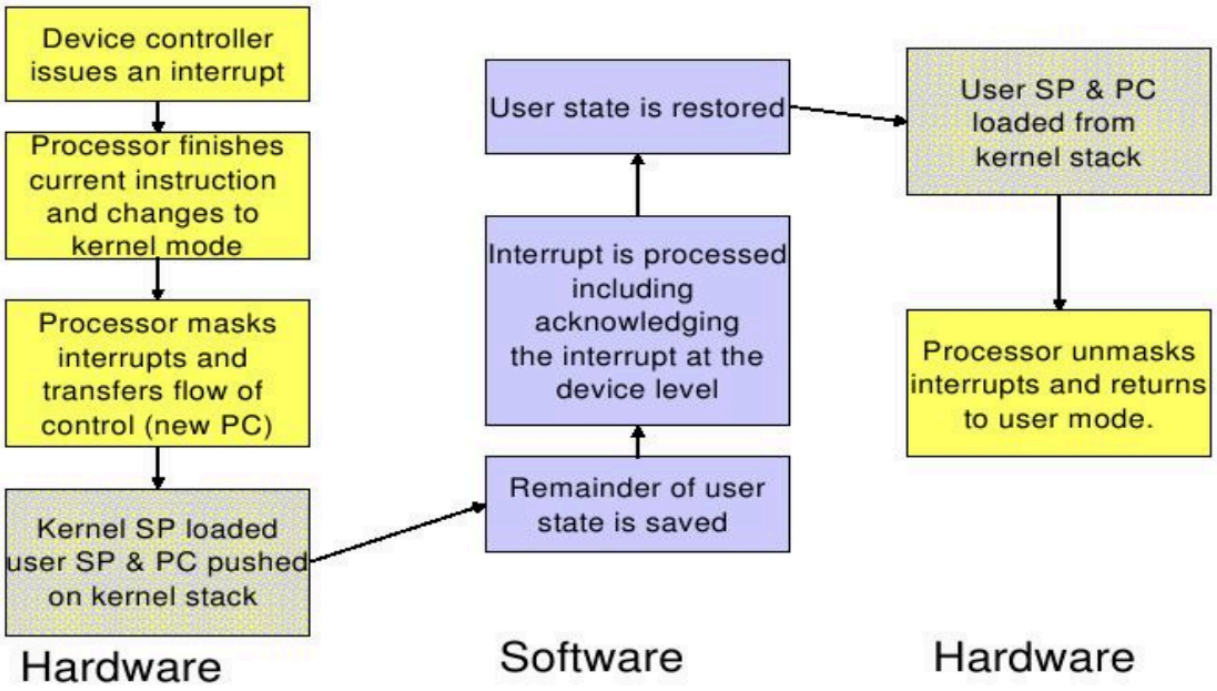


Figure 13: Interrupt Processing

Effective Access Time

$$T_{eff} = H \times T_1 + (1 - H) \times (T_1 + T_2) \quad (1)$$

where,
T_1 = Access Time of memory1
T_2 = Access Time of memory2
H = hit ratio of memory1
T_{eff} = effective access time of the system

Lab task:

```
Total = (64 + 448) K
Bootloader Start address = 0x08000000
Find OS start address = Bootloader Start address + Bootloader size ; note 1KB =
1024B

MEMORY
{
```

```
        FLASH(RX): ORIGIN = 0x08000000, LENGTH = 64K
        SRAM(RWX): ORIGIN = 0x20000000, LENGTH = 128K /* combined both SRAM1 and
SRAM2 */
}

MEMORY
{
        FLASH(RX): ORIGIN = 0x08010000, LENGTH = 448K
        SRAM(RWX): ORIGIN = 0x20000000, LENGTH = 128K /* combined both SRAM1 and
SRAM2 */
}
```

Crc config:

```
void CRC_Config(void) {
    RCC->AHB1ENR |= RCC_AHB1ENR_CRCEN;  // Enable CRC clock by setting the CRCEN
bit in AHB1ENR register 1 << 12
    CRC->CR = CRC_CR_RESET;            // Reset the CRC calculation unit,
initializing it to 0xFFFF FFFF 1 << 0
}

void CRC_Reset(void) {
    CRC->CR = CRC_CR_RESET;            // Reset the CRC calculation unit,
initializing it to 0xFFFF FFFF
}
uint32_t CRC_Calc(uint8_t *data, uint32_t length) {
    // Loop through each byte in the data array
    for (uint32_t i = 0; i < length; i += 1) {
        CRC->DR = (uint32_t)data[i];   // Write each byte into the CRC data
register (DR)
    }

    return (CRC->DR);  // Return the computed CRC value from the CRC data register
}
```

Pseudocode (Basic CRC Algorithm):

```
1. Initialize CRC register with the predefined value (often 0xFFFFFFFF or
0x00000000).
2. For each byte of data:
   3. Load the byte into the CRC register (combine it with the CRC register).
   4. For each bit of the byte (starting from the most significant bit):
      5. Shift the CRC register one bit left.
      6. If the shifted-out bit is 1, XOR the CRC register with the CRC polynomial
(e.g., 0x04C11DB7 for CRC-32).
7. After processing all bytes, the value in the CRC register is the final CRC.
```

Jamp to os:

```
void jump_to_os(void) {
    typedef void (*void_fn)(void);  // Define a function pointer type for void
functions
    uint32_t* reset_vector_entry = (uint32_t*)(OS_START_ADDRESS + 4U); // Address
of the reset vector entry in the OS
    uint32_t* reset_vector = (uint32_t*)(*reset_vector_entry);         //
Dereference to get the OS reset vector
    void_fn jump_fn = (void_fn)reset_vector;                           // Cast the
reset vector to a function pointer
    jump_fn();                                                         // Call the
function to jump to the OS
}
```

Pseudocode:

```
1. Define function pointer type: void_fn = function that returns void and takes no
arguments
2. Set reset_vector_entry to the address of the OS reset vector (OS_START_ADDRESS +
4)
3. Set reset_vector to the value at reset_vector_entry (i.e., the reset vector
address)
4. Cast reset_vector to function pointer jump_fn
5. Call jump_fn to transfer control to the OS
```