(a)

i. **Design Method for Data Transfer**

- Byte Transfer: Use interrupt-driven I/O or polling for sending/receiving small data (bytes).
- Megabyte Transfer: Use DMA (Direct Memory Access) for larger data transfers, which avoids CPU intervention.
- Sending Status: Track transfer status using a status register or buffer and update after transfer completion via interrupts or polling.

ii. **Method to Improve System Performance (Speed Differences)**

- Caching: Use cache memory to reduce access time between CPU and main memory.
- Virtual Memory: Implement paging or segmentation to handle slower secondary storage (disk) as if it were faster memory.
- DMA: Use Direct Memory Access for efficient data transfer between memory and peripherals.

b)

Reasons for Monolithic Kernels Over Microkernels

- Performance: Monolithic kernels have fewer context switches and are faster due to minimal communication overhead between kernel components.
- Simplicity: A monolithic kernel is often simpler to design and implement, with all system services in one place.
- Compatibility: Monolithic kernels have broad hardware support and are easier to maintain with fewer communication layers.

c)

Threads and Performance

- Granularity: Threads allow parallel execution and improve performance by dividing tasks into smaller units.
- Performance Decrease: In some cases, thread management and context switching can introduce overhead, especially with too many threads or when tasks are too small (e.g., distribution of on-gram sweets among 40 people is inefficient due to overhead in dividing and managing the sweets). Hence, excessive threading may lead to reduced performance due to resource contention.

Variable Locations

- Initialized Global Variables: Stored in the data section of memory.
- Constant or Final Variables: Stored in the read-only section or text section.
- Process Stack Frame: Stored in the stack (dynamic memory for function calls, local variables).
- Local Variables: Stored in the stack.
- Functions: Stored in the text (code) section.

Determine Access Level on SVC

The OS can determine the access level (user or kernel) by checking the Control Register (CONTROL) or Saved Program Status Register (xPSR). Specifically, the SPSEL bit in CONTROL indicates whether the stack pointer is set to PSP (user

mode) or MSP (kernel mode). Based on this, the OS can infer if the call originated from user mode or kernel mode.

Steps to adopt System Call and Scheduling for Multitasking on ARM32F4xxx System

(i) System Call Implementation:

- Define System Call Numbers: Assign unique numbers to system calls (e.g., SVC #0 for read()).
- Write System Call Handler: Implement an SVC Handler that processes system call numbers and arguments.
- Trigger System Call: Application triggers system call using SVC #<call_number>.
- Switch to Supervisor Mode: SVC triggers an exception, switching from Thread mode to privileged Handler mode.
- Return from System Call: After handling, the system restores control to the calling process using an exception return.

(ii) Scheduling for Multitasking:

- Context Switch: Save the current process's state and load the next process state (using PSP or MSP).
- Interrupt Handling: Use interrupts like PendSV for task switching.
- Preemptive Scheduling: Use a timer interrupt (SysTick) to preemptively switch between tasks.
- Task Management: Maintain task control blocks (TCBs) for each task, including its state, stack, and scheduling priority.
- Scheduling Algorithm: Implement a scheduling algorithm (e.g., Round Robin) to decide which task to run next.

(b) **Process States for Events:**

- External Interrupt: Process moves to Interrupt state (ISR is executed).
- Systick & PendSV: Process enters Waiting state or Ready state, depending on the interrupt context.
- Invalid Memory Access: Process enters Terminated or Crash state due to segmentation fault (exception).
- Process Printing to UART: Process is in Running state while the UART operation executes, may enter Blocked state if waiting for I/O.

To determine the service number of a system call (SVC #5):

- SVC Instruction Execution: When a user process calls SVC #5, it triggers an exception.

- Stack Push: The CPU pushes the context, including the PC (Program Counter) and XPSR (Program Status Register), onto the stack.

- Accessing the Service Number:

The SVC instruction's immediate value (#5) is stored in the memory location pointed to by PC - 2.
The kernel can access this value by reading the instruction at PC - 2 to retrieve the system call number.

- Final Step: The kernel can then use this number to identify which system call (SVC) the user requested to execute.

Comparing Application and Kernel Level Thread Implementation
Application-Level Threads:

- Managed by user-level libraries.
- Switching is faster as it doesn't involve switching between privileged and unprivileged mode.
- Threads are unaware of kernel and cannot access kernel services directly.

Kernel-Level Threads:

- Managed by the OS kernel.
- Context switching takes more time as it involves mode switching (5ms from unprivileged to privileged mode).
- Can directly access kernel services and hardware.

Conclusion: Kernel-level threads are slower (due to mode switching) but are more flexible and can interact with the kernel. Application-level threads are faster but limited in functionality.

Contents of Interrupt Vector Table and Microprocessor Steps
Contents of the Interrupt Vector Table:

- The interrupt vector table contains the addresses of service routines for each interrupt (like USART, SVC, etc.).
- The system uses this table to jump to the appropriate interrupt service routine (ISR) when an interrupt occurs.

Steps when USART Interrupt is Triggered with BASEPRI = 5:

- The microprocessor checks the interrupt priority.
- If the priority of the USART interrupt is greater than or equal to 5, the interrupt is serviced.
- The CPU pushes the current context (registers) to the stack.
- The CPU jumps to the interrupt handler address from the interrupt vector table.
- After executing the ISR, the CPU pops the saved context from the stack and resumes the main program.

How can you speed up paging using TLB?
Answer:
Use of TLB (Translation Lookaside Buffer):

The TLB stores recently used virtual-to-physical page mappings. By using the TLB, the operating system can reduce the time spent on page table lookups, as TLB entries can be accessed much faster than reading from main memory.
TLB hit: If the required page mapping is found in the TLB, the access time is significantly reduced.
TLB miss: If the mapping is not found, a slower page table lookup is performed, but the TLB is updated with the new mapping for future access.
Result: The TLB helps reduce the effective access time for paging, leading to faster memory access and improved performance.

Process States for the Following Events:

Process experiencing page fault:

- State: Blocked/Waiting
- Reason: The process is waiting for the required page to be loaded into memory.

An 'EXTR' interrupt occurs:

- State: Interrupted
- Reason: The currently running process is interrupted, and control is transferred to the interrupt service routine (ISR).

Context switching:

- State: Blocked or Ready
- Reason: The process may either be moved to the ready queue or blocked depending on the scheduling decision.

Unsuccessful attempt to enter a critical section:

- State: Blocked
- Reason: The process is blocked because it couldn't acquire the necessary lock for the critical section.

Which Methodology (Microkernel vs Monolithic Kernel) Should Be Preferred for Smart Home Management?

Answer:

Preferred Method: Microkernel
Reasons:

- Modularity: Easier to develop and maintain due to separate modules for different services (e.g., lighting, temperature control).
- Fault Tolerance: Faults in one service won’t crash the entire system as they run in isolated processes.

- Security: Better security by isolating device drivers and other critical services in user space, reducing the attack surface.
- Flexibility: Easier to update and add new features without impacting the whole system.

(a) OS Design: Monolithic Kernel vs. Microkernel

- (i) Convenient access to operating system data structures by the kernel-level process:

  Answer: Monolithic Kernel
  Reason: Direct access to kernel data structures without needing IPC, as everything runs in the same address space.
- (ii) Adding/modifying operating system components by kernel developers:

  Answer: Monolithic Kernel
  Reason: Easier to modify or add components since everything runs in one address space, reducing complexity.
- (iii) Strong security and reliability:

  Answer: Microkernel
  Reason: Better isolation of services, reducing the risk of crashes and security vulnerabilities as services are separated.

(b) **User-Level vs Kernel-Level Operations**

- (i) Disable all interrupts:

  Answer: Kernel-Level
  Reason: Disabling interrupts is a privileged operation that can affect system stability, requiring kernel access.
- (ii) scanf(...):

  Answer: User-Level
  Reason: scanf is a standard input operation and does not need kernel privileges, so it runs in user space.

(c) **Passing Parameters in a System Call (Matrix Multiplication)**

- Parameter to pass:

  Answer: Pointers to matrix data and size information
  Reason: The user program must pass pointers to the matrices and their dimensions to the system call for processing by the kernel.
  Steps to pass parameters:

  User Mode: Program prepares the matrix data and computes the size.
  System Call: The program places the pointers to the matrices and their size into registers (or stack).
  Transition to Kernel: A system call interrupt (e.g., SVC) triggers the kernel.
  Kernel Mode: The kernel retrieves the parameters (from registers or stack) and invokes the DSP unit to multiply the matrices.

(a) **Benefits and Disadvantages**

- i) Synchronous vs Asynchronous Communication

  Synchronous:
  Benefits: Easy to implement, ensures consistency.
  Disadvantages: Blocks sender, reduces performance.
  Asynchronous:
  Benefits: Non-blocking, improves throughput.
  Disadvantages: Complex, may cause data inconsistency.
- ii) Automatic vs Explicit Buffering

  Automatic Buffering:

  Benefits: Simpler, less programming effort.
  Disadvantages: Less control, potential overhead.
  Explicit Buffering:

  Benefits: More control, better performance.
  Disadvantages: Increases complexity, more error-prone.

(b) **Multi-Core System Performance Implications**

- i) Kernel Threads < Cores:

  mplication: Underutilizes CPU, potential idle cores.
- ii) Kernel Threads = Cores:

  Implication: Efficient CPU usage, optimal performance.
- iii) Kernel Threads > Cores < User Threads:

  Implication: Thread contention, context switching overhead.

(c) **Shared Program State Across Threads**

- Shared:

  Heap memory
  Global variables
- Not Shared:

  Register values
  Stack memory

(a) **Relation Between Algorithm Sets**

- Priority and SJF: If priority is based on burst time, SJF is a special case of Priority Scheduling.
- Multilevel Feedback Queues and FCFS: MLFQ uses multiple queues and can include FCFS at lower priority levels, so FCFS is a subset of MLFQ.
- Priority and FCFS: Independent algorithms. FCFS schedules by arrival time, Priority schedules based on priority values. They can coexist but are not directly related.

- RR and SJF: RR (Round Robin) is a time-slice based algorithm, while SJF schedules the shortest burst time. SJF is non-preemptive, while RR is preemptive.

(b) **Conflicting Scheduling Criteria**

- CPU Utilization vs. Response Time: Maximizing CPU utilization can increase waiting time and reduce responsiveness.
- Average Turnaround Time vs. Maximum Waiting Time: Minimizing average turnaround time may increase maximum waiting time, especially for longer processes.
- I/O Utilization vs. CPU Utilization: Optimizing CPU utilization can leave I/O devices idle, and maximizing I/O device usage can lead to underutilized CPU.