**Chapter 6 :: Limited Direct Execution**

**User mode**

**Kernel mode**

**System calls**

**Trap instructions to trap into the kernel and return-from trap back into user mode**

**Trap tables and trap handlers**

**Chapter 7 :: Scheduling**

* **Scheduling Algorithms**
  + **First In, First Out** (**FIFO**) or sometimes **First Come, First Served** (**FCFS**)
    - Problem: **convoy effect** – where a number of relatively-short potential consumers of a resource get queue behind a heavy resource consumer.
  + **Shortest Job First (SJF)** 
    - it runs the shortest job first, then the next shortest, and so on.
    - Also suffers from convoy effect (jobs not arriving at the same time)
    - Bad for response time
  + **Shortest Time-to-Completion First** (**STCF**) or **Preemptive Shortest Job First** (**PSJF**)
    - Any time a new job enters the system, the STCF scheduler determines which of the remaining jobs (including the new job) has the least time left, and schedules that one.
    - STCF would preempt A and run B and C to completion; only when they are finished would A’s remaining time be scheduled.
    - Mitigates convoy effect
    - Bad for response time
  + **Round Robin**
    - instead of running jobs to completion, RR runs a job for a **time slice** (sometimes called a **scheduling quantum**) and then switches to the next job in the run queue. It repeatedly does so until the jobs are finished.
    - Note that the length of a time slice must be a multiple of the timer-interrupt period; thus if the timer interrupts every 10 milliseconds, the time slice could be 10, 20, or any other multiple of 10 ms.
    - Time slice – The shorter it is, the better the performance of RR under the response-time metric. making it too short is problematic. Cost of context switching will dominate overall performance.
    - RR is one of the *worst* policies if turnaround time is our metric.
* **Tresponse** = Tfirstrun − Tarrival
* **Tturnaround** = Tcompletion − Tarrival
* **context switch** – stopping one running process temporarily and resuming (or starting) another.

**Chapters 26/27 :: Introduction to Concurrency**

**Motivations for Concurrency:**

1. **Parallelism:**
   * **Motivation:** Achieve faster execution by simultaneously executing multiple tasks.
   * **Example:** Parallel processing of data, where different parts of a large dataset are processed concurrently.
2. **Responsiveness:**
   * **Motivation:** Ensure that an application remains responsive to user input, even when performing lengthy operations.
   * **Example:** Running background tasks while still allowing a user to interact with a graphical user interface.
3. **Resource Utilization:**
   * **Motivation:** Efficiently use system resources by allowing tasks to execute in parallel, maximizing CPU and I/O device utilization.
   * **Example:** Concurrently handling multiple network requests in a server application.
4. **Modularity:**
   * **Motivation:** Divide a complex system into smaller, manageable, and independent units that can be developed, tested, and maintained separately.
   * **Example:** Designing a system with different modules or components that can run concurrently.

**Problems and Challenges of Concurrency:**

1. **Race Conditions:**
   * **Problem:** Unpredictable behavior when multiple threads access shared data concurrently without proper synchronization.
   * **Challenge:** Ensuring proper synchronization to avoid conflicts and maintain data consistency.
2. **Deadlocks:**
   * **Problem:** Threads are blocked indefinitely because each is waiting for the other to release a resource.
   * **Challenge:** Implementing strategies to prevent and resolve deadlocks.
3. **Starvation:**
   * **Problem:** Some threads are denied access to resources for an extended period, leading to reduced overall system performance.
   * **Challenge:** Fairly allocating resources among competing threads to prevent starvation.
4. **Priority Inversion:**
   * **Problem:** Lower-priority threads holding resources needed by higher-priority threads, causing delays.
   * **Challenge:** Implementing mechanisms to prevent or minimize priority inversion.

**Basics of the Threading API:**

1. **Thread Creation:**
   * **API Calls:** Typically, programming languages provide functions or classes for creating threads. Examples include Thread class in Java or pthread\_create in C.
2. **Thread Synchronization:**
   * **API Features:** Mutexes, semaphores, and condition variables are common synchronization primitives provided by threading APIs.
   * **Usage:** Protecting critical sections of code and coordinating access to shared resources.
3. **Thread Joining:**
   * **API Feature:** Ability to wait for the completion of a thread using functions like join in high-level threading APIs.
   * **Usage:** Ensuring that one thread completes its task before another continues.
4. **Thread Priority:**
   * **API Features:** Setting and adjusting the priority of threads to influence their scheduling.
   * **Usage:** Controlling the order in which threads are scheduled to run.
5. **Thread Safety:**
   * **API Guidelines:** Threading APIs often provide guidelines and best practices for writing thread-safe code.
   * **Usage:** Adhering to these guidelines to avoid data corruption and ensure correctness.
6. **Thread Pooling:**
   * **API Features:** High-level threading APIs may include features for managing thread pools.
   * **Usage:** Efficiently reusing threads to avoid the overhead of thread creation and destruction.

**Chapters 28/29 :: Locks**

1. **Load-Link/Store-Conditional (LL/SC):**
   * **Motivation:** Designed to support optimistic concurrency control.
   * **Concept:**
     + **LL** loads a value from memory into a register.
     + **SC** stores a value back to memory only if the content of the memory location has not changed since the corresponding **LL**.
   * **Usage:**
     + Used in some multiprocessor architectures for implementing locks or managing shared data in a way that minimizes contention.
2. **Compare-and-Swap (CAS):**
   * **Motivation:** Provides a way to update a memory location only if it still holds a particular expected value.
   * **Concept:**
     + Compares the current value in memory with an expected value.
     + If the values match, it swaps the current value with a new value.
   * **Usage:**
     + Building blocks for implementing lock-free data structures, such as linked lists, queues, and more.
3. **Fetch-and-Add (FAA) / Fetch-and-Increment (FAI):**
   * **Motivation:** Allows for atomic increment operations.
   * **Concept:**
     + Atomically adds a specified value to the content of a memory location.
   * **Usage:**
     + Useful for implementing counters and managing shared resources where increment operations need to be performed atomically.
4. **Test-and-Set (TAS):**
   * **Motivation:** Ensures mutual exclusion for critical sections.
   * **Concept:**
     + Atomically sets a flag in memory and returns the previous value.
   * **Usage:**
     + Building blocks for implementing locks or semaphores to coordinate access to shared resources.
5. **Memory Barriers (Fence Instructions):**
   * **Motivation:** Controls the order of memory operations in a multithreaded environment.
   * **Concept:**
     + Ensures that memory operations before the barrier are visible to other threads before those after the barrier.
   * **Usage:**
     + Ensures proper visibility of shared data and enforces ordering constraints to prevent race conditions.

**Spin Locks:**

1. **How Spin Locks Work:**
   * **Lock Acquisition:**
     + A thread attempting to acquire a spin lock will repeatedly check if the lock is available.
     + If the lock is available, the thread atomically acquires the lock and proceeds with the critical section.
     + If the lock is not available, the thread "spins" or loops until it can acquire the lock.
   * **Lock Release:**
     + The thread releases the spin lock when it completes its critical section.
     + Other threads waiting for the lock will detect its release and attempt to acquire it.
   * **Pros:**
     + Simple and lightweight.
     + No context-switching overhead when the lock is not available.
   * **Cons:**
     + Wasteful spinning (busy waiting) can be inefficient, especially in high-contention scenarios.
     + Potential for priority inversion if a higher-priority thread is waiting for the lock held by a lower-priority thread.

**Alternatives to Spin Locks:**

1. **Mutex (Mutual Exclusion Lock):**
   * **How Mutex Works:**
     + Similar to a spin lock, but with the ability to put the waiting thread to sleep rather than continuously checking.
     + When a thread cannot acquire a mutex, it is put to sleep, and the operating system scheduler can assign CPU time to another thread.
     + The sleeping thread is awakened when the mutex becomes available.
   * **Pros:**
     + Reduces CPU usage and avoids wasteful spinning.
     + Effective in scenarios with low contention.
   * **Cons:**
     + Slightly higher overhead due to context switching.
2. **Semaphore:**
   * **How Semaphore Works:**
     + A semaphore is a counter that can be used to control access to a resource by multiple threads.
     + It allows a specified number of threads to access the critical section simultaneously.
     + Threads attempting to enter the critical section decrement the semaphore, and when the semaphore reaches zero, subsequent threads are blocked.
   * **Pros:**
     + More versatile than a simple lock, allowing control over the number of concurrent accesses.
   * **Cons:**
     + Complexity may increase with more advanced use cases.
3. **Reader-Writer Lock:**
   * **How Reader-Writer Lock Works:**
     + Allows multiple readers to access a resource concurrently, but exclusive access is granted to a single writer.
     + Balances the need for concurrency with the need for exclusive updates.
   * **Pros:**
     + Efficient for scenarios where reads significantly outnumber writes.
   * **Cons:**
     + Complexity and potential for priority inversion.
4. **Condition Variables:**
   * **How Condition Variables Work:**
     + Used for signaling between threads.
     + A thread can wait on a condition variable until another thread signals that a particular condition has been met.
   * **Pros:**
     + Allows for more sophisticated synchronization patterns.
   * **Cons:**
     + Requires careful usage to avoid potential issues like spurious wake-ups.

**Chapters 36, 37, 39, 40 :: I/O and File Systems**

**Status register – can be read to see the current status of the device**

**Command register – to tell the device to perform a certain task**

**Data register – to pass data to the device, or get data from the device**

**Four Steps:**

1. **The OS waits until the device is ready to receive a command by repeatedly reading the status register; we call this polling the device.**
2. **The OS sends some data down to the data register. When the main CPU is involved with the data movement, we refer to it as programmed I/O (PIO).**
3. **The OS writes a command to the command register; doing so implicitly lets the device know that both the data is present and that it should begin working on the command.**
4. **The OS waits for the device to finish by again polling it in a loop, waiting to see if it is finished (it may then get an error code to indicate success or failure).**

**Interrupts:**

**Instead of polling the device repeatedly, the OS can issue a request, put the calling process to sleep, and context switch to another task. When the device is finally finished with the operation, it will raise a hardware interrupt, causing the CPU to jump into the OS at a predetermined interrupt service routine (ISR) or more simply an interrupt handler. The handler is just a piece of operating system code that will finish the request (for example, by reading data and perhaps an error code from the device) and wake the process waiting for the I/O, which can then proceed as desired.**

**Interrupts thus allow for overlap of computation and I/O, which is key for improved utilization.**

**Disk Geometry:**

**Platter – a circular hard surface on which data is stored persistently by inducing magnetic changes to it.**

* **A disk may have one or more platters; each platter has 2 sides, each of which is called a surface.**
* **are all bound together around the spindle.**
* **rotations per minute (RPM)**
* **Data is encoded on each surface in concentric circles of sectors; we call one such concentric circle a track.**

**Disk head –**

* **To read and write from the surface.**
* **there is one such head per surface of the drive.**
* **Is attached to a single disk arm, which moves across the surface to position the head over the desired track.**
  + **Outer tracks tend to have more sectors than inner tracks.** **Often referred to as multi-zoned disk drives**

**Cache, sometimes called a track buffer – is just some small amount of memory (usually around 8 or 16 MB) which the** drive can **use to hold data read from or written to the disk.**

**Disk Scheduling:**

* **Shortest-Seek-Time-First (SSTF) (also called shortest-seek-first or SSF). SSTF orders the queue of I/O requests by track, picking requests on the nearest track to complete first.**
  + Services the request with the shortest seek time first.
  + Aims to minimize the amount of head movement, improving response time.
  + May result in starvation for some requests.
* **Elevator (a.k.a. SCAN or C-SCAN) moves back and forth across the disk servicing requests in order across the tracks.**
  + **HANDLES DISK STARVATION**
  + Disk arm moves in one direction servicing requests until the end of the disk is reached, then reverses direction.
  + May lead to increased average response time for requests farther from the current position of the disk arm.
  + **C-SCAN** 
    - Similar to SCAN but only scans in one direction, then jumps to the other end without servicing requests.
    - Reduces the variance in response time compared to SCAN.
* **SPTF: Shortest Positioning Time First also called shortest access time first or SATF**
  + **ACCOUNT FOR DISK ROTATION COSTS**
  + The algorithm selects the disk request that requires the shortest positioning time.
  + Prioritizes requests based on the physical location of the data on the disk.

**How Unix works with files:**

1. **File System Hierarchy:**
   * Unix organizes files in a hierarchical file system, with the root directory ("/") at the top.
   * Directories contain files and subdirectories, forming a tree-like structure.
2. **File Types:**
   * Unix treats everything as a file, including regular files, directories, devices, and special files.
   * Files are classified into various types, such as text files, binary files, directories, symbolic links, and device files.
3. **File Permissions:**
   * Unix uses a permission system to control access to files.
   * Permissions are represented for user (owner), group, and others (everyone else) as read, write, and execute.
   * Commands like **chmod** and **chown** are used to modify file permissions.
4. **File Manipulation Commands:**
   * Common commands for file manipulation include:
     + **ls**: List files and directories.
     + **cp**: Copy files.
     + **mv**: Move or rename files.
     + **rm**: Remove files.
     + **touch**: Create an empty file or update the access/modification time of a file.
5. **File Viewing and Editing:**
   * **cat**, **more**, and **less** are used to display file contents.
   * **vi** or **vim** and **nano** are popular text editors.
   * Output redirection **(>** and **>>)** is used to redirect command output to a file.
6. **File System Navigation:**
   * ‘**cd**’: Change directory.
   * ‘**pwd**’: Print working directory.
   * ‘**..**’**:** Represents the parent directory.
7. **Symbolic Links:**
   * Unix supports symbolic links, which are references to other files or directories.
   * **ln -s** is used to create symbolic links.
8. **File Attributes:**
   * **ls -l** shows detailed information about files, including permissions, owner, group, size, modification time, and more.
   * **stat** provides even more detailed information.
9. **File System Integrity:**
   * Unix uses inodes (index nodes) to store metadata about files.
   * The **fsck** (file system check) command is used to check and repair file system integrity.
10. **Special Files:**
    * Device files in **/dev** represent hardware devices.
    * **/proc** contains information about processes.
11. **File Compression and Archiving:**
    * **tar** is used for file archiving.
    * **gzip** and **bzip2** are used for compression.

**Basic Relationships in a File System:**

1. **Inode:**
   * An inode (index node) is a data structure that stores metadata about a file.
   * Metadata includes information like file type, permissions, owner, size, timestamps, and pointers to data blocks.
2. **Data Blocks:**
   * The actual data of a file is stored in data blocks.
   * Each file has a series of data blocks that store its content.
3. **Block Addressing:**
   * File systems use block addressing to locate data blocks associated with an inode.
   * The block size is a critical factor in this relationship.
4. **I/O Address:**
   * I/O (Input/Output) addresses are used to access data on storage devices.
   * Translating file-related operations to I/O addresses involves mapping inodes to specific blocks on the storage device.

**Calculating Block and I/O Address of a Given Inode:**

1. **Determine Block Size:**
   * Identify the block size used by the file system (e.g., 4 KB, 8 KB).
   * This information is crucial for calculating block addresses.
2. **Calculate Block Address of Inode:**
   * Each inode points to a set of data blocks.
   * Determine the block number of the inode within the file system.
   * Multiply the block number by the block size to get the offset in bytes from the beginning of the file system.

**Formula:** Block Address of Inode=Block Number of Inode×Block SizeBlock Address of Inode=Block Number of Inode×Block Size

1. **Calculate I/O Address of a Data Block:**
   * Once you have the block address of the inode, determine the block number of the specific data block within the inode.
   * **Multiply** the **block number** by the **block size** to get the offset within the inode.

**Formula:**

I/O Address of Data Block = **Block Address of Inode** + **Block Number within Inode** × **Block Size**

**Example:**

Let's say you have a file system with a block size of 4 KB. If the inode of a file is located at block number 5, and the file's data is in the second block within the inode, you can calculate the I/O address.

1. **Calculate Block Address of Inode:**

Block Address of Inode = 5 × 4 KB = 20 KB

1. **Calculate I/O Address of Data Block:**

I/O Address of Data Block = 20 KB + 2 × 4 KB = 28 KB