APPLICATION-SPECIFIC ALGORITHMS

Parallel Algorithms and Multithreading

Parallel Algorithm Model

This defines how a computation is divided into tasks, assigned to processors, executed simultaneously, and coordinated to solve a problem efficiently.

It exploits the availability of **multiple processors or cores** to execute parts of a program concurrently. The purpose is to reduce computation time and handle larger problems that are impractical for sequential processing.

A *parallel algorithm* is a procedure designed for **parallel execution**, where tasks may interact through **communication** and **synchronization** while maintaining correctness.

Terminologies:

- **Task** – a unit of computation executed independently or with limited dependencies.
- **Processor** – a logical or physical computational unit executing tasks.
- **Communication** – exchange of data among processors.
- **Synchronization** – coordination of execution order to respect dependencies.
- **Granularity** – the amount of computation per communication; coarse-grained means large independent chunks, fine-grained means frequent interaction.

Types of Parallel Models

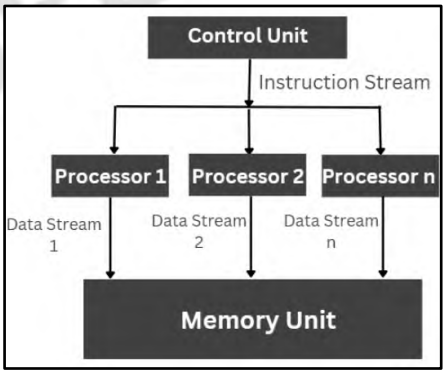
Data-Parallel Model

Divides a large data set into smaller chunks and distributes them across multiple processors. Each processor executes the same instruction or operation on its portion of the data simultaneously. This is the foundation of SIMD (Single Instruction, Multiple Data) architectures and GPU computing.

Mechanics

1. **Data Partitioning:** The dataset (array, list, matrix, etc.) is split evenly among processors. Example: An array of 1,000 elements is divided into four 250-element blocks.
2. **Uniform Operation:** Each processor performs the same function (e.g., addition, multiplication, filtering) on its subset.
3. **Synchronization:** Once each processor finishes its part, the results are combined (e.g., summed, concatenated, or merged).

For example: Dense Matrix Multiplication



In the example, the instruction stream is divided into the available number of processors. Each processor computes the data stream it is allocated and accesses the memory unit for read and write operations. As seen, the data stream 1 is allocated to processor 1, and once it completes the calculation, the result is stored in the memory unit.

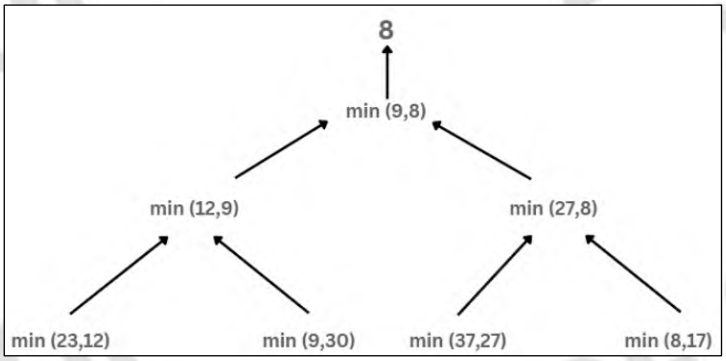
Task Graph Model

Divides computation into a set of distinct tasks that can run concurrently, each performing different operations. Tasks may have data or control dependencies, often represented as a Directed Acyclic Graph (DAG). This model is foundational to workflow systems, compiler optimization, and parallel functional programming.

Mechanics

1. **Task Identification:** Break the problem into smaller subtasks with well-defined input/output.
2. **Dependency Analysis:** Build a DAG showing which tasks depend on which.
3. **Scheduling:** Assign independent tasks to different processors.
4. **Execution and Synchronization:** Processors run independent tasks simultaneously; dependent tasks wait until prerequisites are complete.

For example: Finding the minimum number



In the example, the task graph model works in parallel to find the minimum number in the given stream. As shown, the minimum of 23 and 12 is computed and passed on further by one process; similarly, at the same time, the minimum of 9 and 30 is calculated and passed on to the further process. This approach of computation requires less time and effort.

Work Pool Model

In the work-pool model, all tasks are stored in a **shared pool or queue**. This model dynamically balances workload such that processors that finish early take on new tasks, improving overall efficiency.

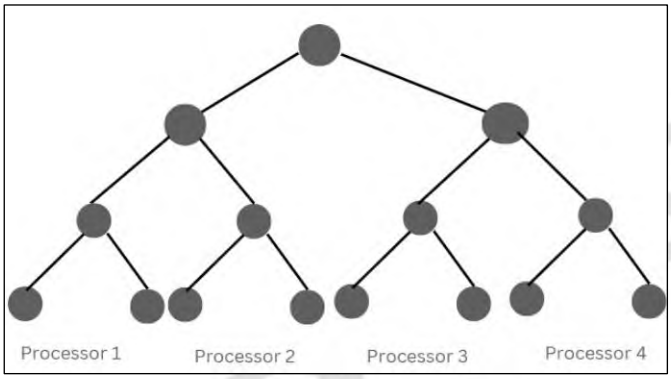
Each processor repeatedly:

1. Fetches a task from the pool.
2. Executes it.
3. Returns results.
4. Takes another task until none remain.

Mechanics

1. A centralized or distributed queue contains pending tasks.
2. Processors “pull” work as they become idle.
3. The pool shrinks as tasks complete, reducing the total workload dynamically.

For example: Parallel Tree Search



In the example, it uses the work pool model for its computation of four (4) processors simultaneously. The four (4) sub-tress are allocated to four (4) processors, and they carry out the search operation.

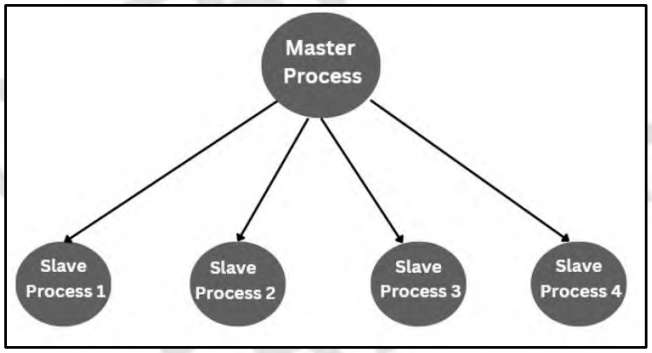
Master-Worker Model

or master–slave, uses a central controller (master) that distributes tasks to several worker processors. Workers perform computation and return results to the master, which coordinates the workflow.

Mechanics

1. **Task Distribution**: The Master assigns different parts of the problem to workers.
2. **Concurrent Execution:** Workers process their tasks in parallel.
3. **Result Aggregation:** Master collects outputs and assembles final results.

For example: Distribution of workload across multiple slave nodes by the master process



As shown, the workload distribution is done across multiple processes. In the diagram, a node is the master process that allocates the workload to the other four (4) slave processes. This way, each sub-computation is carried out by multiple slave processes.

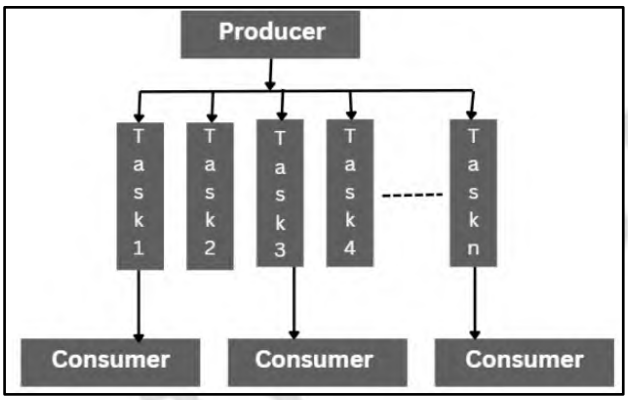
Pipeline Model

Divides a computation into ordered stages, similar to an assembly line. Each stage performs a part of the computation and passes results to the next stage. When the pipeline is full, multiple data items are processed concurrently, one at each stage.

Mechanics

1. **Decomposition:** Split the problem into sequential stages.
2. **Stage Assignment:** Assign each stage to a different processor.
3. **Data Flow:** Each processor processes its stage and passes output to the next.

For example: Parallel LU Factorization Algorithm

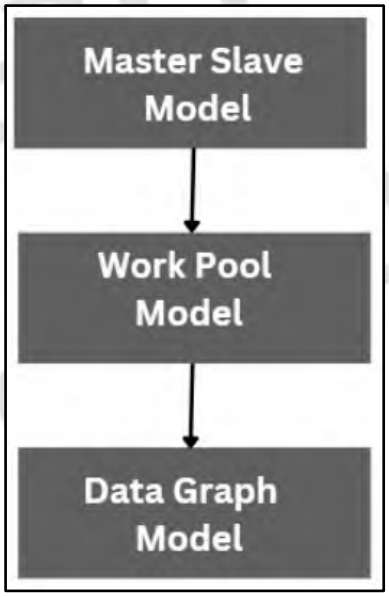


As shown, in this model, the producer reads the input matrix and generates the tasks that are required for computing the LU factorization as an output. The producer divides this input matrix into a smaller size of multiple tasks and shares them in a shared task queue. The consumers then retrieve these blocks and perform the LU factorization on each independent block.

Hybrid Model

Combines features of multiple models to exploit the best of each, depending on the problem phase or architecture. This model reflects modern **heterogeneous computing systems** (multi-core CPUs + GPUs + distributed clusters).

For example: A combination of master-worker, work pool, and data graph model.



As shown, three (3) different models are used at each phase. Consider the example where the master-worker model is used for the data transformation task. The master process distributes the task to multiple slave processes for parallel computation.

In the second phase, the work pool model is used for data analysis, and similarly, the data graph model is used for making the data visualization.

Multithreading

A concurrency technique where a single process runs multiple *threads* that share the same address space. Each thread is an independent sequence of instructions that can execute logically in parallel (on multi-core CPUs) or interleave (on a single core).

In multithreaded programming, managing shared data and ensuring thread safety are critical challenges. To address these challenges, specific data structures are designed or utilized to facilitate safe and efficient concurrent access by multiple threads.

These data structures help prevent issues like race conditions, deadlocks, and data corruption while optimizing performance.

- **Concurrent Queues**: Allows multiple threads to safely enqueue and dequeue items without explicit synchronization. They are essential for producer-consumer scenarios where threads produce data and other threads consume it.

  Use cases include task scheduling in thread pools, message passing between threads, and buffering data between producers and consumers.

  For example, the program below shows how two (2) threads (Producer and Consumer) safely share a blocking queue (`LinkedBlockingQueue`) without interfering with each other, thanks to built-in thread synchronization.

```
import java.util.concurrent.*;

// Short example: Concurrent queue with producer and consumer threads
public class ShortConcurrentQueue {
    public static void main(String[] args) {
        // Thread-safe queue shared by all threads
        BlockingQueue<String> queue = new LinkedBlockingQueue<>();

        // Producer thread: adds items to the queue
        Thread producer = new Thread(() -> {
            try {
                for (int i = 1; i <= 3; i++) {
                    String item = "Task-" + i;
                    queue.put(item); // blocks if full
                    System.out.println("[Producer] Added: " + item);
                    Thread.sleep(300); // simulate work
                }
                queue.put("DONE"); // signal completion
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        // Consumer thread: retrieves items from the queue
        Thread consumer = new Thread(() -> {
            try {
                while (true) {
                    String item = queue.take(); // blocks if empty
                    if (item.equals("DONE")) break; // exit signal
                    System.out.println("    [Consumer] Processed: " + item);
```

```
                    Thread.sleep(500); // simulate work
                }
                System.out.println("    [Consumer] Finished.");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        // Start both threads
        producer.start();
        consumer.start();
    }
}
```

Output:

```
[Producer] Added: Task-1
    [Consumer] Processed: Task-1
[Producer] Added: Task-2
    [Consumer] Processed: Task-2
[Producer] Added: Task-3
    [Consumer] Processed: Task-3
    [Consumer] Finished.
```

- **Thread-Safe Collections**: Handles concurrent access internally, ensuring that multiple threads can modify them without causing inconsistencies or corrupting the data.

  Use cases include shared caches and lookup tables, managing lists of active threads or tasks, and shared resources in web servers and applications.

  For example, the program below demonstrates concurrent access to a shared data structure — specifically, a non-blocking queue (`ConcurrentLinkedQueue`) — by multiple threads.

```
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;

// Short example: Thread-safe collection with multiple threads
public class ThreadSafeCollectionDemo {
    public static void main(String[] args) throws InterruptedException {
        // Create a non-blocking, thread-safe queue
        Queue<String> sharedQueue = new ConcurrentLinkedQueue<>();

        // Producer thread: adds elements to the queue
        Thread producer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                sharedQueue.add("Task-" + i);
                System.out.println("[Producer] Added Task-" + i);
                try { Thread.sleep(200); } catch (InterruptedException ignored)
{}
            }
        });

        // Consumer thread: removes elements from the queue
```

```
        Thread consumer = new Thread(() -> {
          while (true) {
                String item = sharedQueue.poll(); // removes head or returns null if empty
                if (item != null)
                    System.out.println("    [Consumer] Processed " + item);
                else if (Thread.interrupted()) break; // exit when interrupted
                try { Thread.sleep(150); } catch (InterruptedException e) { break; }
            }
        });

        producer.start();
        consumer.start();

        // Wait for producer to finish, then stop consumer
        producer.join();
        Thread.sleep(1000);
        consumer.interrupt();

        System.out.println("All tasks completed safely using a thread-safe
collection.");
    }
}
```

Output:

```
[Producer] Added Task-1
    [Consumer] Processed Task-1
[Producer] Added Task-2
    [Consumer] Processed Task-2
[Producer] Added Task-3
    [Consumer] Processed Task-3
All tasks completed safely using a thread-safe collection.
```

- **Blocking Queues**: Specialized queues that block threads attempting to enqueue or dequeue when the queue is full or empty, respectively. They are useful for coordinating the flow of data between producer and consumer threads.

  Use cases include implementing producer-consumer patterns, task distribution in thread pools, and managing resources in limited-capacity scenarios.

  For example, the program below is a practical, minimal implementation of the *Producer–Consumer* pattern, where one thread produces data and another consumes it, with automatic thread synchronization handled by the `BlockingQueue`.

```
import java.util.concurrent.*;

// Short example: BlockingQueue for thread-safe communication
public class BlockingQueueExample {
    public static void main(String[] args) {
        // Create a blocking queue with a fixed capacity of 3
        BlockingQueue<String> queue = new ArrayBlockingQueue<>(3);

        // Producer thread: adds elements to the queue
        Thread producer = new Thread(() -> {
```

```
            try {
                for (int i = 1; i <= 5; i++) {
                    String task = "Task-" + i;
                    queue.put(task); // waits if the queue is full
                    System.out.println("[Producer] Added " + task);
                    Thread.sleep(400); // simulate work
                }
                queue.put("DONE"); // sentinel value to signal completion
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        // Consumer thread: removes elements from the queue
        Thread consumer = new Thread(() -> {
            try {
                while (true) {
                    String task = queue.take(); // waits if queue is empty
                    if (task.equals("DONE")) break; // exit condition
                    System.out.println("    [Consumer] Processed " + task);
                    Thread.sleep(700); // simulate slower processing
                }
                System.out.println("    [Consumer] Finished.");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        // Start both threads
        producer.start();
        consumer.start();
    }
}
```

Output:

```
[Producer] Added Task-1
[Producer] Added Task-2
    [Consumer] Processed Task-1
[Producer] Added Task-3
[Producer] Added Task-4   <-- waits until consumer frees space
    [Consumer] Processed Task-2
    [Consumer] Processed Task-3
[Producer] Added Task-5
    [Consumer] Processed Task-4
    [Consumer] Processed Task-5
    [Consumer] Finished.
```

- **Lock-Free and Wait-Free Data Structures**: Designed to allow multiple threads to operate on them without using traditional locking mechanisms (like mutexes). They rely on atomic operations to ensure thread safety, which can lead to higher performance and reduced contention.

  Use cases include high-performance applications where minimizing latency and maximizing throughput are crucial, real-time systems requiring predictable execution times, and scenarios with high contention where traditional locks would cause significant bottlenecks.

- **Immutable Data Structures**: Do not allow modification after creation. Instead of changing the data, new versions are created with the desired changes. This immutability inherently makes them thread-safe, as no thread can alter the state once it is created.

  Use cases include functional programming paradigms where immutability is a core principle, shared configuration data that does not change during runtime, and caching and memorization strategies.

- **Synchronized Stacks and Linked Lists**: These can be made thread-safe by incorporating synchronization mechanisms, such as locks or atomic operations, to manage concurrent access.

  Use cases include managing undo/redo operations in applications, implementing thread-safe LIFO(Last-In-First-Out) or FIFO (First-In-First-Out) structures, and handling browser history or navigation stacks.

- **Priority Queues**: Allow elements to be retrieved based on their priority rather than their insertion order. In a multithreaded context, priority queues need to be thread-safe to handle concurrent access.

- **Read-Write Locks with Data Structures**: Allow multiple threads to read a data structure concurrently while ensuring exclusive access for threads that write to it. This improves performance by allowing high concurrency for read operations.

Algorithmic Complexity

Algorithmic Complexity

A measure that evaluates the order of the count of operations performed by a given algorithm as a function of the size of the input data. Simply, complexity is a rough approximation of the number of steps necessary to execute an algorithm.

Algorithm complexity is commonly represented with the **O(f) notation**, also known as **asymptotic notation** or "**Big O notation**", where **f** is the function of the size of the input data. The asymptotic computational complexity **O(f)** measures the order of the consumed resources (CPU time, memory, etc.) by a certain algorithm expressed as a function of the input data size.

Common Time Complexities

Complexity	Growth Description	Example Algorithms/Operations
O(1) — Constant Time	The operation takes the same amount of time regardless of input size.	Accessing an element by index in an array, pushing/popping on a stack, and inserting into a hash table (average case).
O(log n) — Logarithmic Time	Time increases slowly even as n grows large; typically seen in divide-and-conquer algorithms.	Binary search in a sorted array, inserting/searching in a balanced binary search tree (BST), and heap insertion.

Complexity	Growth Description	Example Algorithms/Operations
O(n) — Linear Time	Time increases directly with input size; the algorithm examines each element once.	Linear search, traversing a linked list, and finding min/max in an unsorted array.
O(n log n) — Linearithmic Time	Slightly slower than linear; common in efficient sorting algorithms.	Merge sort, Heap sort, Quick sort (average case), and some divide-and-conquer algorithms.
O(n^2) — Quadratic Time	Time increases with the square of input size; typical in algorithms with nested loops.	Bubble sort, Insertion sort, Selection sort, and comparing all pairs of elements.
O(n^3) — Cubic Time	Three nested loops or triple combinations of data elements.	Matrix multiplication (naïve method), Floyd–Warshall algorithm (graph all-pairs shortest path).
O(2^n) — Exponential Time	Time doubles with each additional input; it grows extremely fast.	Recursive Fibonacci, brute-force subset or permutation generation, traveling salesman problem (brute force).

Common Space Complexities

Complexity	Description	Example
O(1) — Constant Time	Uses a fixed amount of memory regardless of input size.	Iterative variable swap, in-place sorting.
O(log n) — Logarithmic Time	Space grows logarithmically, often due to recursion depth.	Recursive binary search, balanced tree recursion.
O(n) — Linear Time	Memory proportional to input size.	Storing an array or list of size n, a BFS queue.
O(n^2) — Quadratic Time	Space grows quadratically with input size.	2D matrix storage, adjacency matrix representation of a graph.

Typical Complexities in Common Data Structures

Operation	Array	Linked List	Stack/Queue	Binary Search Tree	Hash Tabel
Access	O(1)	O(n)	O(n)	O(log n)	-
Search	O(n)	O(n)	O(n)	O(log n)	O(1)*
Insert	O(n)	O(1)	O(1)	O(log n)	O(1)*
Delete	O(n)	O(1)	O(1)	O(log n)	O(1)*

*Average case; worst case for hash tables is O(n) if collisions occur.

Scheduling Algorithms

Scheduling Algorithms

Methods used by the operating system (OS) to decide which process or thread should run on the CPU and for how long. As there are usually more processes than available CPU cores, the OS needs a fair and efficient way to schedule tasks.

Terminologies

- **Arrival Time:** The time at which the process arrives in the ready queue.
- **Completion Time:** The time at which the process completes its execution.
- **Burst Time:** Time required by a process for CPU execution.
- **Turn Around Time:** Time Difference between completion time and arrival time.

  Turn Around Time = Completion Time – Arrival Time

- **Waiting Time**: Time Difference between turn around time and burst time.

  Waiting Time = Turn Around Time – Burst Time

Types of Scheduling Algorithms

Non-preemptive

A CPU scheduling method where once a process starts execution, it runs to completion (or until it switches to a waiting state for I/O) before another process can use the CPU. The CPU is not forcibly taken away.

It is best for systems where fairness and simplicity are more important than response time (e.g., batch processing, background jobs).

Preemptive

A CPU scheduling method where the OS can interrupt and pause a running process to assign the CPU to another process, often based on priority, time-slice, or shorter CPU bursts.

It is best for systems requiring fast response, fairness, and real-time reactivity (e.g., Windows, Linux, mobile OS, interactive servers).

Common Scheduling Algorithms

- **First-Come, First-Served (FCFS):** A non-preemptive scheduling algorithm where the CPU executes processes in the order they arrive in the ready queue. The first process to request the CPU runs until completion. It is simple to implement but can lead to long waiting times, especially for short processes that arrive after long ones (**convoy effect**).

  This is best for batch processing systems where jobs are long, predictable, and do not require frequent user interaction, such as printing large documents in the order they are submitted.

- **Shortest Job First (SJF):** A non-preemptive scheduling algorithm that selects the process with the shortest CPU burst time next. It minimizes average waiting time and is optimal in theory, but requires knowledge or estimation of burst times and can cause starvation of long processes.

  This is best for offline or batch systems where burst times can be estimated, such as automated payroll processing or jobs where execution times are known in advance.

- **Shortest Remaining Time First (SRTF):** A preemptive version of SJF. The CPU is always assigned to the process with the least remaining execution time. If a new process arrives with a shorter burst time than the currently running process, it preempts the CPU. It improves response time but may also cause starvation.

  This is best for real-time systems requiring rapid response, such as emergency alert systems or time-critical sensor input processing, where shorter tasks must always be prioritized.

- **Priority Scheduling:** A scheduling method (either preemptive or non-preemptive) that assigns the CPU to the process with the highest priority value. Priorities may be based on system needs, user levels, or resource requirements. However, low-priority processes may starve, so **aging** is often used to prevent this.

  This is best for mission-critical systems, such as medical software, flight systems, or antivirus scans, where some tasks must run before all others, regardless of arrival time.

- **Round Robin:** A preemptive, time-sharing scheduling algorithm where each process is given a fixed time slice (called a **time quantum**) in cyclic order. If a process doesn't finish before its quantum expires, it is preempted and placed at the end of the ready queue. This ensures fairness and is ideal for interactive systems.

  This is best for interactive and multi-user environments, such as modern operating systems (Windows, Linux) that run many user applications and must remain responsive to keyboard/mouse inputs.

- **Multilevel Queue (MLQ):** A scheduling method that divides processes into **multiple queues**, each with its own scheduling policy (e.g., RR for foreground jobs and FCFS for background jobs). Processes are permanently assigned to a queue based on characteristics such as type, priority, or memory needs. Higher-level queues have priority over lower ones.

  This is best for systems that categorize users or workloads, such as separating system processes, interactive processes, and background jobs in embedded or server systems.

- **Multilevel Feedback Queue (MLFQ):** An advanced scheduling method that allows processes to move between queues based on behavior and CPU usage. New or short jobs get priority, while longer jobs are gradually moved to lower-priority queues. It reduces starvation, adapts to different workloads, and is used in many modern operating systems.

  This is best for general-purpose, modern OS scheduling as it adapts to different workloads and improves both response time and throughput. Used in UNIX/Linux variants.

Application Scenario Example

A personal computer is running multiple tasks at the same time. A **System Update (P1)** starts first. After 1 millisecond, the **Antivirus Scan (P2)** starts in the background. At time 2, the user opens the **Browser (P3)**, which is considered a high-priority interactive process. At time 3, a **Backup Task (P4)** begins. Finally, at time 4, an **Email Notification (P5)** triggers. The OS must schedule these processes using different CPU scheduling algorithms.

Process (Job)	Arrival Time	Burst Time	Priority *(1 = Highest)*
P1 – System Update	0	8	2
P2 – Antivirus Scan	1	4	4
P3 – User App (Browser)	2	2	1
P4 – Background Backup	3	6	5
P5 – Email Notification	4	3	3

Output for each Scheduling Algorithm:

Scheduling Algorithm	How CPU Will Behave in This Scenario
FCFS *(Non-Preemptive)*	P1 → P2 → P3 → P4 → P5 (strictly by arrival time; browser waits too long → poor responsiveness)
SJF *(Non-Preemptive)*	CPU always picks shortest burst next (P3 first when it arrives, then P2, P5, etc., not optimal for interactive fairness)
SRTF *(Preemptive SJF)*	P3 interrupts others when it arrives because it has the shortest remaining time; improves response time
Priority Scheduling *(Preemptive)*	P3 runs first (highest priority), then P1, then P5, then P2, then P4 (risk of starvation for low-priority jobs)
Round Robin (q = 3) *(Preemptive)*	All jobs get equal time slices (keeps system responsive; good for multitasking)
Multilevel Queue *(MLQ)*	Foreground jobs (P3, P5) handled with RR, while background jobs (P1, P2, P4) handled with FCFS (rigid separation)
Multilevel Feedback Queue (MLFQ) *(Preemptive)*	P3 gets highest priority for responsiveness; long jobs drop to lower queues; most "real OS-like" behavior

References:

Geeks for Geeks. (2025). *Parallel algorithm models in parallel computing.* Retrieved on October 17, 2025, from https://www.geeksforgeeks.org/mobile-computing/parallel-algorithm-models-in-parallel-computing/

Oracle. (2025). *Understanding basic multithreading concepts..* Retrieved on October 17, 2025, from https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032e/index.html

Redwood Software. (2024). *Job scheduling algorithms: Which is best for your workflow?.* Retrieved on October 20, 2025, from https://www.redwood.com/article/job-scheduling-algorithms/