Processor Management

Madiba Hudson-Quansah

Contents

Chapter 1

Processor Management

1.1 Definitions

Definition 1.1.1: Processor

The part of the hardware that performs calculations and executes programs.

Definition 1.1.2: Program

An inactive unit, stored on disk. A work unit submitted by a user.

Definition 1.1.3: Process / Task

A program that has been given CPU time to execute. A program loaded into memory with a current activity. A process requires requires a set of resources, including a processor and special registers to perform its function.

Definition 1.1.4: Thread

A portion of a process created by a process that can be scheduled for execution independently of it's parent process. A process can consist of multiple threads where the parent process can own allocated resources, and becomes passive as its threads utilize these resources, including CPU time.

Definition 1.1.5: Multithreading

Allows applications to manager a separate process with several threads of control.

Definition 1.1.6: Multiprogramming

Requires the processor be allocated to each job or to each process for a period of time and deallocated at an appropriate moment.

Definition 1.1.7: Interrupt

A signal to the processor emitted by hardware or software indicating an event that needs immediate attention. Facilitates the switching of the CPU from one process or job to another.

Definition 1.1.8: Context Switch

Saving job processing information when an interrupt occurs, so that the job can be continued correctly at a later time. This is done via the PCB saving the current state of the job, including the program counter and register contents.

1.2 Scheduling Submanagers

- The process manager is composed of at least two submanagers:

  Job Scheduler - A higher-level scheduler that handles job scheduling, and job initiation based on certain criteria.

  Process Scheduler - A lower-level scheduler that handles process scheduling, determining execution steps, based on certain criteria.

Jobs can be views either as a series of global job steps, i.e. compilation, loading and execution, or as one monolithic step, like execution. The scheduling of jobs is handled on two levels by most operating systems in a hierarchical manner between the Job Scheduler and the Process Scheduler.

1.2.1 Job Scheduler

- Selects incoming jobs from the job queue.
- Places them into the process queue
- Decides on job initiation criteria based on process scheduling algorithm and priority.
- The goals of the job scheduler include:
  - Sequence jobs to ensure efficient system resource utilization.
  - Balance I/O interaction and computation.
  - Keep most system components busy.

Scheduling jobs are based on priority, with each job initiated by the job scheduler based on certain criteria and characteristics. The Job Scheduler's goal is to put the jobs in a sequence that best meets the designers or administrator's goals, for example efficient resource utilization. Once a job is selected for execution, the Process Scheduler determines when each step, or set of steps of the job is executed, also based on certain criteria.

1.3 Process Scheduler

- Determines which job gets CPU time, when and for how long.
- Decides when to interrupt a running process.
- Determines queues for job movement during execution.
- Recognizes job conclusion and determines job termination.
- Assigns CPU time to individual jobs placed in the ready queue by the job scheduler.
- The Process Scheduler exploits common computer program traits
  - CPU and I/O cycles - Programs alternate between CPU and I/O cycles.
  - Frequency and CPU cycle durations - Most programs spend more time in I/O cycles than in CPU cycles.
  - I/O bound and CPU bound programs - I/O bound programs spend more time in I/O cycles, while CPU bound programs spend more time in CPU cycles.

After a job a has been accepted by the Job Scheduler it is placed in the ready queue, where the Process Scheduler takes over. The Process Scheduler determines which processes will get the CPU, when and for how long. The Process Scheduler also determines what to do when processing is interrupted, which waiting lines (queues) the job should be allocated to during its execution, and when the job is completed.

The Process Scheduler is a low-level scheduler that assigns the CPU time to execute instructions for those jobs placed on the ready queue by the Job Scheduler, this orchestration allows for context switching between jobs, and the illusion of simultaneous execution. In scheduling the CPU, the Process Scheduler takes into account common traits of computer programs, mostly how they alternate between different cycles of CPU and I/O activity. Although the duration and frequency of CPU cycles vary between programs there are general trends most programs exhibit, i.e.:

- CPU Bound Jobs / Programs - Utilize the CPU more frequently than I/O devices, thus spending more time in CPU cycles. Examples include scientific calculations and graphics processing.
- I/O Bound Jobs / Programs - Utilize I/O devices more frequently than the CPU, thus spending more time in I/O cycles. Examples include word processors and database management systems.

In very interactive systems there is a third layer of the Processor Manager called the Middle-Level Scheduler, which handles active job removal in the case of system overload. This allows other jobs to be completed faster as it removes less important jobs from the system temporarily allowing more important jobs to be completed faster. These removed jobs are reintroduced into the system when the load decreases.

1.3.1 Job and Process States

- HOLD
- READY
- WAITING
- RUNNING
- FINISHED

As a unit of execution (job, process, thread) moves through the system it changes status, often from HOLD, to READY, to RUNNING, to WAITING, and eventually, to FINISHED. These are called job status, process status, or thread status depending on the unit of execution being described. The following are the common states a job, process or thread can be in as show by the state diagram.

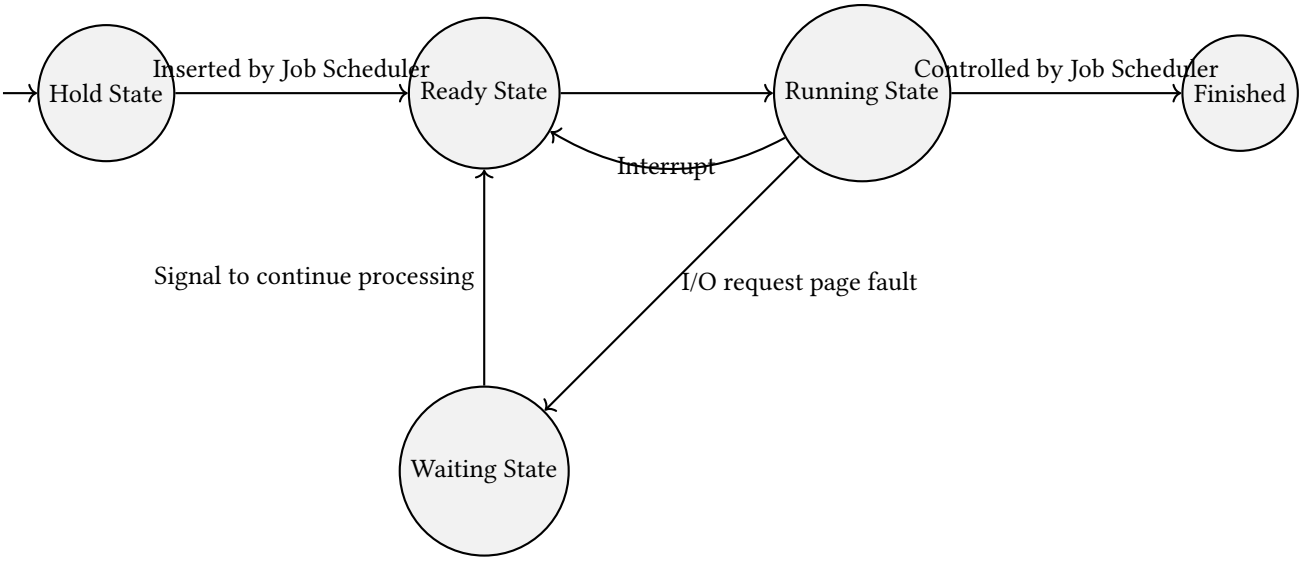

Figure 1.1: Job / Process State Diagram

When a job is accepted by the system it goes through the following states:

The job is put on HOLD and placed in a queue. In some systems the job spooler / disk controller creates a table with the characteristics of each job in the queue, and notes the important features of each job such as estimated CPU time, priority,

special I/O devices, required and maximum memory required. This table is used by the Job Scheduler to decide which job to run next.

Then the job moves to the READY state after the interrupts have been resolved or placed on the READY queue directly in some systems. The process then moves to RUNNING when the Process Scheduler allocates CPU time to it. WAITING means the process can't continue until a specific resource is allocated or an I/O operation as finished, and then it can move back to READY. When the process is completed it moves to the FINISHED state and is removed from the system.

Transitions from job statuses (HOLD → READY, RUNNING → FINISHED) are controlled by the Job Scheduler, while transitions between process / thread statuses, i.e. all other transitions, are controlled by the Process Scheduler.

1.3.2 Thread States

- READY
- RUNNING
- WAITING
- DELAYED
- BLOCKED

Threads move through five states as they are processed by the system, as show n in the state diagram below. When a process creates a thread it is made ready by allocating to it the needed resources and placing it in the READY queue. The thread state changes from READY to RUNNING when the Process Scheduler assigns it to a processor.

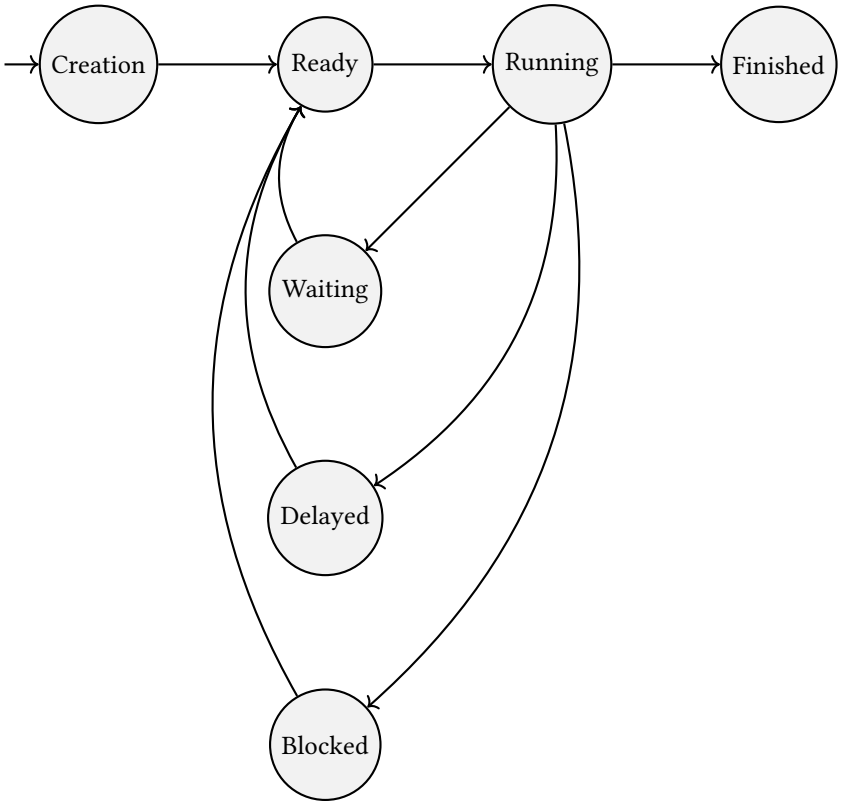

Figure 1.2: Thread State Diagram

A thread can move from RUNNING to WAITING when it has to wait for an event outside it's control to occur, like, an I/O operation or an input from the user, then back to READY when the event has occurred. Alternatively another thread that is finished running can send a signal indicating that the waiting thread can continue processing.

When an application has the ability to delay the processing of a thread for some time, it causes the thread to move from RUNNING to DELAYED. When he delay time has passed the thread moves from DELAYED back to READY, for example the saving of a document every five minutes.

A thread moves from RUNNING to BLOCKED when an I/O request is issued. After the I/O operation is done, the thread returns to the READY state. When a thread is finished it moved to the FINISHED state and all the allocated resources are deallocated.

1.3.3 Control Blocks

Definition 1.3.1: Process Control Block (PCB)

Contains information about each process in a system, including:

- Process Identifier (PID) - Unique Identifier assigned by the Job Scheduler
- Process Status - The current status of the job, i.e. READY, RUNNING, WAITING
- Process state - Contains all the information needed to indicate the current state of the job
  - Process Status Word - The current instruction counter and register contents when the job isn't running but is either on HOLD, READY or WAITING.
  - Register Contents - The contents of the register if the job has been interrupted and is waiting to resume processing.
  - Main Memory - The address where the job is stored, and in the case of virtual memory, the linking between the virtual and physical memory locations.
  - Resources - Information about all the resources allocated to this job, with each resources having an identification field listing its type, details of its allocation such as the sector address on a disk.
  - Process Priority - Used by priority scheduling algorithms to select which job to run next.
- Accounting Information - Information used for billing purposes and performance measurement.

Definition 1.3.2: Thread Control Block (TCB)

Contains information about each thread in a system, including:

- Thread Identifier (TID) - Unique Identifier assigned by the Process Scheduler when the thread is created.
- Thread State - The current state of the thread, i.e. READY,
- CPU Information - Everything the operating system needs to know.
  - Program Counter - How far the thread has progressed in its execution, and which instruction is currently being executed.
  - Register Contents - The contents of the registers when the thread is not running.
- Thread Priority - Used to indicate the weight of a thread relative to other thread. Determines which thread should be selected from the READY queue next.
- Pointer to process that created the thread - Indicates the process that created the thread.
- Pointers to all other threads created by this thread - Indicates child threads created by this thread.

Each process in the system is represented by a Process Control Block(PCB), and each thread is represented by a Thread Control Block(TCB). These control blocks contain necessary management information about the processes and threads, allowing the operating system to manage them effectively.

The PCB is created when the Job Scheduler accepts the job and its updated as the job progresses from the beginning to the end of the its execution, while the TCB is created when a thread is created by a process and updated as the thread progresses through its states.

Queues use control blocks to track jobs. As the job moves through the system its progress is noted in the control block. This allows for interrupted processes and threads to be resumed after a delay. The control block is linked to other control blocks to form queues, for example the READY queue.

1.4 Scheduling Policies and Algorithms

Definition 1.4.1: Turnaround Time

The total time taken between the submission of a process and its completion. It includes the time spent waiting to get into memory, waiting in the ready queue, executing on the CPU and doing I/O operations.

Definition 1.4.2: Natural Wait

The time a process spends waiting for I/O operations to complete.

Definition 1.4.3: Pre-emptive Scheduling

A scheduling strategy that interrupts the processing of a job and transfers the CPU to another job.

Definition 1.4.4: Non-Pre-emptive Scheduling

A scheduling strategy that allows a job to run until it completes or switches to the WAITING state.

In a multiprogramming environment, the operating system must decide how to resolve three limitations of the system before scheduling processes effectively:

- There are a finite number of resources
- Some resources are non-shareable once allocated.
- Some resources require operator intervention to allocate or deallocate.

A good scheduling policy should aim to achieve the following goals, even though some goals may contradict each other:

Maximize Throughput - Run as many jobs as possible in a given amount of time.

Minimize Response Time - Quickly turn around interactive requests.

Minimize Turnaround time - Move entire jobs in and out of the system quickly.

Minimize Waiting Time - Move jobs out of the READY queue as quickly as possible.

Maximize CPU efficiency - Keep the CPU busy 100% of the time.

Ensure fairness for all jobs - Give every job an equal amount of CPU and I/O time.

1.5 Scheduling Algorithms

Definition 1.5.1: Scheduling Algorithm

A set of rules that the operating system uses to allocate the CPU in the best way for moving jobs through the system efficiency.

1.5.1 First-Come, First-Served (FCFS)

A nonpreemtive scheduling algorithm that handles all incoming objects according to their arrival time: The earlier they arrive the sooner they're processed. This algorithm is commonly used for batch systems, and is unacceptable for many interactive systems because interactive users expect quick response time.

With FCFS as a new job enters the system, its PCB is placed at the end of the READY queue, and when the CPU becomes available the job at the front of the READY queue is selected for processing. In a strict FCFS system there is no WAITING state because each job is run to its completion, although there are many systems allowing context switches on natural wait requests, with execution resuming when the I/O operation is completed. Assuming there's no multiprogramming,

turnaround time is unpredictable with a strict FCFS policy, as each jobs runs to completion so all other jobs must wait until the job is done.

Example 1.5.1

Given the following jobs arrive in the order listed:

- Job A - 15 ms CPU time
- Job B - 2 ms CPU time
- Job C - 1 ms CPU time

If all three jobs arrive at time 0, the turnaround time (finish time − arrival time) for each job is:

$$\text{Job A} = 15 - 0 = 15$$
$$\text{Job B} = 17 - 0 = 17$$
$$\text{Job C} = 18 - 0 = 18$$

The average turnaround time is:

$$\frac{15 + 17 + 18}{3} = 16.67$$

However if the jobs arrived in the order C, B, A, the turnaround times, are:

$$\text{Job C} = 1 - 0 = 1$$
$$\text{Job B} = 3 - 0 = 3$$
$$\text{Job A} = 18 - 0 = 18$$

The average turnaround time is:

$$\frac{1 + 3 + 18}{3} = 7.34$$

Due to the unpredictability of turnaround times, the average turnaround times are rarely minimized. If one job monopolizes the system the effect on system performance depends on the type of job, i.e. CPU bound or I/O bound. With a CPU bound job using the CPU, the other jobs in the system that are waiting for processing or finishing I/O requests have to wait longer to for CPU time. If I/O requests are not being serviced the I/O queue remain stable while the READY queue grows with new arrivals, with extreme cases having the READY queue filled to capacity while the I/O queue is empty.

On the other hand if an I/O bound job is using the CPU, the other jobs in the system that are waiting for processing or finishing I/O requests have to wait less time for CPU time. Since I/O bound jobs frequently issue I/O requests, the I/O queue grows while the READY queue shrinks as jobs complete their I/O requests and return to the READY queue, leaving the CPU idle waiting for the next job to arrive.

1.5.1.1 Key Points of FCFS

- Simple and easy to implement.
- Non pre-emptive
- Mostly used in batch systems.
- FIFO queue.
- Turnaround time is unpredictable and not minimized.

1.5.2 Shortest Job Next (SJN) / Shortest Job First (SJF)

A nonpreemtive scheduling algorithm that chooses jobs based on the length of their CPU cycle time. This algorithm works best in an environment where the estimated CPU time required to run the job is given in advance by each user at the start of the job, i.e. a batch system. In interactive systems this information is not usually available so SJN is not commonly used.

Example 1.5.2

Given the following jobs all arriving at time 0:

- Job A - 5 CPU cycles
- Job B - 2 CPU cycles
- Job C - 6 CPU cycles
- Job D - 4 CPU cycles

The SJN algorithm would schedule the jobs with the shortest CPU cycles first, resulting in this order: B, D, A, C. This gives the following turnaround times:

$$\text{Job B} = 2 - 0 = 2$$
$$\text{Job D} = 6 - 0 = 6$$
$$\text{Job A} = 11 - 0 = 11$$
$$\text{Job C} = 17 - 0 = 17$$

Resulting in an average turnaround time of:

$$\frac{2 + 6 + 11 + 17}{4} = 9$$

The SJN algorithm consistently minimizes the average turnaround time because the time for the first job has the biggest impact on the average turnaround time as it affects the waiting time of all other jobs in the READY queue. By executing the shortest jobs first, SJN reduces the waiting time for all other jobs, leading to a lower average turnaround time overall.

However SJN is only optimal if all jobs arrive at the same time, and the CPU can accurately predict the CPU time required for each job.

1.5.2.1 Key Points of SJN

- Simple and easy to implement.
- Non pre-emptive
- Mostly used in batch systems.
- Requires knowledge of CPU time required for each job.
- Minimizes average turnaround time.
- Only optimal if all jobs arrive at the same time and accurate CPU time predictions are available.

1.5.3 Priority Scheduling

Nonpreemtive algorithm that gives preference to important jobs based on some criteria. Important jobs aren't interrupted until their CPU cycles are completed or a natural wait occurs. If two or more jos with equal priority are present in the READY queue, the processor is allocated using FCFS.

Priorities can be assigned by a system administrator using extrinsic job characteristics. With a priority algorithm job PCBs are usually placed at the end of one of several READY queues by the Job Scheduler based on their priority so that the Process Scheduler manages multiple READY queues. Some intrinsic characteristics that can be used to assign priorities include:

Memory Requirements - Jobs requiring large amounts of memory could have lower priorities, or vice versa.

Number and type of I/O devices - Jobs requiring many I/O devices could be given lower priorities, than those requiring few I/O devices.

Total CPU time - Jobs having a long CPU cycle or estimated run time could be given lower priorities, than those with short CPU cycles.

Amount of time already spent in the system - Some systems increase the priority of jobs that have been in the system for an unusually long time to prevent starvation. This is called **ageing**.

1.5.3.1 Key Points of Priority Scheduling

- Non pre-emptive
- Can be used in both batch and interactive systems.
- Requires knowledge of job characteristics to assign priorities.
- Can lead to starvation of low-priority jobs.

1.5.4 Shortest Remaining Time (SRT)

A preemptive version of the SJN algorithm, that allocates jobs closest to completion, allowing new jobs with shorter remaining times to interrupt running jobs. When using SRT if several jobs have the same amount of time remaining the algorithm falls back to FCFS.

This algorithm cannot be implemented in an interactive system because it requires knowledge of the exact CPU time required to finish each job, which is only available in batch systems. SRT involves more overhead than SJN in the form of context switching and monitoring of remaining times.

Example 1.5.3

Given the following jobs:

- Job A - 6 ms CPU time, arrives at time 0
- Job B - 3 ms CPU time, arrives at time 1
- Job C - 1 ms CPU time, arrives at time 2
- Job D - 4 ms CPU time, arrives at time 3

The finish times for each job using SRT are:

- Job A - Starts at time 0, interrupted at time 1, resumes at time 9, finishes at time 14
- Job B - Starts at time 1, interrupted at time 2, resumes at time 3, finishes at time 5
- Job C - Starts at time 2, finishes at time 3
- Job D - Starts at time 5, finishes at time 9

For each job the turnaround time is:

$$\text{Job A} = 14 - 0 = 14$$
$$\text{Job B} = 5 - 1 = 4$$
$$\text{Job C} = 3 - 2 = 1$$
$$\text{Job D} = 9 - 3 = 6$$

Resulting in an average turnaround time of:

$$\frac{14 + 4 + 1 + 6}{4} = 6.25$$

SRT is often affected by the time needed for context switching, especially if many jobs with short remaining times arrive frequently, causing numerous interruptions. How context switching is done depends on the CPU architecture, which

the overhead depending on the way the CPU handles interrupts. Although SRT appears to be faster, in a real operating environment its advantages may be negated by the time spent context switching.

1.5.4.1 Key Points of SRT

- Pre-emptive
- Mostly used in batch systems.
- Requires knowledge of CPU time required for each job.
- Minimizes average turnaround time.
- Affected by context switching overhead.

1.5.5 Round Robin (RR)

A preemptive scheduling algorithm, used in interactive systems, that allocates CPU time to each job in a FCFS manner for a fixed time period called a **time quantum** or **time slice**.

Jobs are placed in the READY queue using a FIFO structure, with the job at the front of the queue being allocated the CPU for a time quantum. If processing isn't finished when the time quantum expires, the job is interrupted and put at the end of the READY queue, with it's information saved in its PCB. The next job in the READY queue is then allocated the CPU for a time quantum, and this process continues until all jobs are completed.

If the job's CPU cycle is smaller than the time quantum one of two actions can take place:

- If this is the job's last CPU cycle and the job is finished, then all resources allocated to it are released and the completed job is returned to the user.
- If the CPU has been interrupted by an I/O request, then the information about the job is saved in its PCB and the job is placed in the WAITING queue. When the I/O operation is completed the job is placed at the end of the READY queue.

Example 1.5.4

Given the following jobs:

- Job A - 8 ms CPU time, arrives at time 0
- Job B - 4 ms CPU time, arrives at time 1
- Job C - 9 ms CPU time, arrives at time 2
- Job D - 5 ms CPU time, arrives at time

With a time quantum of 4 ms, the finish times for each job using RR are:

- Job A - Starts at time 0, interrupted at 4, resumes at 16 and finishes at time 20
- Job B - Starts at time 4, interrupted at time 8, finishes at time 8
- Job C - Starts at time 8, interrupted at time 12, resumes at time 20, and is interrupted at time 24, resumes at time 25 and finishes at time 26
- Job D - Starts at time 12, interrupted at time 16, resumes at time 24 and finishes at time 25

For each job the turnaround time is:

$$\text{Job A} = 20 - 0 = 20$$
$$\text{Job B} = 8 - 1 = 7$$
$$\text{Job C} = 26 - 2 = 24$$
$$\text{Job D} = 25 - 3 = 22$$

The length of the time quantum is critical to the performance of the RR algorithm. If the system is a time-critical system requiring low response times, then the time quantum should be small allowing jobs to be processed quickly. If it's an archival system, turnaround time and overhead become critically important, so a larger time quantum is preferred.

The general rules for selecting a time quantum are:

1. It should be long enough to allow 80% of the CPU cycles to run to completion
2. It should be at least 100 times longer than the time required to perform one context switch.

1.5.5.1 Key Points of RR

- Pre-emptive
- Mostly used in interactive systems.
- Time quantum critical to performance.
- Provides good response time.
- Can lead to high turnaround time and overhead if time quantum is too small.

1.5.6 Multiple-Level Queues (MLQ)

Multiple-Level queues isn't a specific scheduling algorithm, but works in conjunction with several schemes to handle jobs that can be grouped according to one or more common characteristics. An example is the priority scheduling algorithm where jobs are grouped according to their priority levels.

Another approach is to group jobs according to their processing characteristics, i.e. CPU bound or I/O bound, with the Process Scheduler alternating between jobs from each group to keep the system balanced ensuring that neither CPU nor I/O devices are idle for long periods.

Another example would be a preemptive approach in a hybrid environment where batch jobs are put into one queue, the background queue, and interactive jobs are put into another queue, the foreground queue. The jobs in the foreground queue are given higher priority, focusing on faster turnaround, than those in the background queue.

There are four primary methods for determining movement:

- No Movement Between Queues
- Movement Between Queues
- Variable Time Quantum Per Queue
- Ageing

1.5.6.1 No Movement Between Queues

A simple policy that works best with high priority jobs. The processor is allocated to the jobs in the highest priority queue in a FCFS manner until the queue is empty, then the next highest priority queue is processed, and so on. This policy works best when there are relatively free users with hight priority jobs so that the top queues quickly empty out, allowing the processor to spend most of its time processing lower priority jobs.

1.5.6.2 Movement Between Queues

A policy that adjusts the priorities assigned to each job. High priority jobs are treated like all others once they enter the system, allowing them to move between queues based on their behaviour. When a time quantum interrupt occurs the job is moved to the end of the next lower priority queue, and so on. A job can also have its priority increased in cases when it issues an I/O request before its time quantum expires, moving it to the end of the next higher priority queue.

This policy works best in a system where jobs are handled according to their computing cycle characteristics, i.e. CPU bound or I/O bound. This assumes that a job that exceeds its time quantum is CPU bound and will require more CPU time than one that requests I/O before the time quantum expires. There fore the CPU-bound jobs are placed at the end of the next lower-level queue when they're interrupted because of the time quantum expiring, while I/O bound jobs are returned to the end of the next higher-level queue once their I/O request has finished. This facilitates I/O-bound jobs and works well for interactive systems.

1.5.6.3 Variable Time Quantum Per Queue

A variant of the movement between queues policy, allowing faster turnaround for CPU-bound jobs. Each queue is given a time quantum twice as long as the one before it. For example if the highest priority queue has a time quantum of 100 ms, the next lower priority queue has a time quantum of 200ms, and so on.

If a job doesn't finish its CPU cycle using the first time quantum, it is moved to the end of the next lower-level queue, and when the processor is next allocated to it the job is allowed to execute for longer and longer periods of time, thus improving its chances of finishing in a reasonable amount of time.

1.5.6.4 Ageing

Definition 1.5.2: Starvation

A situation where a low-priority job is perpetually denied access to the CPU because higher-priority jobs keep arriving.

Used to ensure that jobs in the lower-level queues will eventually complete their execution. The operating system keeps track of each job's waiting time, and when a job gets too old, i.e. when it has waited in the queues for so long that it has reached a certain time limit, the system moves the job into the next highest queue and so on, until it reaches the highest priority queue. Ageing prevents indefinite postponement of low-priority jobs, ensuring that all jobs eventually get processed. Eventually the situation could lead to the old job's starvation, causing it to never be processed. This poses a significant threat to a system if the job is allocated critical resources.

1.5.7 Earliest Deadline First (EDF)

A Dynamic preemptive priority algorithm used to address the critical processing requirements of real-time systems and their deadlines. With EDF the priority of a job can be adjusted as the job moves through execution from START to FINISHED.

The goal of EDF is to process all jobs in the order that is most likely to allow each to run to completion before reaching their respective deadlines. Initially jobs are assigned priorities based on the amount of time remaining until the job's impending deadline, i.e. inversely proportional to its absolute deadline. Therefore the closer the deadline, the higher the priority. As new jobs arrive or existing jobs complete, the priorities are recalculated to ensure that the job with the nearest deadline is always given precedence, and if two or more jobs have the same deadline, the job that arrived first is given priority.

Example 1.5.5

Given the following jobs:

- Job A - 3 ms CPU time, arrives at time 0, deadline at 6
- Job B - 1 ms CPU time, arrives at time 2, deadline at 2
- Job C - 5 ms CPU time, arrives at time 2, deadline at j

1.6 Managing Interrupts

Definition 1.6.1: Interrupt Handler

A control program that handles the interruption sequence of events when an interrupt occurs.

Interrupts can be sent by many events including:

- Page faults
- I/O requests
- Timer expirations
- Illegal arithmetic operations

- Illegal job instructions

Interrupts are handled by the interrupt handler, when operation system detect an error that is not recoverable, the interrupt handler typically follows this sequence:

1. The type of interrupt is described and stored to be passed on to the user as an error message.
2. The state of the interrupted process is saved, including the value of the program counter, the mode specification, and the contents of all registers.
3. The interrupt is processed:
    - The error message and the state of the of the interrupted process are sent to the user
    - The program execution is halted
    - Any resources allocated the job are released
    - And the job exits the system.
4. The process resumes normal operation.

However if he interrupt is nonrecoverable the job is terminated in Step 3.

Chapter 2

Process Synchronization

2.1 Consequences of Poor Synchronization

Definition 2.1.1: Deadlock

A system-wide tangle of resource requests that begins when two or more jobs are put on hold, each waiting for a vital resource to be become available, but if the resources needed by those jobs are the resources held by the other jobs in the deadlock, none of the jobs can proceed. The deadlock is complete if the remainder of the system comes to a halt too.

Fixing deadlocks is critical to maintaining system stability and performance. Deadlocks can lead to several negative consequences, including:

Starvation - Low-priority jobs may never get access to resources if higher-priority jobs keep arriving.

Reduced System Throughput - Deadlocks can lead to a significant reduction in system throughput as resources are tied up in deadlocked processes, preventing other processes from executing.

Increased Response Time - Deadlocks can lead to increased response times for processes as they wait for resources to become available.

2.2 Examples of Deadlocks

A deadlock can occur when a nonshareable and nonpreemtive resources, such as printers or scanners, are allocated to jobs that eventually require other nonshareable, nonpreemtive resources that have been allocated to other jobs.

2.2.1 Deadlocks on File Requests

If jobs are allowed to request and hold files for the duration of their execution a deadlock can occur. In this case each job holds one file and requests another file held by another job. For example Job A holds File 1 and requests File 2, while Job B holds File 2 and requests File 1. Neither job can proceed because each is waiting for a file held by the other job, resulting in a deadlock.

2.2.2 Deadlocks in Databases

A deadlock can also occur if two process access and lock records in a database. For example Process A locks Record 1 and requests Record 2, while Process B locks Record 2 and requests Record 1. Neither process can proceed because each is waiting for a record locked by the other process, resulting in a deadlock.

2.2.3 Deadlocks in Dedicated Device Allocation

2.2.4 Deadlocks in Multiple Device Allocation

2.2.5 Deadlocks in Spooling

2.2.6 Livelock while Disk Sharing

Definition 2.2.1: Livelock

A situation where two or more processes continuously change their state in response to changes in the other processes without making any progress.

2.3 Necessary Conditions of Deadlocks

In each example of deadlock/livelock, there was some interaction of several processes and resources but each time, it was preceded by the simultaneous occurrence of four conditions that the operating system could have recognized:

Mutual Exclusion - At least one resource must be held in a nonshareable mode, meaning that only one process can use the resource at any given time.

Resource Holding - A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

No Preemption - Resources cannot be forcibly removed from processes holding them until the resources are used to completion.

Circular Wait - A set of processes must exist such that each process is waiting for a resource that is held by the next process in the chain, forming a circular chain of processes where each process is waiting for a resource held by the next process.

All four conditions must hold simultaneously for a deadlock to occur, and as long as these four conditions persist, the deadlock will persist. So if one condition is removed the deadlock will be resolved, in fact if the four conditions can be prevented from ever occurring simultaneously then deadlocks can be completely avoided.

2.4 Strategies for Handling Deadlocks

- Prevention - Prevent occurrence of one condition.
  - Mutual Exclusion - Some resources must allocate exclusively. Bypassed if I/O device uses spooling.
  - Resource Holding
    - Bypassed if jobs request every necessary resource at creation time.
    - Multiprogramming degree significantly decreased
    - Idle peripheral devices
- Avoidance - Avoid deadlock if it becomes probable.
- Detection - Detect deadlock when it occurs.
- Recovery - Resume in a graceful manner.

Generally operating systems use a combination of several strategies to deal with deadlocks.

Prevention - Preventing deadlocks by ensuring that at least one of the necessary conditions cannot hold.

Avoidance - Avoiding deadlocks by careful resource allocation based on knowledge of future resource requests.

Detection - Detecting deadlocks when they occur using algorithms that periodically check the system for deadlocks.

Recovery - Recovering from deadlocks by terminating or rolling back one or more processes involved in the deadlock.

2.4.1 Prevention

To prevent a deadlock the operating system must eliminate one of the four necessary conditions for deadlock, further complicated by the fact that the same conditions can't be eliminated from every resource.

Definition 2.4.1: Spooling

A process where data is temporarily gathered and stored to be accessed and processed by a device, program or the user at a later time.

Mutual exclusion is necessary in any computer system because some resources, such as memory, CPU, and dedicated drivers, must be exclusively allocated to one user at a time. In the case of I/O devices such, the mutual exclusion may be bypassed by spooling which allows the output from many jobs to be stored in separate temporary spool files tat the same time, and each complete output file is then selected for use when the I/O device becomes available. However this solution may trade one problem (Deadlock in Device Allocation) for another (Deadlocks in Spooling).

Resources holding can be resolved by forcing each job to request every resource it will need to run to completion at the time of its creation. This would significantly decrease the degree of multiprogramming in the system because many resources would be tied up waiting for jobs to complete, and I/O devices would be idle much of the time.

No preemption could be bypassed by allowing the operating system to deallocate resources from jobs, using context switching to save the current state of the job. This approach would also require complex recovery tasks in the cause of the preemption of a dedicated I/O resource, or files modified by the job.

Circular wait can be resolved if the operating system prevents the formation of a circular chain of processes. This can be done by numbering the available resources in a system and requiring that each job requests resources in ascending order, i.e. a job holding Resource 1 can only request Resource 2, 3, and so on, This removes the possibility of a circular wait preventing deadlocks. However this approach may lead to long idle times for unused resources allocated in the lower numbers.

2.4.2 Avoidance

The operating system can avoid deadlocks if the system knows ahead of time the sequence of requests associated with each of the active processes.

The Banker's Algorithm proposed by Dijkstra (1965) can be used to regulate resource allocation to avoid deadlocks. It is based on a bank with a fixed amount of capital that operates on the following principles:

- No customer will be granted a loan exceeding the bank's total capital.
- All customers will be given a maximum credit limit when opening an account.
- No customer will be allowed to borrow over the limit.
- The sum of all loans won't exceed the bank's total capital.

To remain in a safe state the bank must ensure it has sufficient funds to satisfy at least one customer.

Example 2.4.1

Given a system with 10 resources available and three processes, with the following maximum resource requirements:

Job No.	Devices Allocated	Maximum Required	Remaining Needs
1	0	4	4
2	2	5	3
3	4	8	4
Total number of devices allocated = 6			
Total number of devices in system = 10			
Available devices = 4			

This is a **safe state** because available devices (4) ⩾ smallest remaining need (3 for Job 2), so at least one job can complete.
The OS allocates more resources to Jobs 1 and 2 and enters an unsafe state:

Job No.	Devices Allocated	Maximum Required	Remaining Needs
1	2	4	2
2	3	5	2
3	4	8	4
Total number of devices allocated = 9			
Total number of devices in system = 10			

This state is **unsafe** because with only one device available, the system can't satisfy anyone's minimum remaining needs (all jobs need at least 2 devices). If the system allocated the last resource to any process, it would enter a deadlock because it wouldn't be able to satisfy the remaining needs of any process. An unsafe state doesn't necessarily lead to a deadlock, but it does indicate the possibility of one occurring.
The operating system must never satisfy a request that moves it from a safe state to an unsafe state. Therefore, as user's requests are satisfied, the OS must identify the job with the smallest number of remaining resources and make sure that the number of available resources is always equal to or greater than the number needed for this job to run to completion.
The correct sequence of resource allocation to keep the system in a safe state is as follows:
Step 1: Starting from the initial safe state, allocate 3 resources to Job 2 (smallest remaining need):

Job No.	Devices Allocated	Maximum Required	Remaining Needs
1	0	4	4
2	5	5	0
3	4	8	4
Total number of devices allocated = 9			
Total number of devices in system = 10			

Job 2 completes and releases all 5 devices. Available devices = 1 + 5 = 6.
Step 2: Allocate 4 resources to Job 1:

Job No.	Devices Allocated	Maximum Required	Remaining Needs
1	4	4	0
2	*finished*	–	–
3	4	8	4
Total number of devices allocated = 8			
Total number of devices in system = 10			

Job 1 completes and releases all 4 devices. Available devices = 2 + 4 = 6.
Step 3: Allocate 4 resources to Job 3:

Job No.	Devices Allocated	Maximum Required	Remaining Needs
1	*finished*	–	–
2	*finished*	–	–
3	8	8	0
Total number of devices allocated = 8			
Total number of devices in system = 10			

Job 3 completes and releases all 8 devices. All jobs have completed successfully.
The safe sequence is: **Job 2 → Job 1 → Job 3**.

In implementing the Banker's Algorithm the system must setup a resource assignment table for each type of resource, and tracks each table to keep the system in a safe state.

Although the Banker's Algorithm can effectively avoid deadlocks, it has several limitations:

- As jobs enter the system they must declare the maximum number of resources they need, which is difficult in interactive systems.
- The number of total resources for each class must remain constant. If a device fails and becomes unavailable the algorithm may no longer be able to keep the system in a safe state.
- The number of jobs must remain fixed, which is hard to do in an interactive system.
- High overhead as the algorithm must be executed each time a jobs requests a resource.

2.5 Detection

Deadlock detection can be modelled as a directed graph problem, allowing use of cycle detection algorithms to identify deadlocks. Unlike avoidance algorithms which must be executed on each request, cycle detection algorithms can be executed when appropriate, i.e. every hour, once a day, or manually by the system administrator.

Beginning with a system in use, the steps to detect deadlocks are:

1. Find a process that is currently using a resource and waiting for one . This process can be removed from the graph by disconnecting the link tying the resource to the process. The resource can then be returned to the available list. This is possible because the process would eventually finish and return the resource.
2. Find a process that's waiting only for resources classes that aren't fully allocated. This process isn't contributing to deadlock since it would eventually get he resource it's waiting for, finish it's work and return the resource at he available list.
3. Repeat step 1 and continue with steps 1 and 2 until all the lines connect resources to processes have been removed.

If there are any edges left in the graph this indicates that request of the process in question can't be satisfied and the process is deadlocked.

2.6 Recovery

Once a deadlock is detected it must be resolved and the system returned to normal as quickly as possible. There are several recovery algorithms but they all have a common approach: sacrificing at least one victim, i.e. an expendable job, which when removed from the deadlock, will free the system. Removal generally involves restarting the job from the beginning or from a previously saved checkpoint.

The first approach is to terminate every job that's active in the system and restart them from the beginning. This is a simple approach but is drastic and results in significant loss of processing time.

The second approach is to terminate only the jobs involved in the deadlock and ask their users to resubmit them. This approach is less drastic but still results in significant loss of processing time for the terminated jobs.

The third approach is to identify which jobs are involved in the deadlock and terminate them one at a time, checking to see if the deadlock is resolved after each removal, until the deadlock is resolved. This approach minimizes the number of jobs terminated but requires more overhead to check for deadlock after each removal.

The fourth approach can be used only if the job keeps a record or a checkpoint of its progress so it can be interrupted and continued without starting again from the beginning of its execution.

The following methods focus on the non-deadlocked jobs and the resources they hold

The fifth approach is to select a non-deadlocked job, then preempt the resources it's holding, and allocates then to deadlocked process so that it can resume execution, resolving the deadlock. This approach requires the ability to preempt resources, which may not be possible for all resource types.

The sixth method stops new jobs from entering the system, allowing the non-deadlocked jobs to finish so they'll release their resources to the deadlocked jobs. This approach is the only one listed that doesn't require a sacrifice. This method

can be effective if the deadlock involves only a few jobs and the non-deadlocked jobs will finish in a reasonable amount of time, and is not guaranteed to work unless the number of avail be resources is greater than the number needed by at least one of the deadlocked jobs.

Several factors must be considered when selecting a sacrifice that will have the least-negative effect on the system, including:

Priority - High priority jobs are usually not sacrificed.

CPU time used - Jobs close to completion are usually not sacrificed.

Number of releated jobs - Jobs with many related jobs are usually not sacrificed.

2.7 Starvation

Definition 2.7.1: Starvation

A single job is prevented from completing its execution because it is waiting for resources that never become available.

Starvation can occur in systems that use priority scheduling, and can have the following consequences:

Increased Response Time - Starvation can lead to increased response times for low-priority jobs as they wait indefinitely for resources to become available.

Reduced System Throughput - Starvation can lead to a reduction in system throughput as low-priority jobs are unable to complete their execution.

Unfair Resource Allocation - Starvation can lead to unfair resource allocation, where high-priority jobs are given preferential treatment over low-priority jobs.

To address this problem an algorithm can be used to track how long each job has been waiting for resources, the same as **ageing**. Once starvation has been detected, the system can block incoming requests from other jobs until the starving jobs have been satisfied.

Chapter 3

Concurrent Processing

3.1 Parallel Processing

Definition 3.1.1: Parallel Processing

The simultaneous operation and execution of two or more processors in one system at the same time.

The processor manager coordinates the activities of each processor, and synchronizes interaction among CPUS. This allows for increased reliability, as there is failover capability if one processor fails, and faster processing as instructions can be executed in parallel across multiple processors.

Processing can be done in several ways:

- Each CPU could be allocated to each program or job
- Each CPU could be allocated to each working set or parts of it.
- Each individual instruction could be subdivided with each subdivision processed simultaneously

However there are challenges in parallel processing:

- Connecting processors into configurations that allow efficient communication among them.
- Orchestrating processor interaction

3.1.1 Levels Of Multiprocessing

Multiprocessing can occur at three levels:

Job Level - Several jobs are processed simultaneously, each job allocated to a different processor. No explicit synchronization.

Process Level - Several processes within a job are processed simultaneously, each process allocated to a different processor. Some level of synchronization.

Thread Level - Several threads within a process are processed simultaneously, each thread allocated to a different processor. High degree of synchronization needed to track each process and each thread.

3.1.2 Multi-Core Processors

Definition 3.1.2: Multi-Core Processor

A single computing component (chip) with two or more independent processing units called cores, which read and execute program instructions.

Multi-core processors suffer from current leakage (tunnelling) as a result of the small size of the transistors used to build them. This leakage causes increased power consumption and heat generation, which can lead to reduced performance and reliability. To mitigate these issues, multi-core processors often reduce the speed of groups of adjacent cores when they are in use at the same time to reduce heat generation.

3.2 Typical Multiprocessing Configurations

The configuration of multiple processors in a system affects how they communicate and synchronize with each other. There are three typical configurations:

- Master / Slave Configuration
- Loosely Coupled Configuration
- Symmetric Configuration

3.2.1 Master / Slave Configuration

An asymmetric multiprocessing configuration where one processor/core is designate as a master processor and all the others are designated as slave processors managed by the master. The master processor is responsible for

- Managing the system
- Maintaining the status of all processes
- Managing storage
- Schedules work for other processors
- Executes control programs

This configuration is simple to implement and manage.

However it has several disadvantages:

Reliability - If the master processor fails the entire system may become inoperable.

Poor Resources Usage - As slave processors may be idle while waiting for the master to allocate work to them.

Increased Interrupts - The master processor may become overwhelmed with interrupts from slave processors requesting work or reporting status.

3.2.2 Loosely Coupled Configuration

Several processors each manage their own resources, with each processor communicating and cooperating with others as needed.

This configuration requires several policies for job scheduling and conditions. There is failover capability if one processor fails, and processors can be added or removed from the system as needed however failure is difficult to detect.

This configuration has several disadvantages:

Requires Several Resouces - Each processor must have its own memory and I/O devices, increasing the cost of the system.

Complex Communication - Processors must communicate over a network, which can be slow and unreliable.

Complex Synchronization - Processors must synchronize their activities, which can be difficult to implement.

3.2.3 Symmetric Configuration

All processors share resources and communicate with each other with decentralized processor scheduling, where each processor uses the same scheduling algorithm.

This configuration has several advantages:

Relaiblity - If one processor fails, the others can continue to operate.

Efficient Resource Usage - Processors can share resources, reducing idle time.

Load Balancing - Work can be distributed evenly among processors, improving performance.

Graceful Degradation - The system can continue to operate at a reduced capacity if one or more processors fail.

However it has several disadvantages:

Difficult to Implement - Requires synchronization to prevent race conditions, deadlocks, and starvation.

Increased Overhead - Due to the need for interrupt processing and updating process lists.

More Conficts - As several processors access shared resources simultaneously.

3.3 Process Synchronization Software

Definition 3.3.1: Critical Region / Section

A part of a program that accesses a shared resource (data structure, file, device) that must not be concurrently accessed by more than one thread of execution.

To achieve process synchronization shared resources must be locked to indicate use by another process to prevent conflicts, and only when the resource is released can waiting processes access it.

Mistakes in synchronization can result in starvation, as jobs are left waiting indefinitely and deadlock if two or more jobs are waiting for resources held by each other.

In synchronization a critical region must be defined for each shared resource, allowing only one process to access the resource at a time. Processes within this critical region cannot be interleaved as this threatens the integrity of the operation.

Synchronization can be implemented as a lock-and-key arrangement, where processes determine key availability, i.e. if a process can enter the critical region or must wait. If a process wants to enter a critical region it must first obtain a key to gain access, this makes the key unavailable to other processes until the process leaves the critical region and releases the key.

There are three types of locking mechanisms:

- Test-and-Set
- WAIT and SIGNAL
- Semaphores

3.3.1 Test-and-Set

Definition 3.3.2: Atomic

An instruction that runs to completion without the possibility of interruption.

An atomic instruction that checks if a key is available, and if it is acquires it all in a single cycle. The key can be represented as a single bit in a storage location with zero indicating the key is available and one indicating the key is unavailable.

Before a process enters a critical region, it executes the test-and-set (TS) instruction if there is no process in the critical region, the TS instruction sets the key to one and allows the process to enter, when the process leaves the critical region it sets the key back to zero, allowing other processes to enter.

This approach has several advantages:

- Simple to implement
- Works well for a small number of processes

And several disadvantages:

Starvation - Many processes may be waiting to enter a critical region as processes gain access arbitrarily.

Busy Waiting - Waiting processes remain in unproductive, resource-consuming wait loops.

3.3.2 WAIT and SIGNAL

Improves upon test and set by removing busy waiting. This introduces two mutually exclusive operations:

WAIT - Activated when a process encounters the busy condition code. The process is placed in a waiting queue until the condition code is cleared.

SIGNAL - Activated when process exits the critical region and the condition code is cleared. If there are processes in the waiting queue, one is removed and allowed to enter the critical region.

3.3.3 Semaphores

A non-negative integer variable access through atomic operations used to indicate the status of a resource. It uses two atomic operations:

P (Proberen) - Tests the semaphore value, if it's greater than zero it decrements the value and allows the process to enter the critical region. If the value is zero the process is placed in a waiting queue until the value becomes greater than zero.

V (Verhogen) - Increments the semaphore value when a process exits the critical region. If there are processes in the waiting queue, one is removed and allowed to enter the critical region.

A semaphore with vale 0 indicates that the resource is unavailable, so the process calling the P operation is placed in the waiting queue, and must wait until the value is greater than 0. In this way a semaphore can allow multiple restricted access to a certain number of processes simultaneously by initializing the semaphore to the number of allowed processes.

This ensures mutual exclusion, as only the allowed number of processes can enter the critical region at the same time, while other processes are placed in the waiting queue.

3.4 Process Cooperation

Several processes may need to cooperate to complete a task, sharing data and resources. This requires synchronization and mutual exclusion to ensure that processes access shared resources in a controlled manner, and that data integrity is maintained. This can be approached using several models:

- Producer-Consumer Model
- Reader-Writer Model

3.4.1 Producers and Consumers

A model where one process produces data (producer), placing it into a buffer, and other processes consume the data (consumers). The producer generates data and places

3.4.2 Readers and Writers

A model where multiple processes read data from a shared resource (readers), while other processes write data to the shared resource (writers). Readers can access the shared resource simultaneously, but writers require exclusive access to prevent data corruption.

3.5 Concurrent Programming

A type of multiprocessing where one job executes multiple sets of instructions in parallel using several processors or cores. This requires programming language and system support.

Parallelism can be achieved at several levels:

Data Level Parallelism - The same operation is performed on multiple data items simultaneously.

Instruction Level Parallelism - Multiple instructions are executed simultaneously.

3.5.1 Amdahl's Law

Theorem 3.5.1 Amdahl's Law

Amdahl's Law is a formula that describes the theoretical speedup of a program when running on multiple processors. The formula states that the speedup of a program is limited by the fraction of the program that can be parallelized. Amdahl's Law is used to analyse the performance of parallel computing systems.

$$S(n) = \frac{1}{(1-P) + \frac{P}{n}}$$

Where $S(n)$ is the speedup of the program running on n processors, P is the fraction of the program that can be parallelized.

3.5.2 Order of Operations

The precedence of operations determines the priority in which arithmetic operations are evaluated, usually from left to right making use of BODMAS. These evaluations can be done in parallel if there are no dependencies among them, like in the following example:

$$Z = 10 - A/B + C(D+E)^{F-G}$$

This can be parallelized as follows:

$$\begin{aligned} T1 &= A/B \\ T2 &= D+E \\ T3 &= F-G \\ T4 &= T2^{T3} \\ Z &= 10 - T1 + C * T4 \end{aligned}$$

This could be spread across multiple CPUs using the COBEGIN and COEND constructs to indicate the beginning and end of concurrent processing.

3.5.3 Applications of Concurrent Programming

Concurrent programming can also be used to reduce complexity and computational time in many cases, including:

- Array Operations
- Matrix Multiplication
- Sorting and Merging Files
- Data Mining

3.6 Threads and Concurrent Programming

Threads are lightweight processes that can be executed concurrently within a single process. They share the same memory space and resources of the parent process, allowing for efficient communication and synchronization among threads.

Using threads minimizes overhead as they don't require separate memory allocation and context switching like full processes do. This makes threads ideal for tasks that require frequent communication and synchronization, such as in web servers and database management systems.

Each active process thread has its own program counter, stack, and local variables, but shares the same code segment, data segment, and other resources of the parent process.