Amdahl's Law

Amdahl's Law is a formula that shows the theoretical maximum speedup of a program when part of it is parallelized using multiple processors or threads.It helps us understand how much faster a program can run when we increase the number of processing units.

It is expressed as:

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

Where:

- S = overall speedup
- P= fraction of the program that can be parallelized
- N = number of processors (threads/cores)
- (1−P)= fraction of code that must run sequentially

Explanation:

- The parallel portion (P) can be divided among multiple processors and executed simultaneously.
- The sequential portion (1−P) must still run one step at a time — it becomes a bottleneck.
- As the number of processors (N) increases, the parallel part becomes faster, but the sequential part limits total speedup.

Suppose we have a program where:

- 80% of the code (P = 0.8) can be parallelized
- 20% (1 − P = 0.2) must be executed sequentially
- We use 8 cores (N = 8) for parallel execution

Substitute values:

$$\text{Speedup}(8) = \frac{1}{(1 - 0.8) + \frac{0.8}{8}}$$

$$\text{Speedup}(8) = \frac{1}{0.2 + 0.1} = \frac{1}{0.3} = 3.33$$

Even with 8 cores, the program runs only about 3.3 times faster than on one core.

Example 1: Two fork() calls

This demonstrates how multiple forks create a process tree.

```
#include <stdio.h>
#include <unistd.h>

int main() {
   printf("Start of program\n");

   fork();  // First fork
   fork();  // Second fork

   printf("Hello from process %d\n", getpid());
   return 0;
}
```

Explanation:

- Each fork() doubles the number of processes.
- Number of processes created = 2^n where n = number of fork() calls.
- Here: 2^2 =4 total processes.

Expected Output : order may vary

- Start of program
- Hello from PID 1234
- Hello from PID 1235
- Hello from PID 1236
- Hello from PID 1237

Example 2:

```
#include <stdio.h>
#include <unistd.h>

int main() {
   printf("Parent (PID=%d) started\n", getpid());

   pid_t pid1 = fork();  // First fork

   if (pid1 == 0) {
       printf("Child 1 (PID=%d, PPID=%d)\n", getpid(), getppid());
   } else {
       pid_t pid2 = fork();  // Second fork (only by parent)
```

```
        if (pid2 == 0) {
            printf("Child 2 (PID=%d, PPID=%d)\n", getpid(), getppid());
        }
    }

    return 0;
}
```

So total processes = 3

Expected Output (order may vary):

```
Parent (PID=1000) started
Child 1 (PID=1001, PPID=1000)
Child 2 (PID=1002, PPID=1000)
```

Output may appear in any order because process scheduling is not deterministic.

3. C multithreaded program where two threads execute *different tasks* (i.e., different “programs” or functions) using the POSIX Threads (pthread) library.

Thread 1: Prints all prime numbers between 1 and 50.

Thread 2: Calculates the sum of squares of numbers from 1 to 10.

Both threads execute different functions, concurrently

```
#include <stdio.h>
#include <pthread.h>
#include <math.h>
#include <stdbool.h>

// Function for thread 1 – print prime numbers
void* print_primes(void* arg) {
    int n = 50;
    printf("Thread 1: Prime numbers between 1 and %d:\n", n);

    for (int i = 2; i <= n; i++) {
        bool prime = true;
        for (int j = 2; j <= sqrt(i); j++) {
            if (i % j == 0) {
                prime = false;
                break;
            }
        }
        if (prime)
            printf("%d ", i);
    }
    printf("\n");
    pthread_exit(NULL);
}

// Function for thread 2 – calculate sum of squares
void* sum_of_squares(void* arg) {
    int n = 10;
    int sum = 0;

    for (int i = 1; i <= n; i++) {
        sum += i * i;
    }

    printf("Thread 2: Sum of squares of numbers 1 to %d = %d\n", n, sum);
    pthread_exit(NULL);
}
```

```
int main() {
    pthread_t t1, t2;

    printf("Main: Creating two threads...\n");

    // Create threads
    pthread_create(&t1, NULL, print_primes, NULL);
    pthread_create(&t2, NULL, sum_of_squares, NULL);

    // Wait for both threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Main: Both threads have finished execution.\n");

    return 0;
}
```

4. Different kernel architectures:

The **kernel** is the core component of an operating system, acting as a bridge between hardware and software. It manages critical system resources like CPU, memory, and devices, ensuring smooth communication and efficient multitasking. The kernel is responsible for process management, memory allocation, device handling, file system operations, and security enforcement.

Types of Kernel Architectures

1. **Monolithic Kernel**

In a **monolithic kernel**, all operating system services, such as process management, memory management, and device drivers, run in the kernel space. This design provides high performance due to minimal context switching but can lead to stability issues since a failure in one service can crash the entire system.

Examples: Linux, UNIX.

Advantages:
- High performance due to direct communication between services.
- Simpler design and implementation.

Disadvantages:
- Stability risks as a single failure can crash the system.
- Difficult to maintain and extend due to tightly coupled components.

2. **Microkernel**

A **microkernel** minimizes the functionality running in kernel space, delegating most services (e.g., file systems, device drivers) to user space. This design enhances modularity and stability but may incur performance overhead due to frequent context switching.

Examples: Minix, Mach.

Advantages:
- Improved reliability as failures in user-space services do not crash the kernel.
- Easier to extend and maintain due to modular design.

Disadvantages:
- Slower performance due to increased inter-process communication.
- More complex to design and implement.

3. **Hybrid Kernel**

A **hybrid kernel** combines features of monolithic and microkernels. It retains the performance of monolithic kernels while incorporating the modularity and stability of microkernels.

Examples: Windows NT, macOS.
Advantages:

- Better performance than microkernels.
- Enhanced reliability and flexibility.

Disadvantages:

- Increased complexity in design and maintenance.
- Larger resource usage compared to microkernels

4. Exokernel

An **exokernel** provides minimal abstractions and allows applications to directly manage hardware resources. This design is highly flexible and efficient but requires applications to handle resource management.

Examples: ExOS, Nemesis.

Advantages:

- High performance due to direct hardware access.
- Greater flexibility for application-specific optimizations.

Disadvantages:

- Complex to develop and debug.
- Limited support and adoption.

5. Nano Kernel

A **nano kernel** provides only the most essential hardware abstractions, leaving all system services to be implemented externally. It is similar to microkernels but even more minimalistic.

Examples: EROS.

Advantages:

- Extremely small and efficient.
- High modularity and portability.

Disadvantages:

- Limited functionality.
- Complex to develop and maintain.

Key Functions of a Kernel

- **Process Management**: Scheduling, creation, and termination of processes.

- **Memory Management**: Allocation, deallocation, and virtual memory handling.
- **Device Management**: Communication with hardware devices via drivers.
- **File System Management**: Handling file operations and storage.
- **Security**: Enforcing access control and ensuring system integrity.
- **Inter-Process Communication (IPC)**: Facilitating communication between processes.