**Processes and Multiprogramming**

Early computer systems were used to run a single program at a time. Whenever a user wanted to perform a calculation with a computer, they would submit the [**job**](https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/Glossary.html#term-job) to an administrator and receive the results later. Administrators quickly realized that they could save time by [**batching**](https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/Glossary.html#term-batch) and submitting multiple jobs at the same time. Batch processing reduced the number of times the administrator had to load programs manually, adding and removing the [**monitor**](https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/Glossary.html#term-monitor) code as needed. It also increased the amount of computing time that could be accomplished, as a new job could be started immediately after the previous job finished.

In short, batch processing **reduced the amount of time wasted between the execution of multiple jobs.** Eliminating wasted time like this was very important, as these early computers were very expensive. To justify the expense, designers and users needed to get as much work out of the computer as possible. While batch processing helped in this regard, early computing pioneers quickly realized that another source of wasted time remained: the CPU sat idle while the rest of the computer performed very slow I/O operations. Eliminating this wasted CPU time became the goal of [**multiprogramming**](https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/Glossary.html#term-multiprogramming).

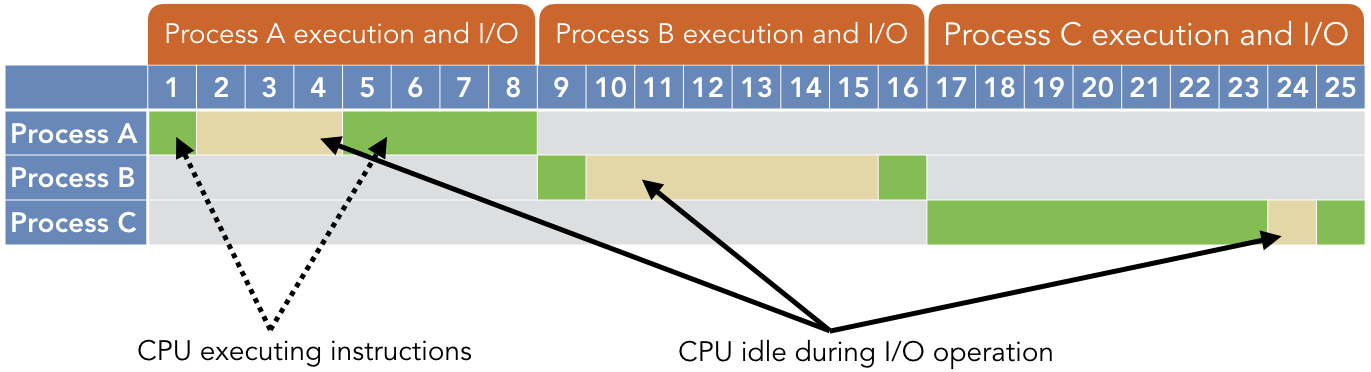
## 2.2.1. Uniprogramming and Utilization

Early batch processing systems used a strategy known as [**uniprogramming**](https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/Glossary.html#term-uniprogramming). In uniprogramming, one program was started and run in full to completion; the next job would start immediately after the first one finished. The problem with this approach is that programs consisted of both CPU instructions and I/O operations. CPU instructions were very fast, as they consisted of electrical signals. I/O operations, however, were very slow. One example of an early I/O device was the drum memory, which was a large magnetic device that had to be mechanically turned for every data access. If a program required many I/O operations to be performed, that created a lot of wasted time where the CPU sat idle instead of executing instructions.

We can quantify this wasted time as the CPU utilization. In general terms, [**utilization**](https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/Glossary.html#term-utilization) can be defined mathematically as the actual usage of a resource divided by the potential usage. Utilization is reported as a unit-less ratio, typically a percentage. For instance, if a system is only used for half of the time that it could be, we would say that it experienced 50% utilization. In regard to CPU time, utilization is calculated as the following ratio:

**CPU utilization=total CPU time/total real time**

Consider the following timeline illustration for three sequential uniprogramming processes.



The green regions indicate times when the CPU is executing instructions in the program, while the yellow indicates that the times where the CPU is idle while waiting on an I/O operation to complete. The following table summarizes the time each process spends executing on the CPU or waiting for I/O:

| **Process** | **CPU time** | **I/O time** |
| --- | --- | --- |
| A B C | 5 2 8 | 3 6 1 |
| **Total** | **15** | **10** |

In this scenario, the CPU was used for a total of 15 out of 25 possible seconds, so the system experienced 60% CPU utilization when running these three jobs:

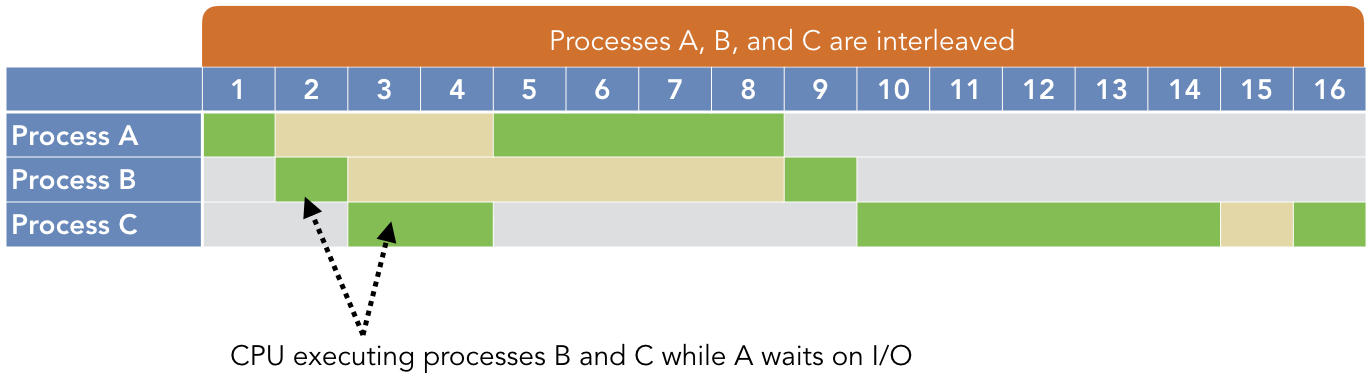
CPU utilization=5+2+8/5+2+8+3+6+1=15/25=60%

**Given that waiting on I/O operations to complete is slow and wasteful, system designers proposed a straightforward solution: switch to another process while an I/O operation is being performed.**

## Multiprogramming and Concurrency

In [**multiprogramming**](https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/Glossary.html#term-multiprogramming), several processes are all loaded into memory and available to run. Whenever a process initiates an I/O operation, the kernel selects a different process to run on the CPU. This approach allows the kernel to keep the CPU active and performing work as much as possible, thereby reducing the amount of wasted time. By reducing this waste, multiprogramming allows all programs to finish sooner than they would otherwise.

Consider the following timeline illustration for the same three processes from [**Example 2.2.1**](https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/Multiprogramming.html#uniprogex), but in a multiprogramming environment.



As before, the green regions indicate CPU execution and the yellow indicates I/O operations. However, note that processes B and C can run while A is waiting on its I/O operation. Similarly, A and C execute while B is waiting on I/O operations. As a result, the CPU is only completely idle while C’s I/O operation is performed at time 15, because A and B have already run to completion.

In our revised CPU utilization calculation, the numerator does not change because the total amount of CPU execution time has not changed. Only the denominator changes, to account for the reduced time wasted waiting on A’s and B’s I/O operations.

CPU utilization=5+2+8/5+2+8+1=15/16=93.75%

There are two forms of multiprogramming that have been implemented. The most common technique is [**preemptive multitasking**](https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/Glossary.html#term-preemptive-multitasking), in which processes are given a maximum amount of time to run. This amount of time is called a [**quantum**](https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/Glossary.html#term-quantum), typically measured in milliseconds. In preemptive multitasking, if a process issues an I/O request before its quantum has expired, the kernel will simply switch to another process early. However, if the quantum has expired (i.e., the time limit has been reached), the kernel will preempt the current process and switch to another. In contrast, with [**cooperative multitasking**](https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/Glossary.html#term-cooperative-multitasking), a process can run for as long as it wants until it voluntarily relinquishes control of the CPU or initiates an I/O request.

A simple way to illustrate multiprogramming in modern software is with the sleep() function. This function’s only argument is the number of seconds to pause the current process. During this time, the system will switch to other processes that need to run. In other words, calling sleep() can be interpreted as a form of cooperative multitasking.

**C library functions - <unistd.h>**

**unsigned sleep(unsigned seconds);**

Suspend the current process for a number of seconds.

*/\* Code Listing 2.2:*

*Repeatedly pausing the current function to illustrate multiprogramming*

*\*/*

**for** (**int** i = 0; i < 10; i++)

{

printf ("Hello!\n");

sleep (1);

}

shows an example of using sleep() to introduce repeated pauses in a process’s execution. This code will print “Hello!” 10 times, pausing for one second between each print. By running this code along with other programs (such as a media player or web browser), you can clearly observe that your computer is still operating during these 10 seconds.

**Time-Sharing Operating Systems** is one of the important type of operating system.

Time-sharing enables many people, located at various terminals, to use a particular computer system at the same time. Multitasking or Time-Sharing Systems is a logical extension of multiprogramming. Processor’s time is shared among multiple users simultaneously is termed as time-sharing.

The main difference between Time-Sharing Systems and Multiprogrammed Batch Systems is that in case of Multiprogrammed batch systems, the objective is to maximize processor use, whereas in Time-Sharing Systems, the objective is to minimize response time.

Multiple jobs are implemented by the CPU by switching between them, but the switches occur so frequently. So, the user can receive an immediate response. For an example, in a transaction processing, the processor executes each user program in a short burst or quantum of computation, i.e.; if n users are present, then each user can get a time quantum. Whenever the user submits the command, the response time is in few seconds at most.

An operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time. Computer systems which were designed primarily as batch systems have been modified to time-sharing systems.

Advantages of Timesharing operating systems are −

* It provides the advantage of quick response.
* This type of operating system avoids duplication of software.
* It reduces CPU idle time.

Disadvantages of Time-sharing operating systems are −

* Time sharing has problem of reliability.
* Question of security and integrity of user programs and data can be raised.
* Problem of data communication occurs.