Part 81
-
Multi-Tasking
-
Intro

Version: 2021-01-14

What is multi-threading?
In a multithreading system, multiple tasks share a common processing resource such as the CPU. In such a system we can run multiple applications and the operating system can quickly switch between each task to give the impression that all the tasks are executing simultaneously. The CPU clock speed is so fast that the tasks run very fast and are switched quickly, resulting in high performance overall.

In single-core CPU based systems, only one task can run at any point in time. Multithreading is implemented in such systems by scheduling which task may run at any given time, and other tasks are waiting to be given CPU/core time.

In a multi-core CPU based system, there is more than one core/processor per CPU and the operating system can execute multiple tasks concurrently, where the tasks can work independently of each other. For example, in a quad-core CPU based system, four applications such as e-mail, word processing, spreadsheet, and web browsing can each access a separate core/processor at the same time. The user in such a system can perform tasks simultaneously, and hence an improved overall performance is obtained.

In a multithreading system, there could be several threads (or tasks) in a program, and the same memory space is used by all the tasks and as a result, precautions should be taken while reading or writing to the common memory.
The Python language supports multithreaded programming and provides libraries to make it easier to create such applications.

What is multi-processing?
In general, multiprocessing is similar to multithreading where more than one process is required to run on the system. Perhaps the main difference between the two is that in multiprocessing the processes are completely independent of each other with each one having its own memory space. Software libraries (APIs) are then used to establish synchronization, pass data between processes, and establish communication between the processes.

Differences between multi-threading and multi-processing
It is important to know the difference between multithreading and multiprocessing based applications. A comparison of both methods is summarised below.

- Threads share the same variables and run in the same memory space. Processes, on the other hand, have separate memories
- Sharing objects between threads is easier, but synchorisation to make sure that two threads will not write to the same object at the same time is important. As a result of this, thread-based programming is more prone to errors. On the other hand, process-based programming is less error-prone.
- Threads have a lower overhead compared to processes. Starting processes takes a longer time.
- Threads cannot achieve true parallelism. Processes on the other hand do not have such restrictions.
- Thread scheduling is done using the Python interpreter, while process scheduling is handled by the operating system.
- Child threads can't be easily terminated. On the other hand, they can be easily killed.
- It is recommended to use threads involving I/O based programming or with programs having user interactions. Processes, on the other hand, should be used in CPU bound computation-intensive programs
- Python offers two libraries in the form of APIs: multiprocessing and threading.

Task Scheduling algorithms

Although many variations of scheduling algorithms are in use today, the three most commonly used algorithms are:

- Co-operative scheduling
- Round-robin scheduling
- Pre-emptive scheduling

The type of scheduling algorithm used, depends on the nature of the application. In general, most applications use either one of the above algorithms, a combination of them, or a modified version of them.

Co-operative scheduling

Co-operative scheduling, also known as non-preemptive scheduling, shown in figure below, is perhaps the simplest algorithm where tasks voluntarily give up CPU usage when they have nothing useful to do or when they are waiting for resources to become available. This algorithm has the main disadvantage that certain tasks can use excessive CPU times, thus not allowing other important tasks to run when needed. Co-operative scheduling is only used in simple multitasking systems where there are no time-critical tasks.

State machines, also called finite-state machines (FSM) are probably the simplest ways of implementing co-operative scheduling. A while loop can be used to execute the tasks, one after one, as shown below for a 3-task application. In the code below, a task is represented with a function:

```
Task1()
{
  Task 1 code
}

Task2()
{
  Task 2 code
}

Task3()
{
  Task 3 code
}

while(1)
{
  Task1();
  Task2();
  Task3();
{
```

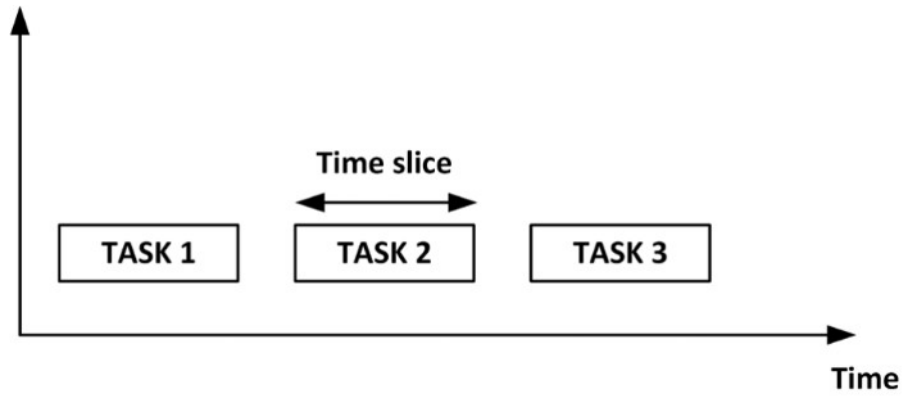
The tasks are executed one after the other inside the main infinite loop formed using a while statement. In this simple approach, the tasks can communicate with each other using global variables declared at the beginning of the program. The tasks in a co-operative scheduler must satisfy the following requirements for the successful running of the overall system:

- Tasks must not block the overall execution, e.g. by using delays or waiting for some resources and not releasing the CPU.
- The execution time of each task should be acceptable to other tasks
- Tasks should exit as soon as they complete their processing
- Tasks do not have to run to completion and they can exit for example before waiting for a resource to be available
- Tasks should resume their operations from the point after they release the CPU.

The last requirement listed above is very important and is not satisfied in the simple scheduler example given above. Resuming a task requires the address of the program counter and important variables when the task releases the CPU to be saved, and then restored when the task resumes (also called context switching) so that the interrupted task can continue normally as if there has not been any interruption.

Round-robin scheduling

Round-robin scheduling allocates each task an equal share of the CPU time. Tasks are in a circular queue and when a task's allocated CPU time expires, it is removed and placed at the end of the queue. This type of scheduling cannot be satisfactory in any real-time application where each task can have a varying amount of CPU requirements depending on the complexity of the processing involved. Round-robin scheduling requires the context of the running task to be saved on the stack when a task is removed from the queue so that the task can resume from the point it was interrupted when it becomes active again. One variation of the pure Round-robin based scheduling is to provide priority-based scheduling, where tasks with the same priority levels receive equal amounts of CPU time.



Round-robin scheduling has the following advantages:

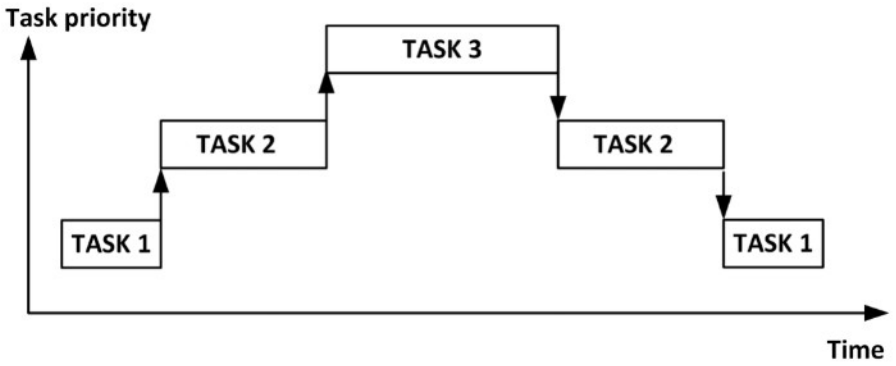
- It is easy to implement
- Every task gets an equal share of the CPU
- Easy to compute the average response time

The disadvantages of Round-robin scheduling are:

- It is not generally good to give the same CPU time to each task
- Some important tasks may not run to completion
- Not suitable for real-time systems where different tasks have different processing requirements

Pre-emptive scheduling

Pre-emptive scheduling is the most commonly used scheduling algorithm in real-time systems. Here, the tasks are prioritized and the task with the highest priority among all other tasks gets the CPU time. If a task with a priority higher than the currently executing task becomes ready to run, the kernel saves the context of the current task and switches to the higher priority task by loading its context. Usually, the highest priority task runs to completion or until it becomes non-computable, for example waiting for a resource to become available, or calling a function to delay. At this point, the scheduler determines the task with the highest priority that can run and loads the context of this task and starts executing it. Although pre-emptive scheduling is very powerful, care is needed as a programming error can place a high priority task in an endless loop and thus not release the CPU to other tasks.



Some multitasking systems employ a combination of Round-robin and pre-emptive scheduling. In such systems, time-critical tasks are usually prioritized and run under pre-emptive scheduling, whereas the non-time-critical tasks run under Round-robin scheduling, sharing the left CPU time among themselves.

It is important to realize that, in a pre-emptive scheduler, tasks at the same priorities run under Round-robin. In such a system, when a task uses its allocated time, a timer interrupt is generated by the scheduler which saves the context of the current task and gives the CPU to another task with an equal priority that is ready to run, provided that there are no other tasks with higher priorities which are ready to run.

The priority in a pre-emptive scheduler can be static or dynamic. In a static priority system, tasks use the same priority all the time. In a dynamic priority-based scheduler, the priority of tasks can change during their courses of execution.

So far, we have said nothing about how various tasks work together in an orderly manner. In most applications, data and commands must flow between various tasks so that the tasks can co-operate and work together. One very simple way of doing this is through shared data held in RAM where every task can access. Modern RTOS systems, however, provide local task memories and inter-task communication tools such as mailboxes, pipes, queues, etc to pass data securely and privately between various tasks. Also, tools such as event flags, semaphores, and mutexes are usually provided for inter-task communication and synchronization purposes and passing data between tasks.

The main advantage of a pre-emptive scheduler is that it provides an excellent mechanism where the importance of every task may be precisely defined. On the other hand, it has the disadvantage that a high priority task may starve the CPU such that lower priority tasks can never have the chance to run. This can usually happen if there are programming errors such that the high priority task runs continuously without having to wait for any system resources and never stops.

Scheduling algorithm goals
It can be said that a good scheduling algorithm should possess the following features:

- Be fair such that each process gets a fair share of the CPU
- Be efficient by keeping the CPU busy. The algorithm should not spend too much time to decide what to do
- Maximize throughput by minimizing the time users have to wait
- Be predictable so that same tasks take the same time when running multiple times
- Minimize response time
- Maximize resource use
- Enforce priorities
- Avoid starvation

Difference between pre-emptive and non-preemptive scheduling
Some differences between a pre-emptive and non-pre-emptive scheduling algorithm summarized

Non-pre-emptive Scheduling	Pre-emptive Scheduling
• Tasks have no priorities	• Tasks have priorities
• Tasks cannot be interrupted	• A higher priority task interrupts a lower priority one
• Waiting and response times are longer	• Waiting and response times are shorter
• Scheduling is rigid	• Scheduling is flexible
• Tasks do not have priorities	• High priority tasks run to completion
• Not suitable for real-time systems	• Suitable for real-time systems

Some other scheduling algorithms
There are many other types of scheduling algorithms used in practice. Most of these algorithms are a combination or a derivation of the basic three algorithms described. Brief details of some other scheduling algorithms are outlined in this section.

First come, first served
This is one of the simplest scheduling algorithms, is known as FIFO scheduling. In this algorithm, tasks are run in the order they become ready. Some features of this algorithm are:

- Throughput is low since long processes can hold the CPU, causing short processes to wait for a long time
- There is no proritization and thus real-time tasks cannot be executed quickly
- It is non-pre-emptive
- The context switching occurs only on task termination and therefore the overhead is minimal
- Each process gets the chance to be executed even if they have to wait for a long time

Shortest remaining time first
In this algorithm, the scheduler arranges the tasks with the least estimated processing time remaining to be next in the queue. Some features of this algorithm are:

- If a shorter task arrives, the currently running task is interrupted, thus causing overhead.
- Waiting time of tasks requiring long processing times can be very long
- If there are too many small tasks in the system, longer tasks may never get the chance to run

Longest remaining time first
In this algorithm, the scheduler arranges the tasks with the longest processing times to be next in the queue. Some features of this algorithm are:

- If a longer task arrives, the currently running task is interrupted, thus causing overhead.
- Waiting time of tasks requiring short processing times can be very long
- If there are too many long tasks in the system, shorter tasks may never get the chance to run

Multilevel queue scheduling
In this type of scheduling, tasks are classified into different groups, such as interactive (foreground) and batch (background). Each group has its scheduling algorithm, Foreground tasks are given higher priorities since the background tasks can always wait.

Dynamic priority scheduling
In dynamic priority scheduling, although the tasks have priorities, their priorities can change, i.e. the priority can be lower or higher than it was earlier. Dynamic priority algorithms achieve high processor utilization, and they can adapt to dynamic environments, where task parameters are unknown. On the contrary, it is not advisable to use dynamic priority in real-time systems because of the uncertainty that an important task may not run in time.

Choosing a scheduling algorithm
When designing a multitasking system with several tasks, a programmer must consider which scheduling algorithm will perform the best for the application to be developed. In simple terms, most real-time systems should be based on preemptive scheduling with fixed priorities where time-critical tasks grab and use the CPU until they complete their processings or wait for a resource to be available. If there are several time-critical tasks, all such tasks should run at the same higher priorities. In general, tasks at the same priorities run as Round-robin and share the available CPU time. Then, all the other tasks which are not time-critical should run at lower priorities. If the time taken for a task to complete is not critical then simple co-operative scheduler algorithms can be employed.