# White Paper: Scheduling Priorities

---

# TMuLimit

# Everything you never wanted to know about OS "Multi-tasking"...



The Microsoft Operating system uses a pre-emptive thread scheduler based on a micro-kernel design. It you know what that means, there probably is no need to read on further. For most of us, though, we need some explanations.

## Multi-tasking

In our personal lives, multi-tasking is doing more than one thing at a time. Maybe it is reading this paragraph while singing to a song on the radio (or internet!) in the background. The capability to perform two unrelated activities, called *tasks*, at the same time is something we take for granted with most computers today.

However whether it is people or computers, neither is really performing two different tasks at the same time. At least not as long as we exclude computers with multiple CPUs and people with more than one head! What happens is that very small portions of time are given to each task. First one, then the other. Back and forth. As long as each task gets enough attention, and it does not take too long between turns, it seems as though we are doing two things at once.

## Priorities

Going back to our singing reader for a moment, it would seem that giving equal time slices to the two tasks might not be the most efficient. And it isn't. Reading this requires more concentration (brain cycles) than singing an old favorite song. If we gave equal cycles to these two tasks it would take much longer to finish reading than if we gave more cycle-time to the reading than the singing. There are parallels in computing as well. Some tasks will need more CPU-cycles than others. So to deal with this we use *priorities*. Priorities allow us to designate how important a task is relative to another task.

Sing Read Sing Read Sing Read

**Example of prioritized tasking**

Now in reality, the human brain is capable of actually doing two tasks at once, even if we aren't conscious of it. The computer, however, cannot. We should probably leave the human mind example behind now, and focus on the computer operating systems.

In the example above, we should not assume that these intervals are fixed intervals. It would be wasteful of CPU cycles if a task completed all it had to do early and we could not allow another task to start until the time-slot is complete. There are two basic ways used

to implement task priorities in computers. We describe these methods as *Pre-emptive* and *Non pre-emptive*. We will describe these in a moment, but first we need to cover some background material on how the operating system multi-tasks.

## The Executive

At the heart of any modern operating system is a portion called the *executive*. This is where multi-tasking takes place. The executive is also often called the *kernel*, although many times modern systems will have a kernel that is built on top of a *micro-kernel* which is where the executive resides. The executive is responsible for (among other things) the prioritized scheduling of the various tasks. This scheduling considers both the relative priorities of the tasks, as well as their state. While task state can get quite complicated in some systems, we can usually summarize the state as indicating that the task either has something to perform or is does not. When a task state indicates that it does not have something to do right now, it will be skipped (in spite of it's priority).

## Pre-emption and Non Pre-emption

The simplest form of scheduling is with Non-preemption. The executive simply prioritizes the order in which tasks are given a chance to run. When the task signals it is done with it's time slice, it contacts the executive so that another task may run for a while.

1. Find highest priority task in run-able state. If there is more than one, take the one that has waited the longest time

2. Let it run until it returns.

3. Go to step 1.

**Example of Non Pre-emptive kernel logic**

Non pre-emptive operating systems are typically used only in devices that require well scheduled periodic slices. An example would be a small device that needed to read or process some external events on a regular basis. In particular, if the device has to poll some hardware for the information (or regularly feed the hardware with it) rather than be interrupted, non pre-emption may be the only solution.

Non pre-emption fails to work when one of the tasks fails to signal the executive for an unreasonable period of time, effectively blocking out all other tasks. This is why you will only find non-preemption used in closed systems like the software running in the microprocessor of a small device. Non pre-emption becomes more difficult for the designer as more tasks are added to the system, thus it is more rarely used on larger systems. A non pre-emptive operating system is often also called a "Real-time" operating system.

In a pre-emptive based executive, the executive allocates time-slots (also called time-slices), letting each task run for a set period of time. At the end of the time, the executive steps in and gives the next task a chance. Priorities are used to regulate how long a time slice the task is given, and/or how often it is given a time slice.

1. Choose the next task to run. Use logic similar to the non pre-emptive case.

2. Let it run until it returns or a certain period of time - whichever occurs first.

3. Go to step 1.

**Example of Pre-emptive kernel logic**

As in the non pre-emptive case, the task may also signal to the executive that is does not need the remaining time in it's slice. This might occur, for example, if the task is waiting for an external event such as a response from a server or for the user to type a key.

Because the executive will pre-empt a user task without the programmer having to signal the executive, it is the choice for modern operating systems such as Apple, Microsoft, Linux, and Unix.

## Executive Performance

Of course no discussion of the executive would be complete without mentioning the performance of the executive. The executive has to monitor the running task - especially in the pre-emptive case. The key component used to describe the executive performance is called *context-switching* time. This is the amount of time (or cpu cycles) needed to swap out one task and restart the next. Other performance factors include how often the executive runs and how complicated the decision making process is in determining task priorities.

The other topic that should be mentioned, but has a hard time finding a home in this paper, is that of the *idle task*. Every executive has an idle task. The idle task is the single lowest priority task and will run only when all other tasks are not in a run-able state. Usually the idle task does nothing more than help track idle time. In a pre-emptive system, every time the executive wakes up (to determine if it should perform a context switch), it notes what task is running. It can get fancy and use the time of context switching also, but most OSs seem to use the former.

## Tasks, Processes, and Threads

So far we have talked generically about tasks. Now it is time to bring in two other terms that are used in modern operating systems. A *thread* is in essence, what has been called a task in this paper so far. It is an executable unit that gets scheduled by the executive.

A *process* is something bigger in the modern OSs. When an application begins, the kernel allocates a number of resources, such as virtual memory, to the application and starts a process. The process may have quite a few threads. Each thread performing portions of the overall application.

In the Microsoft Operating Systems, for example, you might start up a word processor application. That application will, in turn, create a number of threads. For example, when a File->Open request is made, a new thread will be started to handle the dialog box that pops up so that the user can browse to the file that opens. Looking at the *Windows Task Manager*, one can see the process (e.g. winword.exe), but not the threads. You can get visibility into the threads with debugging tools, but for the most part, Microsoft hides the details on a "need to know" basis.

## Microsoft and Priorities

Microsoft applies what it calls a *Priority Class* to processes. Most of the versions of Windows use the following classes:

1 Idle Class
2 Normal Class
3 High Class
4 Real-time Class

**Base Priority Classes Win 9x & NT**

On Windows 2000, two new classes were added:

5 Below Normal Class
6 Above Normal Class

**Base Priority Classes Win 2K & XP**

These fit between Idle/Normal and Normal/High classes. We should also note that the *System Idle* process is actually a priority below that of the Idle Class. It is important that the System Idle Process remain below all other processes in priority.

We are showing their values in the tables because it helps to illustrate the point that these are priority classes and not real priorities. They are classes used by the kernel to set the *base priority* value of any threads that are part of the process. These priority values run in a range from 0 to 31. Each of these classes maps to a base priority value.

Each process, in the Microsoft operating systems, is assigned a Priority Class. If you look at the priority classes of processes on a typical Windows machine, you will quickly notice that almost everything is in the "Normal Class". (Note: In the task manager, right click on the columns to add the priority column to the display).

Each thread in turn inherits a base priority value mapped from the priority class of it's parent process. The thread will also have a *priority offset value*. The operating system will adjust individual thread priorities based upon their activity. It is the combination of the base priority, inherited from the parent process priority class, plus or minus the priority offset value, that is a threads actual priority. Windows primarily relies upon thread priority offsets (plus, of course, state) to manage it's scheduling activities .

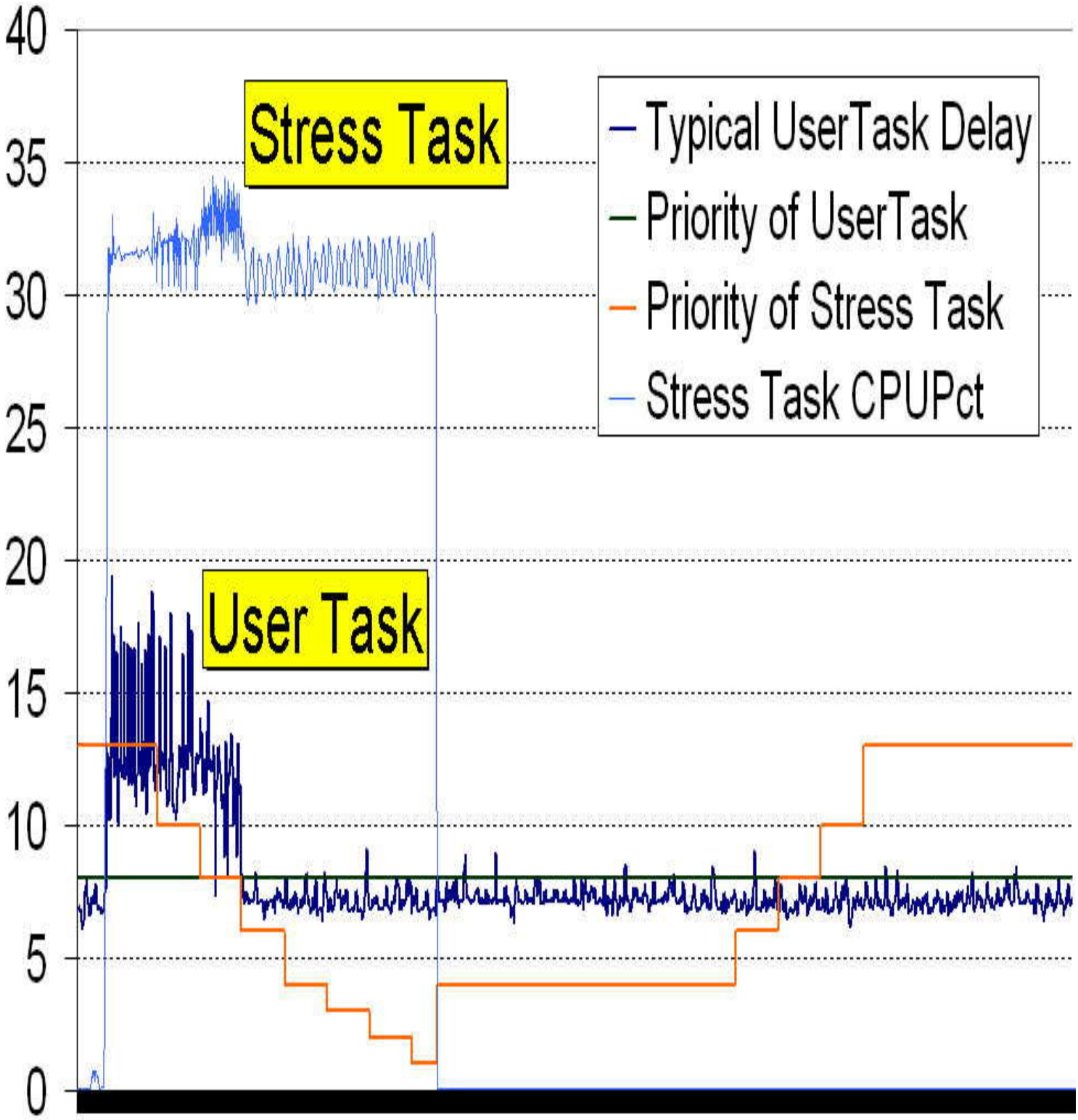
I should also note that Windows has other scheduling complications, such as "foreground" and "background" tasks, plus that of multiple rings (which help create what we know of as "user mode" and "kernel mode"). However, the bottom line, in my humble opinion is that the operating system is more concerned in ensuring that a task that is not ready to run does not than making sure all tasks get a fair shake. While relatively acceptable when there is only one human using the computer (I know that I can't get the start button to pop-up the menu right away because outlook is still shutting down), the situation can be less than tolerable on a Terminal Server where it is because someone else is hogging the CPU.

## ...which is why we have TMuLimit

We can, however, detect thread and process CPU utilization, and help the operating system

to adjust the prioritizations. TMuLimit uses a novel approach, akin to Quality of Service devices in the networking world. Just as TMuLimit deals with multiple users independently requesting a common resource (CPU cycles), these QoS solutions deal with multiple users independently requesting a common resource of network bandwidth. In both cases, the amount of resource an individual entity needs varies greatly over time, and at times there just is not enough to satisfy everyone. By applying thresholds and quotas, we can provide a reasonably quality experience for everyone.

The graph below shows an example of how this works on a Compaq Server. In this example we created a "User Task" that represents a typical user activity (launching a program, doing some graphics, performing some CPU intensive arithmetic). We measure the amount of wall clock time it takes to perform this task each time it is repeated. Under reasonable conditions, it takes about 65ms to perform these tasks. We also created a "Stress Task" that represents a badly behaved application. In this case, the application will perform a variety of functions that will fairly consistently use up 31% of the CPU, when in an active state.




We start the Stress Task as a High priority class(P13), while the User Task runs at Normal priority class(P8). As you can see in the graph, when the Stress Task becomes active, the User Task now takes over twice as long to complete it's task. We specially tuned TMuLimit to take a slow and fairly timid approach to managing the server. We set a 20% CPU threshold, and we can see how it slowly lowers the priority of the Stress Test once it exceeds the 20% limit. Because our server was not fully loaded, the Stress Task is still able to fully perform it's functions. Notice how the performance of the User Task (Dark Blue line) is restored as soon as the Stress Task priority is lowered enough. You can see how the User Task is performing optimally once the priority of the Stress Task (Orange

line) is lowered below that of the User Task (Green Line). In fact, you can't even tell when the Stress Task goes inactive by looking only at the User Task results!  Of course, with a more aggressive configuration TMuLimit could have lowered the priority of the Stress Task in about a tenth of a second of activation (however this test makes a better visual).

**[end]**