The biggest difference, in Linux, is that threads share most of their address space. IE: if one thread makes a change to stuff in memory, other threads can see (and be affected by) that change.

In other OSes, the differences are more profound. But in Linux, threads are almost (but not quite) processes in their own right -- they even have PIDs.

**Multitasking**, in an operating system, is allowing a user to perform more than one computer task (such as the operation of an application program) at a time. The operating system is able to keep track of where you are in these tasks and go from one to the other without losing information

**Multithreading** is the ability of a program or an operating system process to manage its use by more than one user at a time and to even manage multiple requests by the same user without having to have multiple copies of the program running in the computer

A monolithic kernel is a kernel where all services (file system, VFS, device drivers, etc) as well as core functionality (scheduling, memory allocation, etc.) are a tight knit group sharing the same space. This directly opposes a microkernel.

A microkernel prefers an approach where core functionality is isolated from system services and device drivers (which are basically just system services). For instance, VFS (virtual file system) and block device file systems (i.e. minixfs) are separate processes that run outside of the kernel's space, using IPC to communicate with the kernel, other services and user processes. In short, if it's a module in Linux, it's a service in a microkernel, indicating an isolated process.

Do not confuse the term modular kernel to be anything but monolithic. Some monolithic kernels can be compiled to be modular (e.g Linux), what matters is that the module is inserted to and runs from the same space that handles core functionality.

The advantage to a microkernel is that any failed service can be easily restarted, for instance, there is no kernel halt if the root file system throws an abort.

The disadvantage to a microkernel is that asynchronous IPC messaging can become very difficult to debug, especially if [fibrils](http://lwn.net/Articles/219954/) are implemented. Additionally, just tracking down a FS/write issue means examining the user space process, the block device service, VFS service, file system service and (possibly) the PCI service. If you get a blank on that, its time to look at the IPC service. This is often easier in a monolithic kernel. [GNU Hurd](http://www.gnu.org/software/hurd/) suffers from these debugging problems ([reference](http://www.gnu.org/software/hurd/open_issues/bash.html)). I'm not even going to go into checkpointing when dealing with complex message queues. Microkernels are not for the faint of heart.

The shortest path to a working, stable kernel is the monolithic approach. Either approach can offer a POSIX interface, where the design of the kernel becomes of little interest to someone simply wanting to write code to run on any given design.

I use Linux (monolithic) in production. However, most of my learning, hacking or tinkering with kernel development goes into a microkernel, specifically [HelenOS](http://helenos.org/).