Real, User and Sys process time statistics

One of these things is not like the other. Real refers to actual elapsed time; User and Sys refer to CPU time used *only by the process.*

- **Real** is wall clock time - time from start to finish of the call. This is all elapsed time including time slices used by other processes and time the process spends blocked (for example if it is waiting for I/O to complete).
- **User** is the amount of CPU time spent in user-mode code (outside the kernel) *within* the process. This is only actual CPU time used in executing the process. Other processes and time the process spends blocked do not count towards this figure.
- **Sys** is the amount of CPU time spent in the kernel within the process. This means executing CPU time spent in system calls *within the kernel,* as opposed to library code, which is still running in user-space. Like 'user', this is only CPU time used by the process. See below for a brief description of kernel mode (also known as 'supervisor' mode) and the system call mechanism.

`User+Sys` will tell you how much actual CPU time your process used. Note that this is across all CPUs, so if the process has multiple threads it could potentially exceed the wall clock time reported by `Real`. Note that in the output these figures include the `User` and `Sys` time of all child processes (and their descendants) as well when they could have been collected, e.g. by `wait(2)` or `waitpid(2)`, although the underlying system calls return the statistics for the process and its children separately.

As a general rule:

real < user	The process is CPU bound and takes advantage of parallel execution on multiple cores/CPUs.
real ≈ user	The process is CPU bound and takes no advantage of parallel execution.
real > user	The process is I/O bound. Execution on multiple cores would be of little or no advantage.

Origins of the statistics reported by `time (1)`

The statistics reported by `time` are gathered from various system calls. 'User' and 'Sys' come from `wait (2)` or `times (2)`, depending on the particular system. 'Real' is calculated from a start and end time gathered from the `gettimeofday (2)` call. Depending on the version of the system, various other statistics such as the number of context switches may also be gathered by `time`.

On a multi-processor machine, a multi-threaded process or a process forking children could have an elapsed time smaller than the total CPU time - as different threads or processes may run in parallel. Also, the time statistics reported come from different origins, so times recorded for very short running tasks may be subject to rounding errors, as the example given by the original poster shows.

A brief primer on Kernel vs. User mode

On Unix, or any protected-memory operating system, 'Kernel' or 'Supervisor' mode refers to a privileged mode that the CPU can operate in. Certain privileged actions that could affect security or stability can only be done when the CPU is operating in this mode; these actions are not available to application code. An example of such an action might be to manipulate the MMU to gain access to the address space of another process. Normally, user-mode code cannot do this (with good reason), although it can request shared memory from the kernel, which *could* be read or written by more than one process. In this case, the shared memory is explicitly requested from the kernel through a secure mechanism and both processes have to explicitly attach to it in order to use it.

The privileged mode is usually referred to as 'kernel' mode because the kernel is executed by the CPU running in this mode. In order to switch to kernel mode you have to issue a specific instruction (often called a *trap*) that switches the CPU to running in kernel mode *and runs code from a specific location held in a jump table.* For security reasons, you cannot switch to kernel mode and execute arbitrary code - the traps are managed through a table of addresses that cannot be written to unless the CPU is running in supervisor mode. You trap with an explicit trap number and the address is looked up in the jump table; the kernel has a finite number of controlled entry points.

The 'system' calls in the C library (particularly those described in Section 2 of the man pages) have a user-mode component, which is what you actually call from your C program. Behind the scenes, they may issue one or more system calls to the kernel to do specific services such as I/O, but they still also have code running in user-mode. It is also quite possible to directly issue a trap to kernel mode from any user space code if desired, although you may need to write a snippet of assembly language to set up the registers correctly for the call. A page describing the system calls provided by the Linux kernel and the conventions for setting up registers can be found here.

More about 'sys'

There are things that your code cannot do from user mode - things like allocating memory or accessing hardware (HDD, network, etc.). These are under the supervision of The Kernel, and it alone can do them. Some operations that you do (like `malloc` or`fread`/`fwrite`) will invoke these Kernel functions and that then will count as 'sys' time. Unfortunately it's not as simple as "every call to malloc will be counted in 'sys' time". The call to `malloc` will do some processing of its own (still counted in 'user' time) and then somewhere along the way it may call the function in kernel (counted in 'sys' time). After returning from the kernel call, there will be some more time in 'user' and then `malloc` will return to your code. As for when the switch happens, and how much of it is spent in kernel mode... you cannot say. It depends on the implementation of the library. Also, other seemingly innocent functions might also use `malloc` and the like in the background, which will again have some time in 'sys' then.