A process which is executing in kernel mode can not be preempted.

How is it possible?

Consider the case that a process is executing in kernel mode and taking a lot of time. Due to this the rest of processes will remain in ready queue.

For example how would the process respond to timer interrupt (if it is executing in kernel mode)? Or how would it respond to high temperature cut off interrupt.

First things first, your assumptions are incorrect: the Linux kernel does provide limited kernel-mode preemption, if the config options CONFIG\_PREEMPT\_VOLUNTARY or CONFIG\_PREEMPT are set.

With preemption turned on, tasks won't be preempted if they are holding locks; the kernel developers pay very close attention to how long processes hold locks, and try to reduce the amount of time that locks are held. (In part for preempt, in part because the longer locks are held, the less concurrent the system may be, if several processors are contending for the same lock. If the locks are held for short times, there is less contention and potentially higher throughput.)

Furthermore, the kernel developers try to limit the length of time a process may spend in kernel mode. After all, time spent in the kernel is time that isn't spent in the application, doing whatever it is the application does.

If the standard Linux kernel cannot provide you with good enough hard realtime performance handling interrupts, you can of course use a system such as [RTLinux](http://en.wikipedia.org/wiki/Rtlinux), for which [commercial support](http://www.novell.com/products/realtime/) is available.

I just reading an article which says:

Reasons to control the interrupt system generally boil down to needing to provide synchronization. By disabling interrupts, you can guarantee that an interrupt handler will not preempt your current code. Moreover, **disabling interrupts also disables kernel preemption**. Neither disabling **interrupt delivery** nor disabling **kernel preemption** provides any protection from concurrent access from another processor,however.

So I just wonder the difference between interrupt and kernel preemption.

Or could we say disabling kernel preemption also disables interrupts?

When a process is interrupted, the kernel runs some code, which may not be related to what the process does.  
When this is done, two things can happen:  
1. The same process will get the CPU again.  
2. A different process will get the CPU. The current process was preempted.

So preemption will only happen after an interrupt, but an interrupt doesn't always cause preemption.

Explained in your own words, what is preemption and what does it mean to a (linux) kernel?

What are advantages and disadvantages in having a preemptible kernel?

Preemptive multitasking - Running several processes/threads on a single processor, creating the illusion that they run concurrently when actually each is allocated small multiplexed time slices to run in. A process is "preempted" when it is scheduled out of execution and waits for the next time slice to run in.

A preemptive kernel is one that can be interrupted in the middle of a executing code - for instance in response for a system call - to do other things and run other threads, possible that are not in the kernel.  
The main advantage in a preemptive kernel is that sys-calls do not block the entire system. if a sys-call takes a long time to finish then it doesn't mean the kernel can't do anything else in this time.  
The main disadvantage is that this introduces more complexity to the kernel code, having to handle more end-cases, perform more fine grained locking or use lock-less structures and algorithms.

You should really use the term "preemptive." There are different kinds of preemption. Essentially, it is very simple and you probably understand this by another name. A preemptive operating system can switch contexts between user mode threads without any special programming in the preempted application. This allows for multitasking. An OS can switch away and back to a process and this switching is essentially trasnparent. There is also such a thing as preemptive kernel, which allows kernel mode threads to be preempted (most operating systems do not allow this but it is required for certain applications such as in real time systems).

With a non-preemptible kernel, on a single processor system it is possible for kernel developers to be lazy and get away without any locking most of the time - of course this is a big FAIL on SMP. Preemptible kernels allow them to get this pain without more cores.

Threads on different cores

Threads working on completely different data benefit from more separation. These threads would ideally be scheduled in separate NUMA domains and physical cores.

Threads working on the same data will benefit from cache locality, so the idea policy is to schedule them close together so they share cache.

Threads that work on the same data and experience a large amount of pipeline stalls benefit from sharing a hyperthread core. Each thread can run until it stalls, at which point the other thread can run. Threads that run without stalls are only hurt by hyperthreading and should be run on different cores.

Making the ideal scheduling decision relies on a lot of data collection and a lot of decision making. A large danger in OS design is to make the thread scheduling too smart. If the OS spends a lot of processor time trying to find the ideal place to run a thread, it's wasting time it could be using to run the thread.

So often it's more efficient to use a simplified thread scheduler and if needed, let the program specify its own policy. This is the thread affinity setting.

Is there a progamatic method to set CPU affinity for a process in c/c++ for the linux operating system.

For example, to run on CPUs 0 and 2 only:

#include <sched.h>

cpu\_set\_t mask;

CPU\_ZERO(&mask);

CPU\_SET(0, &mask);

CPU\_SET(2, &mask);

result = sched\_setaffinity(0, sizeof(mask), &mask);

Why One Needs CPU Affinity

The first benefit of CPU affinity is optimizing cache performance. I said the O(1) scheduler tries hard to keep tasks on the same processor, and it does. But in some performance-critical situations—perhaps a large database or a highly threaded Java server—it makes sense to enforce the affinity as a hard requirement. Multiprocessing computers go through a lot of trouble to keep the processor caches valid. Data can be kept in only one processor's cache at a time. Otherwise, the processor's cache may grow out of sync, leading to the question, who has the data that is the most up-to-date copy of the main memory? Consequently, whenever a processor adds a line of data to its local cache, all the other processors in the system also caching it must invalidate that data. This invalidation is costly and unpleasant. But the real problem comes into play when processes bounce between processors: they constantly cause cache invalidations, and the data they want is never in the cache when they need it. Thus, cache miss rates grow very large. CPU affinity protects against this and improves cache performance.

The third and final benefit is found in real-time or otherwise time-sensitive applications. In this approach, all the system processes are bound to a subset of the processors on the system. The specialized application then is bound to the remaining processors. Commonly, in a dual-processor system, the specialized application is bound to one processor, and all other processes are bound to the other. This ensures that the specialized application receives the full attention of the processor.

Suppose I have a multi-threaded application (say ~40 threads) running on a multiprocessor system (say 8 cores) with Linux as the operating system where different threads are more essentially LWP (Light Weight Processes) being scheduled by the kernel.

What would be benefits/drawbacks of using the CPU affinity? Whether CPU affinity is going to help by localizing the threads to a subset of cores thus minimizing cache sharing/misses?

The scheduler already tries to keep threads on the same cores, and to avoid migrations. This suggests that there's probably not a lot of mileage in managing thread affinity manually, unless:

* you can demonstrate that for some reason the kernel is doing a bad a job for your particular application; or
* there's some specific knowledge about your application that you can exploit to good effect.

For general-purpose applications, there is no reason to set the CPU affinity; you should just allow the OS scheduler to choose which CPU should run the process or thread. However, there are instances where it is necessary to set the CPU affinity. For example, in real-time systems where the cost of migrating a thread from one core to another (which can happen at any time if the CPU affinity has not been set) can introduce unpredictable delays that can cause tasks to miss their deadlines and which preclude real-time guarantees.

You can take a look at [this article](http://www.cs.wustl.edu/~lu/papers/rtcsa09-mc.pdf) about a [multi-core aware implementation of real-time CORBA](http://www.cs.wustl.edu/~lu/papers/rtcsa09-mc.pdf) that, among other things, had to set the CPU affinity so that CPU migration could not result in missed deadlines.

For applications designed with parallelism and multiple cores in mind, OS-default thread affinity is sometimes not enough. There are many approaches to parallelism, but so far all require involvement of the programmer and knowledge - at some level at least - of the architecture on which the solution will be mapped. This includes the machines, CPU's and threads that are involved.

Operating systems are usually oblivious to the details of modern multicore architecture. For example, say we have 2-socket quadcore processors, and the processor supports [SMT](http://en.wikipedia.org/wiki/Simultaneous_multithreading)(=HyperThreading). In this case, we have 2 processors, 8 cores, and 16 hardware threads. So, OS will see 16 logical processors. If an OS does not recognize such hierarchy, it is highly likely to loose some performance gains. The reasons are:

1. **Caches**: in our example, two different processors (installed on two different sockets) are not sharing any on-chip caches. Say an application has 4 busy-running threads and a lot of data are shared by threads. If an OS schedules the threads across the processors, then we may loose some cache locality, resulting in performance lose. However, the threads are not sharing much data (having distinct working set), then separating to different physical processors would be better by increasing effective cache capacity. Also, more tricky scenario could be happen, which is very hard for OS to aware of.
2. **Resource conflict**: let's consider SMT(=HyperThreading) case. SMT shares a lot of important resources of CPU such as caches, TLB, and execution units. Say there are only two busy threads. However, an OS may stupidly schedule these two threads on two logical processors from the same physical core. In such case, a significant resources are contended by two logical threads.

One good example is Windows 7. Windows 7 now supports a smart scheduling policy that consider SMT ([related article](http://www.tomshardware.com/news/windows-hyperthreading-intel-nehalem-atom,7831.html)). Windows 7 actually prevents the above 2. case. Here is a snapshot of task manger in Windows 7 with 20% load on Core i7 (quadcore with HyperThreading = 8 logical processors):

There is no doubt that context switching in kernel mode, which is trapped in by hardware interrupt or software interrupt. It is also known that context switching should be kept in atomic, but how do implement atomicity? It is known that interrupt gate disables all the interrupt (I don't know whether NMI is included).Does an interrupt gate itself can be seen as atomic sequence naturally?

Atomic operations are implemented in kernels as follows. At a high-level (e.g., from device driver developer's POV), the kernel provides [locks](http://lxr.linux.no/#linux+v3.3.4/include/linux/bit_spinlock.h) that are acquired and released similarly to user-space mutexes. At a lower-level, these locks are implemented using a combination of [atomic operations](http://en.wikipedia.org/wiki/Atomic_operations) and signaling the kernel scheduler that [preemption should not occur](http://lxr.linux.no/" \l "linux+v3.3.4/include/linux/preempt.h#L45).

In the scheduler itself, atomicity is guaranteed by [masking interrupts](http://lxr.linux.no/#linux+v3.3.4/arch/x86/kernel/entry_32.S#L355). This is done using a single instruction (cli or sti), so it is atomic by itself. NMI can indeed occur while interrupts are cleared, however, this is a special case. The NMI handler knows that it can be called in a weird context, so it makes sure that it does [not change the context](http://lxr.linux.no/#linux+v3.3.4/arch/x86/kernel/entry_32.S#L1309).

Context switch and mutual exclusion. I assume that by context switching you mean task switching. I must note that if be honest task switching not necessary need locking. The schedulling itself in almost any case require mutual exclusion on access of some data needed to make scheduling decision, but not context switch per se, that is follow the making scheduling decision.

Almost all modern operation system followed by design, where each thread in the system keep two stacks, one on the user mode side and another one in the kernel mode side. All that you need to make task switch is a three step action:

1. Save the state of the CPU (all registers) on the stack of the task that leave CPU.
2. Switch active stacks
   1. Save actual stack pointer in the current task descriptor (task that leaves CPU)
   2. Switch pointers to the current task descriptor
   3. Install new stack pointer into the stack pointer register from the current task descriptor (task that arrives to the CPU)
3. Restore state of the CPU from the stack (stack of the task that arrived to the CPU).

As you can see, by nature context switch operates with two set of data that are local to the task, but not global. Content of the CPU registers are not an example of shared data. Note! That if there is only one code in the kernel implement context switching all context switches will goes through the same code. And in itsturn this provide guaranty that if task was switched out, the top of kernel stack of that task will contain all CPU state in the well defined format. And every time when the task will be switched in, it will be able to found CPU state data on the top of its kernel stack!

There are two notes exist:

1. Remember, that almost all instructions on the CPU executes as atomic, if they not share data with another processors. That is mean that instruction execution can't be interrupted in the middle. Due to this, switch pointers to the current task descriptor can be safely made by xchg instruction with register-based operands.
2. Task switch can be interrupted by interruptions comming from the hardware. But interrupt handlers are task context independent. Due to this they will preempt task switching safely in regard to the tasks and in regard to itself through the CPU use usually use stack machine design approach.

Mutal exclusion in the kernel. Generally speaking, kenel can be devided into two parts: hardware-driven and software-driven. The fisrt one includes code that can be invoked by the interrupts triged by hardware devices (interrupt handlers) and that isn't rely to the currently executing thread and doesn't rely to ar somehow affect context of the curently executing task, and the second one include code that is explicitly or implicitly invokes by currently executing thread (system calls and exception handlers) and usually require access to the task descriptor datas.

This two kernel parts provides different requirenments for the data locking. The data that is used only by software-driven part of the kernel can use well known synchronization primitivs provided by the kernel environment. I mean the primitives that follow the check-and-wait approach. For example mutexes. If required data is locked task can register self in the wait queue and release CPU for another task.

The data that can be used only by hardware-driven part (only by one particular interrupt handler) can rely to the fact that next interrupt can't be delivered to the CPU if interrupt of the same time is on handling up to the momentwhen handler will notify interrupt controller that it complete interrupt handling (so called EOI (End Of Interrupt) notification). Due to this the data used only by one interrupt handler and that using located between interrupt handler execution start and sending of EOI notification protected in thenatural way and didn't require any additional locking.

And finally, data that is shared between software- and hardware-driven kernel parts or between different priority interrupt handlers provide most stricked requirenment to the mutal exclusion implementations. Such data can be protected neither check-wait locks nor by serial nature of same priority interrupt delivery. There are two main factors that imply to the locking requirenments for such sort of data:

1. Current activity executed on the CPU can be preempted by the interrupt handler in any point of execution in arbitrary unpredictable time. So in other words hardware interrupts handling is full asynchronous process.
2. Handlers can't wait resource release. Attempt to wait the resource release in interrupt handler can easily lead to the deadlock between interrupt handler waiting resource release and in the same moment blocking software-driven kernel part execution, and task owned the resours and blocked to execute as a part of software-driven kernel part to release resource.

Due to this in such cases next synchronization technics is used:

1. Use atomic operations. Note that in described context alnmost each CPU instruction can be considered as atomic. Usually by the term of atomic operation peaple means processor instruction prefixed by 'lock' prefix, but lock is needed only in the case of multiprocessor system, to prevent inconsistent physically parallel access to the same memory cell.
2. Use wait-free algoritms and data structures.
3. Use IST instead of ISR. This design approach assumes that the only work that must be done in the interrupt handler is schedule the Interrupt Handling Thread to run and notify it about interruption. Due to this on the one hand amount of the code running in the interrupt handler and amount of locks that it require is dramatically reduced. And on the other hand code moved from ISR to IST can use locking without any limitations.
4. Most general, most common and most popular method, that attacks one of the main factor of interrupt locking requirenments - preemption. Prevention of the preemption can be done by disabling interrupts acceptance. CPU usually supports some sort of methods to disable interrupts (for example x86 processors provide two special instructions - CLI (disable interrupts) and STI (enable interrups)). If CPU doesn't provide such facilities, interrupts usually can be disabled on side of interrupt controller too (I belive that it is a case for different RISC processors). Interrupt disabling means both that no one interrupt handler will preempt execution of the protected section of code, because processor can't receive signals from the interrupt controller and that thread swithing won't be done (at least implicitly), because timer that usually trig the scheduller won't be able to deliver interrupts as all other devices. This method of synchronization is certainly necessary for kernel implementation but is too brutal. Note! It achive synchronization by disabling ALL interrupts in the system, that negatively impact interrupt handling latencies and kernel responsiveness. Due to this kernel developers tries to make such form of critical sections as rare as possible and as short as possible.

In regard to the interrupt gate. Yes you are right. Intel processors automatically disable interrupts during entering into the interrupt gate and reenable them on iret from the handler. Due to this whole interrupt handler can be considered as executing in the atomic fashion. But! As I described on few lines above, people tries to minimize amount of code protected in such a way. As result, even if the OS kernel use interrupt gate instead of trap gate, it will try to reenable interrupts manually in the interrupt handler as quick as posible.

NMI is a very special case. It occurance usually mean that whole world collapsed. Is anybody will care on synchronization on the moment when all system is already going down?

I have a doubt when each process has it's own separate page table then why is there s system wide page table required ? Also if Page table is such that it maps virtual address to a physical address then I think two process may map to same physical address because all process have same virtual address space . Any good link on system wide page table will also solve my problem?

Each process has its own independent virtual address space - two processes can have virtpage 1 map to different physpages. Processes can participate in shared memory, in which case they each have some virtpage mapping to the same physpage.

The virtual address space of a process can be used to map virtpages to physpages, to memory mapped files, devices, etc. Virtpages don't have to be wired to RAM. A process could memory-map an entire 1GB file - in which case, its physical memory usage might only be a couple megs, but its virtual address space usage would be 1GB or more. Many processes could could do this, in which case the sum of virtual address space usage across all processes might be, say, 40 GB, while the total physical memory usage might be only, say, 100 megs; this is very easy to do on 32-bit systems.

Since lots of processes load the same libraries, the OS typically puts the libs in one set of read-only executable pages, and then loads mappings in the virtpage space for each process to point to that one set of pages, to save on physical memory.

Processes may have virtpage mappings that don't point to anything, for instance if part of the process's memory got written to the page file - the process will try to access that page, the CPU will trigger a page fault, the OS will see the page fault and handle it by suspending the process, reading the pages back into ram from the page file and then resuming the process.

There are typically 3 types of page faults. The first type is when the CPU does not have the virtual-physical mapping in the TLB - the processor invokes the pagefault software interrupt in the OS, the OS puts the mapping into the processor for that process, then the proc re-runs the offending instructions. These happen thousands of times a second.

The second type is when the OS has no mapping because, say, the memory for the process has been swapped to disk, as explained above. These happen infrequently on a lightly loaded machine, but happen more often as memory pressure is increased, up to 100s to 1000s of times per second, maybe even more.

The third type is when the OS has no mapping because the mapping does not exist - the process is trying to access memory that does not belong to it. This generates a segfault, and typically, the process is killed. These aren't supposed to happen often, and solely depend on how well written the software is on the machine, and does not have anything to do with scheduling or machine load.

When a process in the kernel space is holding a spin\_lock, the process cannot be preempted due to any of the following conditions :

1. When the time-slice of the process gets exhausted
2. When a high priority process becomes runnable
3. When an interrupt occurs

However the process can yield the processor if it blocks, sleeps, or explicitly call schedule(). Is my understanding correct?

Current implementations of spin locks use two entirely separate mechanisms to ensure mutual exclusion, one for dealing with inter-processor exclusion and one for dealing with the local processor threads and interrupt handlers.

* There is the spin\_lock itself which is only there to provide mutex between two or more processor cores. Any processor hitting a locked spin lock is basically stuck until another processor releases it. Spin locks serve no purpose on single processor systems - other than to increase the chance of total deadlock - so are usually removed at kernel compile time.
* To provide local processor mutex, spin\_lock() calls preempt\_disable() (on pre-emptive scheduling systems) to prevent any other thread from running whilst the lock is held; similarly spin\_lock\_irqsave() also does the equivalent of local\_irq\_save() to disable interrupts to prevent anything else at all running on the local processor.

As should be obvious from the above, using spin locks can gum up the whole machine so spin locks should just be used for very short periods of time and you should never do anything that might cause a reschedule whilst holding a lock.

The case with mutex\_lock is totally different - only threads attempting to access the lock are affected and if a thread hits a locked mutex then a reschedule will occur. For this reason mutex\_locks cannot be used in interrupt (or other atomic) contexts.