What is the basic differences between Semaphores & Spinlock? & In what best situations or conditions, we can use these.

very simply, a semaphore is a "yielding" synchronisation object, a spinlock is a 'busywait' one. (there's a little more to semaphores in that they synchronise several threads, unlike a mutex or guard or monitor or critical section that protects a code region from a single thread)

You'd use a semaphore in more circumstances, but use a spinlock where you are going to lock for a very short time - there is a cost to locking especially if you lock a lot. In such cases it can be more efficient to spinlock for a little while waiting for the protected resource to become unlocked. Obviously there is a performance hit if you spin for too long.

typically if you spin for longer than a thread quantum, then you should use a semaphore.

Over and above what Yoav Aviram and gbjbaanb said, the other key point used to be that you would never use a spin-lock on a single-CPU machine, whereas a semaphore would make sense on such a machine. Nowadays, you are frequently hard-pressed to find a machine without multiple cores, or hyperthreading, or equivalent, but in the circumstances that you have just a single CPU, you must use semaphores. (I trust the reason is obvious; if the single CPU is busy waiting for something else to release the spin-lock, but it is running on the only CPU, the lock is unlikely to be released.)

When should one use a spinlock instead of a mutex?

**The Theory**

In theory, when a thread tries to lock a mutex and it does not succeed, because the mutex is already locked, it will go to sleep, immediately allowing another thread to run. It will continue to sleep until being woken up, which will be the case once the mutex is being unlocked by whatever thread was holding the lock before. When a tread tries to lock a spinlock and it does not succeed, it will continuously re-try locking it, until it finally succeeds; thus it will not allow another thread to take its place (however, the operating system will forcefully switch to another thread, once the CPU runtime quantum of the current thread has been exceeded, of course).

**The Problem**

The problem with mutexes is that putting threads to sleep and waking them up again are both rather expensive operations, they'll need quite a lot of CPU instructions and thus also take some time. If now the mutex was only locked for a very short amount of time, the time spent in putting a thread to sleep and waking it up again might exceed the time the thread has actually slept by far and it might even exceed the time the thread would have wasted by constantly polling on a spinlock. On the other hand, polling on a spinlock will constantly waste CPU time and if the lock is held for a longer amount of time, this will waste a lot more CPU time and it would have been much better if the thread was sleeping instead.

**The Solution**

Using spinlocks on a single-core/single-CPU system makes usually no sense, since as long as the spinlock polling is blocking the only available CPU core, no other thread can run and since no other thread can run, the lock won't be unlocked either. IOW, a spinlock wastes only CPU time on those systems for no real benefit. If the thread was put to sleep instead, another thread could have ran at once, possibly unlocking the lock and then allowing the first thread to continue processing, once it woke up again.

On a multi-core/multi-CPU systems, with plenty of locks that are held for a very short amount of time only, the time wasted for constantly putting threads to sleep and waking them up again might decrease runtime performance noticeably. When using spinlocks instead, threads get the chance to take advantage of their full runtime quantum (always only blocking for a very short time period, but then immediately continue their work), leading to much higher processing throughput.

**The Practice**

Since very often programmers cannot know in advance if mutexes or spinlocks will be better (e.g. because the number of CPU cores of the target architecture is unknown), nor can operating systems know if a certain piece of code has been optimized for single-core or multi-core environments, most systems don't strictly distinguish between mutexes and spinlocks. In fact, most modern operating systems have hybrid mutexes and hybrid spinlocks. What does that actually mean?

A hybrid mutex behaves like a spinlock at first on a multi-core system. If a thread cannot lock the mutex, it won't be put to sleep immediately, since the mutex might get unlocked pretty soon, so instead the mutex will first behave exactly like a spinlock. Only if the lock has still not been obtained after a certain amount of time (or retries or any other measuring factor), the thread is really put to sleep. If the same system runs on a system with only a single core, the mutex will not spinlock, though, as, see above, that would not be beneficial.

A hybrid spinlock behaves like a normal spinlock at first, but to avoid wasting too much CPU time, it may have a back-off strategy. It will usually not put the thread to sleep (since you don't want that to happen when using a spinlock), but it may decide to stop the thread (either immediately or after a certain amount of time) and allow another thread to run, thus increasing chances that the spinlock is unlocked (a pure thread switch is usually less expensive than one that involves putting a thread to sleep and waking it up again later on, though not by far).

**Summary**

If in doubt, use mutexes, they are usually the better choice and most modern systems will allow them to spinlock for a very short amount of time, if this seems beneficial. Using spinlocks can sometimes improve performance, but only under certain conditions and the fact that you are in doubt rather tells me, that you are not working on any project currently where a spinlock might be beneficial. You might consider using your own "lock object", that can either use a spinlock or a mutex internally (e.g. this behavior could be configurable when creating such an object), initially use mutexes everywhere and if you think that using a spinlock somewhere might really help, give it a try and compare the results (e.g. using a profiler), but be sure to test both cases, a single-core and a multi-core system before you jump to conclusions (and possibly different operating systems, if your code will be cross-platform).

What exactly are spinlocks?

When you use regular locks (mutexes, critical sections etc), operating system puts your thread in the WAIT state and preempts it by scheduling other threads on the same core. This has a performance penalty if the wait time is really short, because your thread now has to wait for a preemption to receive CPU time again.

Besides, kernel objects are not available in every state of the kernel, such as in an interrupt handler or when paging is not available etc.

Spinlocks don't cause preemption but wait in a loop ("spin") till the other core releases the lock. This prevents the thread from losing it's quantum and continue as soon as the lock gets released. The simple mechanism of spinlocks allow a kernel to utilize it in almost any state.

That's why on a single core machine a spinlock is simply a "disable interrupts" or "raise IRQL" which prevents thread scheduling completely.

Spinlocks ultimately allow kernels to avoid "Big Kernel Lock"s (a lock acquired when core enters kernel and released at the exit) and have granular locking over kernel primitives, causing better multi-processing on multi-core machines thus better performance.

EDIT: A question came up: "Does that mean I should use spinlocks wherever possible?" and I'll try to answer it:

As I mentioned Spinlocks are only useful in places where anticipated waiting time is shorter than a quantum (read: milliseconds) and preemption doesn't make much sense (e.g. kernel objects aren't available).

If waiting time is unknown, or if you're in user mode Spinlocks aren't efficient. You consume 100% CPU time on the waiting core while checking if a spinlock is available. You prevent other threads from running on that core till your quantum expires. This scenario is only feasible for short bursts at kernel level and unlikely an option for a user-mode application.

Here is a question at SO addressing that: <http://stackoverflow.com/questions/1456225/spinlocks-how-much-useful-are-they>

How is thread sycnrhonisation implemented at the assembly level?

While I'm familiar with concurrent programming concepts such as mutexes and semaphores, I have never understood how they are implemented at the assembly language level.

I imagine there being a set of memory "flags" saying:

* lock A is held by thread 1
* lock B is held by thread 3
* lock C is not held by any thread
* etc

But how is access to these flags synchronized between threads? Something like this naive example would only create a race condition:

* In practice, these tend to be implemented with [CAS](http://en.wikipedia.org/wiki/Compare-and-swap) and [LL/SC](http://en.wikipedia.org/wiki/Load-Link/Store-Conditional). (...and some spinning before giving up the time slice of the thread - usually by calling into a kernel function that switches context.)
* If you only need a [spinlock](http://en.wikipedia.org/wiki/Spinlock), wikipedia gives you an example which trades CAS for lock prefixed xchgon x86/x64. So in a strict sense, a CAS is not needed for crafting a spinlock - but some kind of atomicity is still required. In this case, it makes use of an atomic operation that can write a register to memory and return the previous contents of that memory slot in a *single step*. (To clarify a bit more: the *lock* prefix asserts the #LOCK signal that ensures that the current CPU has exclusive access to the memory. On todays CPUs it is not necessarily carried out this way, but the effect is the same. By using xchg we make sure that we will not get preempted somewhere between reading and writing, since instructions will not be interrupted half-way. So if we had an imaginary *lock mov reg0, mem / lock mov mem, reg1* pair (which we don't), that would not quite be the same - it could be preempted just between the two movs.)
* On current architectures, as pointed out in the comments, you mostly end up using the atomic primitives of the CPU and the coherency protocols provided by the memory subsystem.
* For this reason, you not only have to use these primitives, but also account for the cache/memory coherency guaranteed by the architecture.
* There may be implementation nuances as well. Considering e.g. a spinlock:
  + instead of a naive implementation, you should probably use e.g. a [TTAS spin-lock with some exponential backoff](http://www.cs.brown.edu/courses/cs176/ch07.ppt),
  + on a Hyper-Threaded CPU, you should probably issue pause instructions that serve as hints that you're spinning - so that the core you are running on can do something useful during this
  + you should really give up on spinning and yield control to other threads after a while
  + etc...
* this is still user mode - if you are writing a kernel, you might have some other tools that you can use as well (since you are the one that schedules threads and handles/enables/disables interrupts).

The x86 architecture, has long had an instruction called xchg which will exchange the contents of a register with a memory location. xchg has always been atomic.

There has also always been a lock prefix that could be applied to ~~any~~ a single instruction to make that instruction atomic. Before there were multi processor systems, all this really did was to prevent an interrupt from being delivered in the middle of a locked instruction. (xchg was implicitly locked).

This article has some sample code using xchg to implement a spinlock<http://en.wikipedia.org/wiki/Spinlock>

When multi CPU and later multi Core systems began to be built, more sophisticated systems were needed to insure that lock and xchg would synchronize all of the memory subsystems, including l1 cache on all of the processors. About this time, new research into locking and lockless algorithms showed that atomic CompareAndSet was a more flexible primitive to have, so more modern CPUs have that as an instruction.

Addendum: In comments **andras** supplied a "dusty old" list of instructions which allow the lock prefix.<http://pdos.csail.mit.edu/6.828/2007/readings/i386/LOCK.htm>

Spinlock how much useful are they?

How often do you find yourself actually using spinlocks in your code? How common is it to come across a situation where using a busy loop actually outperforms the usage of locks?  
Personally, when I write some sort of code that requires thread safety, I tend to benchmark it with different synchronization primitives, and as far as it goes, it seems like using locks gives better performance than using spinlocks. No matter for how little time I actually hold the lock, the amount of contention I receive when using spinlocks is far greater than the amount I get from using locks (of course, I run my tests on a multiprocessor machine).

I realize that it's more likely to come across a spinlock in "low-level" code, but I'm interested to know whether you find it useful in even a more high-level kind of programming?

t depends on what you're doing. In general application code, you'll want to avoid spinlocks.

In low-level stuff where you'll only hold the lock for a couple of instructions, and latency is important, a spinlock mat be a better solution than a lock. But those cases are rare, especially in the kind of applications where C# is typically used.

For my realtime work, particularly with device drivers, I've used them a fair bit. It turns out that (when last I timed this) waiting for a sync object like a semaphore tied to a hardware interrupt chews up at least 20 microseconds, no matter how long it actually takes for the interrupt to occur. A single check of a memory-mapped hardware register, followed by a check to RDTSC (to allow for a time-out so you don't lock up the machine) is in the high nannosecond range (basicly down in the noise). For hardware-level handshaking that shouldn't take much time at all, it is really tough to beat a spinlock.

f your updates satisfy some access criteria then they are good spinlock candidates:

* **fast**, ie you will have time to acquire the spinlock, perform the updates and release the spinlock in a single thread quanta so that you don't get pre-empted while holding the spinlock
* **localized** all data you update are in preferably one single page that is already loaded, you do not want a TLB miss while you holding the spinlock, and you definetely don't want an page fault swap read!
* **atomic** you do not need any other lock to perform the operation, ie. never wait for locks under spinlock.

For anything that has any potential to yield, you should use a notified lock structure (events, mutex, semaphores etc).

Is there a way to make a function atomic in C

Maybe.

It depends entirely on your definition of "atomic".

* In a single core, deeply embedded environment without an operating system involved you can usually disable and enable interrupts. This can be used to allow a function to be atomic against interrupt handler code. But if you have a multi-master bus, a DMA engine, or some other hardware device that can write memory independently, then even masking interrupts might not provide a strong enough guarantee in some circumstances.
* In an RTOS (real time operating system) environment, the OS kernel usually provides low level synchronization primitives such as critical sections. A critical section is a block of code that behaves "essentially" atomically, at least with respect to all other critical sections. It is usually fundamental to the OS's implementation of other synchronization primitives.
* In a multi-core environment, a low level primitive called a spinlock is often available. It is used to guard against entry to a block of code that must be atomic with respect to other users of the same spinlock object, and operates by blocking the waiting CPU core in a tight loop until the lock is released (hence the name).
* In many threading environments, more complex primitives such as events, semaphores, mutexes, and queues are provided by the threading framework. These cooperate with the thread scheduler such that threads waiting for something to happen don't run at all until the condition is met. These can be used to make a function's actions atomic with respect to other threads sharing the same synchronization object.

A general rule would be to use the highest level capabilities available in your environment that are suited to the task. In the best case, an existing thread safe object such as a message queue can be used to avoid needing to do anything special in your code at all.

f you want to make sure your function won't be interrupted by signal, use sigprocmask() to mask and unmask signals, although some signals cannot be blocked (like SIGKILL) and behaviour for blocking some signals (like SIGSEGV) is undefined.

Why spinlocks are used in interrupt handlers?

Semaphores cause tasks to sleep on contention, which is unacceptable for interrupt handlers. Basically, for such a short and fast task (interrupt handling) the work carried out by the semaphore is overkill. Also, spinlocks can't be held by more than one task.

Example: your driver is executing and has just taken out a lock that controls access to its device. While the lock is held, the device issues an interrupt, which causes your interrupt handler to run. The interrupt handler, before accessing the device, must also obtain the lock. Taking out a spinlock in an interrupt handler is a legitimate thing to do; that is one of the reasons that spinlock operations do not sleep. But what happens if the interrupt routine executes in the same processor as the code that took out the lock originally? While the interrupt handler is spinning, the noninterrupt code will not be able to run to release the lock. That processor will spin forever.

Spinlocks on a uni processor

Short answer: no.

**According to**[**http://docsrv.sco.com/cgi-bin/man/man?Intro+3synch**](http://docsrv.sco.com/cgi-bin/man/man?Intro+3synch)

Spin locks must not be used on a single processor system. In the best case, a spin lock on a single processor system will waste resources, slowing down the owner of the lock; in the worst case, it will deadlock the processor.

**From:**[**http://blogs.microsoft.co.il/blogs/sasha/archive/2008/08/10/practical-concurrency-patterns-spinlock.aspx**](http://blogs.microsoft.co.il/blogs/sasha/archive/2008/08/10/practical-concurrency-patterns-spinlock.aspx)

On single-processor systems, spinlocks are not needed because spinlock synchronization is required on high IRQLs only. On high IRQLs (above dispatch IRQL) a context switch cannot occur, so instead of spinning the acquiring thread can simply request an interrupt on the relevant IRQL and return; the interrupt will be masked until the releasing thread lowers the IRQL below the requested IRQL.

For single processor systems, the kernel will ignore the spin count value, and treat it as zero - essentially making a spinlock a no-op.

Yes, spin locks can be useful, and improve efficiency of some operations. However, generally you should start with a mutex, and if profiling show it to be a bottleneck, you may want to consider a spinlock.

Your observation is good: on a uniprocessor system, there is no point in spinning to wait for a resource, because you will may as well switch threads sooner rather than later. Mutexes and semaphores do exactly this.

On a multiprocessor system, a thread on another processor may release the lock without you context-switching. Spinlocks can be useful, then, if you don't expect to be waiting long, because it may be faster just to hang around until the other thread unlocks the thing. If you go to sleep on a mutex, you're basically assured some significant dead time before you will get rescheduled.

In kernel code, however, the situation changes: Interrupt handlers need to access shared resources with the rest of the kernel, but they cannot sleep. Mutexes will put the kernel to sleep, so you can't use them, but spinlocks aren't useful either because nothing will interrupt an interrupt handler on a uniprocessor (well, maybe another interrupt, but that's scary).

In a kernel, then, spinlocks *within* an interrupt handler compile to a no-op. They are completely elided, just like you might think. At the same time, to prevent races, spinlocks *in the rest of the kernel* disable interrupts just before they actually spin on something (because kernel tasks can be scheduled). These codes only need spinlocks (as opposed to mutexes) if they share code with an interrupt handler.

In general, you're right: **spinlocks really don't make much sense on a uniprocessor *if you have mutexes*, because mutexes waste less time.**

Safety nets in multithreaded code

There are a number of things we do in our product (a hypervisor designed to help you find concurrency bugs in applications) that are more generally useful. Note that we do these in our code itself (because its a highly concurrent piece of software) and that some of these are useful whether or not you are writing concurrent code.

* Like you, we have the ability to assert(lock\_held(...)) and use it.
* We also (because we have our own scheduler) can assert(single\_threaded()) for those (rare) situations where we count on no other thread being active in the system.
* Memory corruption from one thread to another is pretty common (and hard to debug) so we do two things to address this: sprinkled throughout our thread stack are some magic cookies. We periodically (in our get\_thread\_id()) function invoke a "validate\_thread\_stack()" function that checks these cookies to make sure the stack is not corrupted.
* Our malloc sticks magic cookies before and after a malloc block of memory and checks these on free. If anyone overruns their data these can be used to find the corruption early.
* On free() we blast a well known pattern (in our case 0xdddd...) over the memory. This nicely corrupts anyone else who had a dangling pointer left over to that memory region.
* We have a guard page (a memory page not mapped into the address space) near the bottom of the thread stack. If the thread overruns its stack, we catch it via page fault and drop into our debugger.
* Our locks are witnessed. Checkout the FreeBSD lock witness code. Its like that but homebrew. Basically the witness code is a lightweight way of detecting potential deadlocks by looking at cycles in the lock acquisition graph.
* Our locks are also wrapped with accessors that record the file/line number of acquisition and release. For double unlocks or double locks, you get pretty debug information on your screwup.
* Our locks are also profiled. Once you get your code working you want it working well. We track the usual things like how many acquisitions, how long it took to acquire it.
* In our system, we have an expectation that locks are not contended (we carefully designed the code this way). So if you wait for a spin lock longer than a second or two in our system you get dropped into the debugger because its most likely not a good thing.
* Our variables that are meant to be updated atomically are wrapped inside of C struct's. The reason for this is to prevent sloppy code where you mix good use: atomic\_increment(&var); and bad use var++. We make it very hard to write the latter code.
* "volatile" is forbidden in our code base because its ambiguously implemented by compilers. Its a bad way to try and cobble together synchronization.
* And of course code reviews. If you can't explain your concurrency assumptions and locking discipline to a colleague, then there's definitely issues with the code :-)

Splinlock on non premetive kernels

I read that on a system with 1 CPU and non preemtive linux kernel (2.6.x) a spin\_lock call is equivalent to an empty call, and thus implemented that way.

I can't understand that: shouldn't it be equivalent to a sleep on a mutex? Even on non-preemtive kernels interrupt handlers may still be executed for example or I might call a function that would put the original thread to sleep. So it's not true that an empty spin\_lock call is "safe" as it would be if it was implemented as a mutex.

Is there something I don't get?

If a nonpreemptive uniprocessor system ever went into a spin on a lock, it would spin forever; no other thread would ever be able to obtain the CPU to release the lock (because it couldn't yield). Because of this, spinlock operations on uniprocessor systems without preemption enabled are optimized to do nothing, with the exception of the ones that change the IRQ masking status (in Linux, that would bespin\_lock\_irqsave()). Because of preemption, even if you never expect your code to run on an SMP system, you still need to implement proper locking.

If you're interested in a spinlock that can be taken by code running in interrupt context (hardware or software), you must use a form of spin\_lock\_\* that disables interrupts. Not doing so will deadlock the system as soon as an interrupt arrives while you have entered your critical section.

f you were to use spin\_lock() on a non-preemptive kernel to shield data against an interrupt handler, you'd deadlock (on a single-processor machine).

If the interrupt handler runs while other kernel code holds the lock, it will spin forever, as there is no way for the regular kernel code to resume and release the lock.

Spinlocks can only be used if the lock holder can always run to completion.

The solution for a lock that might be wanted by an interrupt handler is to use spin\_lock\_irqsave(), which disables interrupts while the spinlock is held. With 1 cpu, no interrupt handler can run, so there will not be a deadlock. On smp, an interrupt handler might start spinning on another cpu, but since the cpu holding the lock can't be interrupted, the lock will eventually be released.

What is a uninterruptable process

Sometimes whenever I write a program in Linux and it crashes due to a bug of some sort, it will become an uninterruptable process and continue running forever until I restart my computer (even if I log out). My questions are:

* What causes a process to become uninterruptable?
* How do I stop that from happening?
* This is probably a dumb question, but is there any way to interrupt it without restarting my computer?

An uninterruptable process is a process which happens to be in a system call (kernel function) that cannot be interrupted by a signal.

To understand what that means, you need to understand the concept of an interruptable system call. The classic example is read(). This is a system call that can take a long time (seconds) since it can potentially involve spinning up a hard drive, or moving heads. During most of this time, the process will be sleeping, blocking on the hardware.

While the process is sleeping in the system call, it can receive a unix asynchronous signal (say, SIGTERM), then the following happens:

* The system calls exits prematurely, and is set up to return -EAGAIN to userspace.
* The signal handler is executed.
* If the process is still running, it gets the return value from the system call, and if it is written correctly it will make the same call again.

The crux of the issue is that (for some reason I do not really understand), the execution needs to get out of the system call for the userspace signal handler to run.

On the other hand, some system calls are not allowed to be interrupted in this way. If the system calls stalls for some reason, the process can indefinitely remains in this unkillable state.

LWN ran a [nice article](http://lwn.net/Articles/288056/) that touched this topic in July.

To answer the original question:

* How to prevent this from happening: figure out which driver is causing you trouble, and either stop using, or become a kernel hacker and fix it.
* How to kill an uninterruptible process without rebooting: somehow make the system call terminate. Frequently the most effective manner to do this without hitting the power switch is to pull the power chord. You can also become a kernel hacker and make the driver use TASK\_KILLABLE, as explained in the LWN article.

When a process is on user mode, it can be interrupted at any time (switching to kernel mode). When the kernel returns to user mode, it checks if there are any signals pending (including the ones which are used to kill the process, such as SIGTERM and SIGKILL). This means a process can be killed only on return to user mode.

The reason a process cannot be killed in kernel mode is that it could potentially corrupt the kernel structures used by all the other processes in the same machine (the same way killing a thread can potentially corrupt data structures used by other threads in the same process).

When the kernel needs to do something which could take a long time (waiting on a pipe written by another process or waiting for the hardware to do something, for instance), it sleeps by marking itself as sleeping and calling the scheduler to switch to another process (if there is no non-sleeping process, it switches to a "dummy" process which tells the cpu to slow down a bit and sits in a loop — the idle loop).

If a signal is sent to a sleeping process, it has to be woken up before it will return to user space and thus process the pending signal. Here we have the difference between the two main types of sleep:

* TASK\_INTERRUPTIBLE, the interruptible sleep. If a task is marked with this flag, it is sleeping, but can be woken by signals. This means the code which marked the task as sleeping is expecting a possible signal, and after it wakes up will check for it and return from the system call. After the signal is handled, the system call can potentially be automatically restarted (and I won't go into details on how that works).
* TASK\_UNINTERRUPTIBLE, the uninterruptible sleep. If a task is marked with this flag, it is not expecting to be woken up by anything other than whatever it is waiting for, either because it cannot easily be restarted, or because programs are expecting the system call to be atomic. This can also be used for sleeps known to be very short.

TASK\_KILLABLE (mentioned in the LWN article linked to by ddaa's answer) is a new variant.

This answers your first question. As to your second question: you can't avoid uninterruptible sleeps, they are a normal thing (it happens, for instance, every time a process reads/writes from/to the disk); however, they should last only a fraction of a second. If they last much longer, it usually means a hardware problem (or a device driver problem, which looks the same to the kernel), where the device driver is waiting for the hardware to do something which will never happen. It can also mean you are using NFS and the NFS server is down (it is waiting for the server to recover; you can also use the "intr" option to avoid the problem).

Finally, the reason you cannot recover is the same reason the kernel waits until return to user mode to deliver a signal or kill the process: it would potentially corrupt the kernel's data structures (code waiting on an interruptible sleep can receive an error which tells it to return to user space, where the process can be killed; code waiting on an uninterruptible sleep is not expecting any error).

How does an OS generally go about manageing kernel memory and page handling?

A good starting point for all these questions is to look at how Unix does it. As a famous quote says, "Those who don't understand UNIX are doomed to reinvent it, poorly."

First, about calling kernel functions. It is not enough to simply have the functions somewhere a program can call, since the program is most probably running in "user mode" (ring 3 on IA-32) and the kernel has to run in "kernel mode" (usually ring 0 on IA-32) to do its priviledged operations. You have to somehow do the transition between both modes, and this is very architecture specific.

On IA-32, the traditional way is to use a gate in the IDT together with a software interrupt (Linux uses int 0x80). Newer processors have other (faster) ways to do it, and which ones are available depends on whether the CPU is from AMD or Intel, and on the specific CPU model. To accomodate this variation, recent Linux kernels use a page of code mapped by the kernel at the top of the address space for every process. So, on recent Linux, to do a system call you call a function on this page, which will in turn do whatever is needed to switch to kernel mode (the kernel has more than one copy of that page, and choses which copy to use on boot depending on your processor's features).

Now, the memory management. This is a **huge** subject; you could write a large book about it and not exaust the subject.

Be sure to keep in mind that there are at least *two* views of the memory: the physical view (the real order of the pages, visible to the hardware memory subsystem and often to external peripherals) and the logical view (the order of the pages seen by programs running on the CPU). It's quite easy to confuse both. You will be allocating *physical* pages and assigning them to *logical* addresses on the program or kernel address space. A single physical page can have several logical addresses, and can be mapped to different logical addresses in different processes.

The kernel memory (reserved for the kernel) is usually mapped at the top of the address space of every process. However, it is set up so it can only be acessed on kernel mode. There is no need for fancy tricks to hide that portion of memory; the hardware does all the work of blocking the access (on IA-32, it is done via page flags or segment limits).

The programs allocate memory on the rest of the address space in several ways:

* Part of the memory is allocated by the kernel's program loader. This includes the program code (or "text"), the program initialized data ("data"), the program uninitialized data ("bss", zero-filled), the stack, and several odds and ends. How much to allocate, where, what should be the initial contents, which protection flags to use, and several other things, are read from the headers on the executable file to be loaded.
* Traditionally on Unix, there is an area of memory which can grow and shrink (its upper limit can be changed via the brk() system call). This is traditionally used by the heap (the memory allocator on the C library, of which malloc() is one of the interfaces, is responsible for the heap).
* You often can ask the kernel to map a file to an area of address space. Reads and writes to that area are (via paging magic) directed to the backing file. This is usually called mmap(). With an anonymous mmap, you can allocate new areas of the address space which are not backed by any file, but otherwise act the same way. The kernel's program loader will often use mmap to allocate parts of the program code (for instance, the program code can be backed by the executable itself).

Acessing areas of the address space which are not allocated in any way (or are reserved for the kernel) is considered an error, and on Unix will cause a signal to be sent to the program.

The compiler either allocates memory statically (by specifying it on the executable file headers; the kernel's program loader will allocate the memory when loading the program) or dynamically (by calling a function on the language's standard library, which usually then calls a function in the C language standard library, which then calls the kernel to allocate memory and subdivides it if necessary).

I've seen several question on here about [exceptions](http://stackoverflow.com/questions/tagged?tagnames=exceptions&sort=votes&pagesize=50), and some of them hint at [interrupts as exceptions](http://stackoverflow.com/search?s=interrupt+exception), but none make the connection clear.

* What is an interrupt?
* What is an exception? (please explain what exceptions are for each language you know, as there are some differences)
* When is an exception an interrupt and vice-versa?

An interupt is a CPU signal generated by hardware, or specific CPU instructions. These cause interupt handlers to be executed. Things such as I/O signals from I/O hardware generate interupts.

An exception can be thought of as a software-version of an interupt, that only affects its process.

I'm not sure on the exact details, but an exception *could* be implemented by an interupt.

Your processor is going to have a number of external interrupt pins. Typically these pins are connected to hardware and are used to indicate when some external event occurs. For example, if you are using a serial port the UART will raise raise a pin that is connected to one of the interrupt pins on the processor to indicate that a byte has been received.

Other peripherals like timers, usb controllers, etc. will also generate interrupts on the basis of some external event.

When the processor receives a signal on one of it's external interrupt pins it will immediately jump to some nominated location in memory and start executing. The code executed is typically called an ISR, or interrupt service routine. Unless you're implementing drivers or doing embedded software of some sort it's unlikely that you'll ever come across ISRs.

Unfortunately the answer to the question about exceptions is a little less clear - there have been 3 different meanings listed in other answers on this page.

Ron Savage's answer refers to the software construct. This is purely an application level exception, where a piece of code is able to indicate an error that can be detected by some other piece of code. There is no hardware involvement here at all.

Then there is the exception as seen by a task. This is an operating system level construct that is used to kill a task when it does something illegal - like divide by 0, illegally accessing memory etc.

And thirdly, there is the hardware exception. In terms of behaviour it is identical to an interrupt in that the processor will immediately jump to some nominated memory location and start executing. Where an exception differs from an interrupt is that an exception is caused by some illegal activity that the processor has detected. For example the MMU on the processor will detect illegal memory accesses and cause an exception. These hardware exceptions are the initial trigger for the operating system to perform it's cleanup tasks (as described in the paragraph above).