System Calls

http://wiki.osdev.org/System_Calls

System Calls are used to call a kernel service from user land. The goal is to be able to switch from user mode to kernel mode, with the associated privileges. Provided system calls depend on the nature of your kernel.

Possible methods to make a System Call

Interrupts

The most common way to implement system calls is using a software interrupt. It is probably the most portable way to implement system calls. Linux traditionally uses interrupt 0x80 for this purpose.
To do this, you will have to create your interrupt handler in Assembly. This is because several common compilers do not support interrupt handlers, and most of the C compilers that do support interrupts make the interrupt function of the form

```
IntHandler:
    PUSHAD
    ; Code
    POPAD
    IRETD
```

Which will give you problems when you want to return values to the user. One method to fix this is by writing a handler that simply calls another function, so that registers are not preserved.

```
IntHandler:
    CALL C_Function
    IRETD
```

Many protected mode OSes use EAX to hold the function code. DOS uses the AX register to store the function code — AH for the service and AL for functions of the service, or AH for the functions if there are no services. For example, let's say you have read() and write(). The codes are 1 for read() and 2 for write() from the interrupt 0A9h (an arbitrary choice, possibly wrong). You can write

```
IntA9Handler:
    CMP AH, 1
    JNE .write
    CALL _read
    JMP .done
.write:
    CMP AH, 2
    JNE .badcode
    CALL _write
    JMP .done
.badcode:
    MOV EAX, 0FFFFFFFFh
.done:
    IRETD
```

More parameters are usually passed through other registers such as EBX, ESI, etc.

Sysenter/Sysexit (Intel)

Main article: Sysenter

On Intel CPU, starting from the Pentium II, a new instruction pair sysenter/sysexit has appeared. It allows a faster switch from user mode to kernel mode, by limiting the overhead of changing mode.
A similar instruction pair has been created by AMD: Syscall/Sysret. However the behaviour of these instructions are different from Intel's.

Trap

Some OSes implement system calls by triggering a CPU Trap in a determined fashion such that they can recognize it as a system call. This solution is adopted on some hardware by Solaris, by L4, and probably others.
For example, L4 use a "LOCK NOP" instruction on x86. Since it is not permitted to perfrom a lock on the "NOP" instruction a trap is triggered. The problem with this approach is that there is no guarantee the "LOCK NOP" will have the same behavior on future x86 CPU. They should probably have used the "UD2" instruction, since it is defined for this purpose.

Call Gates (Intel)

The 80386 family of processors offer various call gates as part of the GDT. The call gate is a far pointer that can be called similar to calling a normal function. Very few operating systems use call gates.

Passing Arguments

Registers

The easiest way to pass arguments to a System Call handler are the registers. The BIOS takes arguments this way.

Pros:

- very fast

Cons:

- limited to the number of available registers
- caller has to save/restore the used registers if it needs their old values after the System Call

Stack

It is also possible to pass arguments through the stack.

Pros:

- nested System Calls are possible
- it is easy to implement a System Call handler in C because C uses the stack to pass arguments to functions, too
- not limited

Cons:

- insecure (if the caller passes more/less arguments than the callee asumes to get)

Memory

The last common way to pass arguments is to store them in memory. Before making the System Call the caller must store a pointer to the argument's location in a register for the System Call handler. (assuming this location is not fixed)

Pros:

- not limited
- secure

Cons:

- one register is still needed
- nested System Calls are not possible without copying arguments

On the user land side

While the developper can trigger a system call manually, it is probably a good idea to provide a library to encapsulate such call. Therefore you will be able to switch the system call technique without impacting user applications.
Another way is to have a stub somewhere in memory that the kernel places there, then once your registers are set up, call that stub to do the actual system call for you. Then you can swap methods at load time rather than compile time.

Strategies Conlusion

The system call strategy depends on the platform. You may want to use different strategy depending on the architecture, and even switch strategy depending on the hardware performance. You might also need more parameter copying on a microkernel than you will need on a monolithic one.