The OS controls the hardware of the computer, so no matter how hard you may try, you probably cannot make an OS-independent program without it just turning into OS itself. The OS protects the system from danger by layering itself, and the application needs to communicate with it in order to get the necessary resources and data to run properly. They have to make system calls to get access to the hardware to run code, but the way to access the hardware on different OS's is also different. That means you cannot just create universal code to work on all devices. The only way to do that is to use something like JDK, and that is just because you're adding another layer between you and the application to talk to the OS for you.

The first thing I would want to check is whether the code has pieces that can easily be split up. This would be the case if there were little to no variables that need to be passed from one section to the next in a long sequence of events. Additionally, it is not very useful on a linear, predictable set of events, so I would look for code that attempts to solve an algorithm or generate a value. Although, if an aforementioned linear, predictable set of code were to need constant repetition, then it could be a useful thing to multithread. It really all depends on the code I'm looking at, what it is supposed to accomplish, and how much speedup is needed and/or can be provided.

Processes are big, clunky clusters of code, and they have a very difficult time properly communicating with each other. It has to go through a whole bunch of metaphorical red tape, setting up mailboxes and the like. Threads are much smaller, more manageable bites (not bytes) of code, and therefore don't require the setup and effort to communicate. Multithreading is a good way to speed up code and help different parts work concurrently, or even in parallel. Multiprocessing isn't impossible, but it is much more difficult and has a smaller set of useful instances than multithreading.

The biggest thing to worry about when designing for mobile devices is to make the programs run as efficiently as possible. This isn't because the phones are not powerful enough, but because the battery isn't large enough. Running out of battery on a mobile device is a very unpleasant experience for the consumer, and if your application is the thing causing it, it will not be very popular.

For things like embedded systems, you need to ensure your code is as bug-free as possible and is equally efficient. These devices do not have the ability to be updated most of the time, and they most likely run on very minimal hardware. There is little room for error in a program for an embedded system.

The options are shared memory and message passing. Shared memory offers much better performance as it skips all the steps necessary in message passing. However, it also can be more difficult to implement, as it has to worry about not stepping on the memory of the other process. Each have their ups and downs, but shared memory will offer faster performance, at the cost of all the memory being lumped together (which is not always desirable).