Things you may need to consider in Creating an Operating System

1. **Decide what medium you want to load your OS on**. It can be a CD, DVD, Thumb drive or a hard disk
2. **Decide what you want your OS to do**. Whether it is a fully capable OS with a GUI or something a bit more minimalistic, you'll need to know what direction you are taking it before beginning.
3. **Target what processor platform your operating system will support**. If you are not sure, your best bet is to target the X86 (32 bit) processor platform as most computers use X86 platform processors.
4. **Decide if you would rather do it all yourself from the ground up, or if there is an existing kernel you would like to build on top of**. Linux from Scratch is a project for those that would like to build their own Linux distro.
5. **Decide if you're going to create your own bootloader or a pre-created one such as GRUB**. While coding your own bootloader will give a lot of knowledge of the hardware and the BIOS, it may set you back on the programming of the actual kernel.
6. **While it is possible to create an operating system in a language such as Pascal or BASIC you will be better off using C or Assembly**. Assembly is absolutely necessary as some vital parts of an operating system require it. C++ contains keywords that need another fully built OS to run. Don't use it.
7. **Start small**. Begin with small things such as displaying text and interrupts before moving on to things such as memory management and multitasking.
8. **Decide on your design**. There are monolithic kernels and microkernels. Monolithic kernels implement all the services in the kernel, while microkernels have a small kernel combined with user daemons implementing services. In general, monolithic kernels are faster, but micro kernels have better fault isolation and reliability.
9. **After all development is done, decide if you want to release the code as Open source, or proprietary**

Computer System Architecture

1. **Single processor**
2. **Multi-processor**

Operating System Services

Operating System Structure

Safe Operating System

1. **Protect own data: Design components in a way that they allow other components access to their private data only in a way user intended.**
2. **Protect other systems: Don't allow components to connect to other computers without permission.**
3. **Protect privacy: Don't allow components to notify their presence to other computers without permission.**

Input

Input is divided to three categories:

1. **Trusted: Input from physical devices (eg. keyboard and mouse) is fully trusted. Interfaces doing potentially harmful actions should be allowed to be executed only via trusted input.**
2. **Non-automated: Input originated from physical device. It may have gone through multiple components to get here, but it couldn't have been done automatically. This is because a single physical event can be sent only once to each component as non-automated input. The component can get the time when the event occured, so it may also check that the event hadn't been deliberately delayed for too long.**
3. **Untrusted: The input may have come from anywhere.**

User Interface

Windows are constructed from elements. Elements are owned by components. One window can contain elements from multiple components. Only the element owner can "see" what is inside element, others can manipulate the element only via component interfaces.

Operating system prevents components from hiding other components' elements by moving them or placing other elements on top of them.

Only the component having focus sees keyboard or mouse events. Components can however register keyboard shortcuts which can be triggered from any component. Keyboard shortcuts cannot be overridden.

Some problems and solutions:

- User grants access without noticing what actually was granted
  - Don't use "yes/no" dialogs to grant access. Ever.
  - Avoid popping up security related dialogs. If user wants to give special access to something, user needs to request it explicitly.
- User grants access simply to stop it from constantly asking about it
  - Don't let components request privileges. They must be able to run without it. User can give it the privileges later if he wants to.
  - Preferrably components shouldn't be able to know if they have some privilege.
- Programs may hide themselves to run in background to perform some processing and send results over network without user's knowledge.
  - Show program in "taskbar" as long as it exists, even if it has no visible windows (like Mac OS X does). For programs that are supposed to be run in background, user can hide them manually.
  - Mark program in some way if it accesses network in it's life time. For example a small icon in it's "taskbar" entry and in it's title bar. Clicking it could show a history of where the program has connected and how much data it has transferred.

  While these don't prevent such programs, they make it much easier for user to detect them.

Shared Files

Normally components cannot open files not created by themselves. This is fine for configuration and private data, but not if user wants to share the file between multiple components. Typically this is done with *Open file* and *Save file* operations.

Shared files component allows user to give untrusted component access to files matching a given search criteria - usually one or more selected files. The access is granted by giving the untrusted component a token which can be used to access the file until the token is invalidated. The token is invalidated when:

- The component requests it by itself (ie. the file is closed)
- The component dies (ie. program is closed)
- User requests it

The token can be permanently saved as well, either internally for the component or attached into saved shared file as metadata. If it's attached with the file, then any component opening the file gets access to the tokens.

Permanent tokens could be implemented by making shared files component sign it with a cryptographic signature. This would allow storing the tokens in untrusted storage.

Connecting to Other Computers

Programs doing connections could be divided to three categories:

1. Clients connecting to servers, where the server name is written by user at least once in program's lifetime.
2. Programs connecting to a predefined set of servers.
3. Programs starting connections with 1) or 2), but depending on input they may connect to more computers.

First one is simple to solve. We'll have a *connect element* where user can type the server(s) where to connect to. The element returns the caller component an authorization token which can be used to connect to those specified servers.

This works very similiar to opening files. The authorization token can be saved with configuration file, so once user types the server name the first time in initial configuration, he also grants the program access to the server for as long as the configuration file exists.

Second and third ones are tricky. We would rather not allow spyware to announce it's presence to outside world, or run a DDoS client flooding some poor computers. But we would rather not want to specifically grant access to every program wanting to connect outside.

This should be configurable, but default should probably be to allow *toplevel components* to connect outside (and pass that privilege to it's child components if needed). Creating toplevel components requires special privileges. Typically you'd create them only using system's built-in components such as when you doubleclick a program in desktop or file browser.

Also because of spamming no-one would be able to connect to SMTP ports automatically (but see mail client example below).

Types of user interfaces

1. Command Line
2. Batch
3. GUI