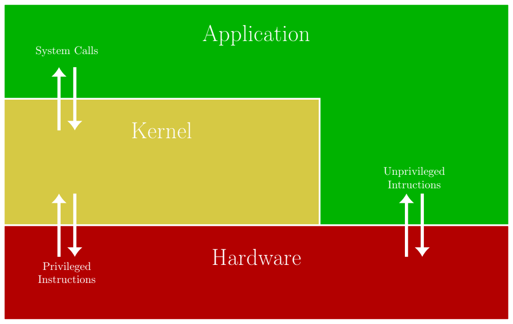
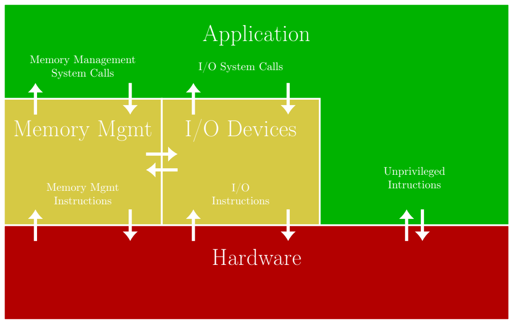
The standard design for an operating system can be viewed as follows:



(a) Course granularity view



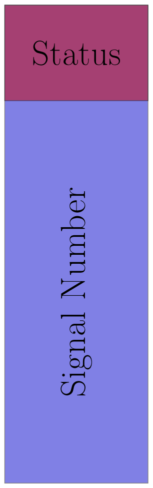
(b) Fine granularity view

We can see that our model has three general levels of instruction privilege:

- risky code with direct access to resources (OS only code) such as inb, outb
- code that accesses resources directly some of the time (iffy code) such as movl (movl is privileged if memory section to be accessed is in kernel)
- code with no access to code (unprivileged)

Untrusted modules can use code which does not have direct access; i.e. unprivileged code and (depending on context) iffy code.

To allow for these restrictions on an untrusted module, the system boots in privileged mode, sets the environment for the untrusted code, and jumps to the untrusted code in unprivileged mode. However, if the untrusted application desires access to system resources, we send an *interrupt* with a *supervisor call* instruction.

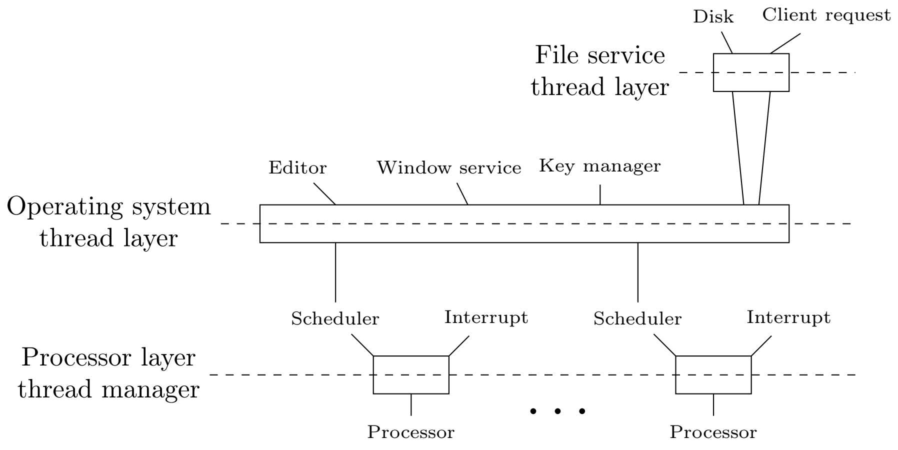


If the error code matches the error code corresponding to a given operation, the privileged instruction is run. The *trap* signal jumps to the kernel and raises the permission. The *return from trap* signal jumps to the user and lowers the permission. Trap locations are defined by *trap handlers*. These are held in a *trap table* in the kernel stack. The key for this hash table is the *system call number*, commonly known as the *interrupt number*.

On the hardware side, when a trap occurs:

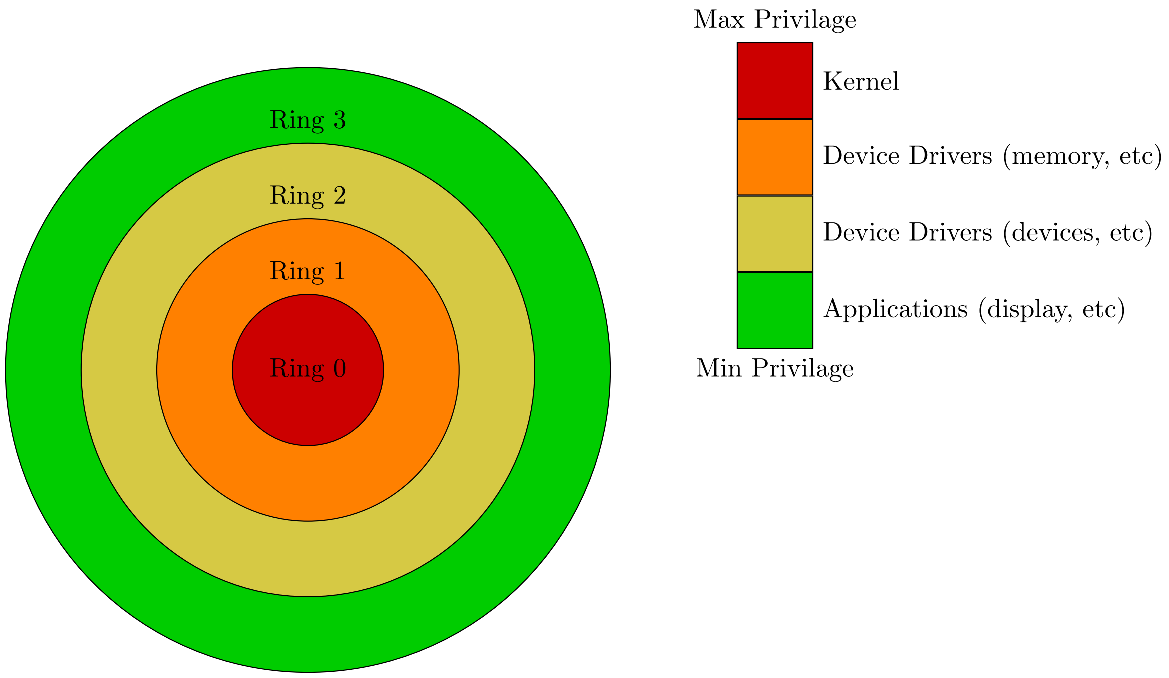
1. Running program information will be pushed onto the stack.
2. **eip** will be set to interrupt service routine.
3. Stack info will be popped from the stack when **reti** is called.

For interrupts, we can think of a processor as a hard-wired thread manager with two threads, the processor thread that runs the scheduler and the interrupt thread running in kernel mode. Threads with decreased privelage can be built on top of each other like so:



We define functions that give access to privileged instructions as *system calls*. This is as opposed to a *procedure call*, which is a function implemented on the stack that can call other functions. x86-64 Linux system calls use the convention of %rax for the interrupt number and %rdi, %rsi, %rdx, %r10, %r8, %r9, in that order, for arguments.

System calls are an example of the *protected transfer of control*, wherein the instruction pointer is set to a location known to be safe. On x86-64, there are 2 privilege bits corresponding to 4 levels of privilege that require different amounts of protection for control transfer. These can be visualized as follows:


Ring 3
Ring 2
Ring 1
Ring 0
Max Privilage
Kernel
Device Drivers (memory, etc)
Device Drivers (devices, etc)
Applications (display, etc)
Min Privilage