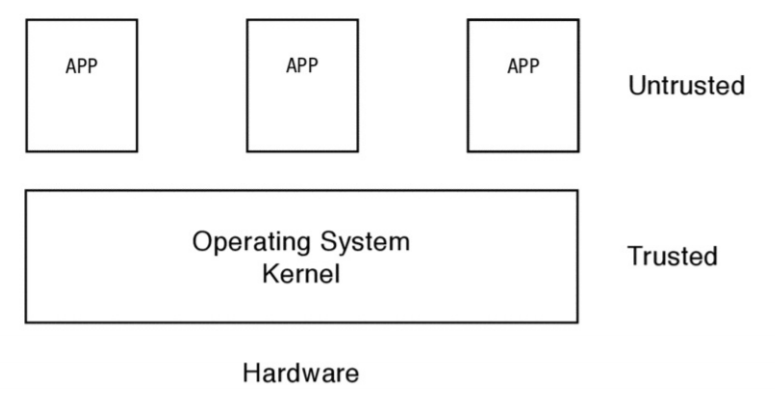
Chapter 2. The Kernel Abstraction

«good fences make good neighbors»

As discussed in the last chapter protection is a central and important role in an OS. It is essential to achieving several of the OS goals: ***Reliability, Security, Privacy and Fair resource allocation.***

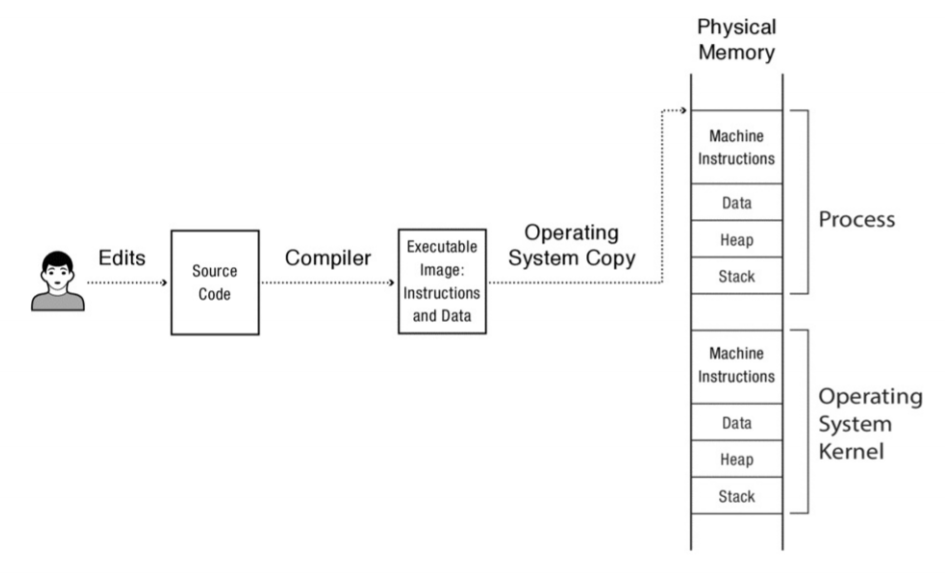
The OS kernel had the job of implementing this protection. The kernel which is the lowest level of software on the system, has full access to the machine hardware. Therefore, the kernel is trusted to do anything with the hardware, however untrusted software is running in a restricted environment with less complete access to the full power of the hardware.



*A process*: the execution of an app with restricted rights.

A process needs permission from the OS kernel before accessing the memory of any other process, before reading and writing to the disk, before changing hardware setting, and so on.

# 2.1 The Process Abstraction

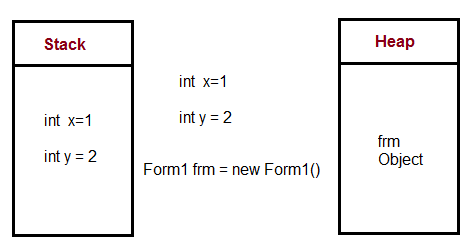


A programmer writes the source code in high level language, then compiles it to an executable image, which contain instructions and data. To be able to run the program, the instructions and data is copied to physical memory. The OS also sets aside a memory region called the *execution stack*. The OS also set aside a memory region called the *heap*.

**Execution stack**: Holds the state of local variables during procedure calls.

**Heap**: Stores any dynamically allocated data structure the program might need.

The OS also need to load itself into memory, with its own stack and heap, to able to run the program.



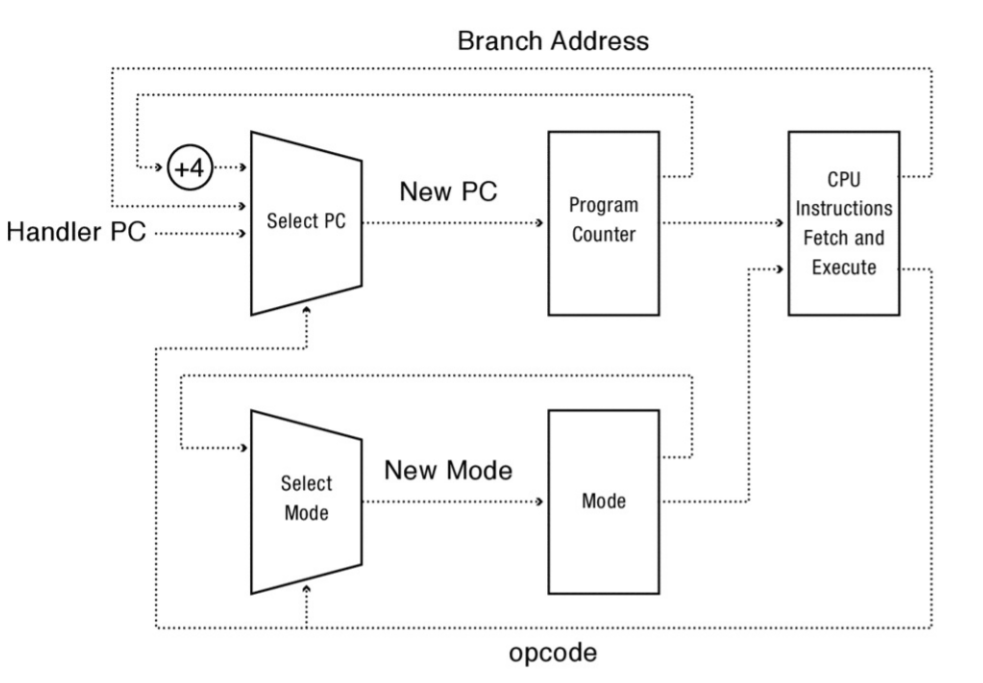
If we ignore protection, once the program is loaded into memory, the OS can start running it by setting the stack pointer and jumping to the program’s first instruction. To be able to run multiple programs the OS must load several instructions, static data, heap and stack in memory, though this it not the case since OS use the same copy for several instances of the same program.

Thus, a process is an instance of a program. The OS keeps track of these processes using a data structure called process control block, also referred to as PCB. This stores all the info the OS needs; where it is stored in memory, where its executable image is on the disk, which user asked to execute, what privileges the process has and so forth.

**Threads**: when a program consists of multiple concurrent activities, such as web browsers which must load scripts and at the same time get network information and load the web page. These separate activities run on its own program counter and stack but operates on the same code and data as the other threads.

A process executes a program, consisting of one or more threads running inside a protection boundary.

# 2.2 Dual-Mode Operation





**Dual mode operation**: Allowing the safe instructions to execute directly on the hardware.

This switch between the two modes, user-mode and kernel-mode, is represented by a single bit in the processor status register.

Code that is unnecessary to have in the kernel will be in user-lever code, since the kernel has full control over the hardware and must be trusted, which makes the OS more reliable.

The hardware must support these three things to let the OS kernel protect apps and users from one another, yet also let user code run directly on the processor:

1. **Privileged instructions**: All potentially unsafe instructions are prohibited when executing in user mode.
2. **Memory protection**: All memory accesses outside of a process’s valid memory region are prohibited when executing in user mode.
3. **Timer interrupts**: Regardless of what the process does, the kernel must have a way to periodically regain control from the current process.
4. **Privileged instructions**

**Privileged instructions**: instructions available in kernel mode, but not in user mode.

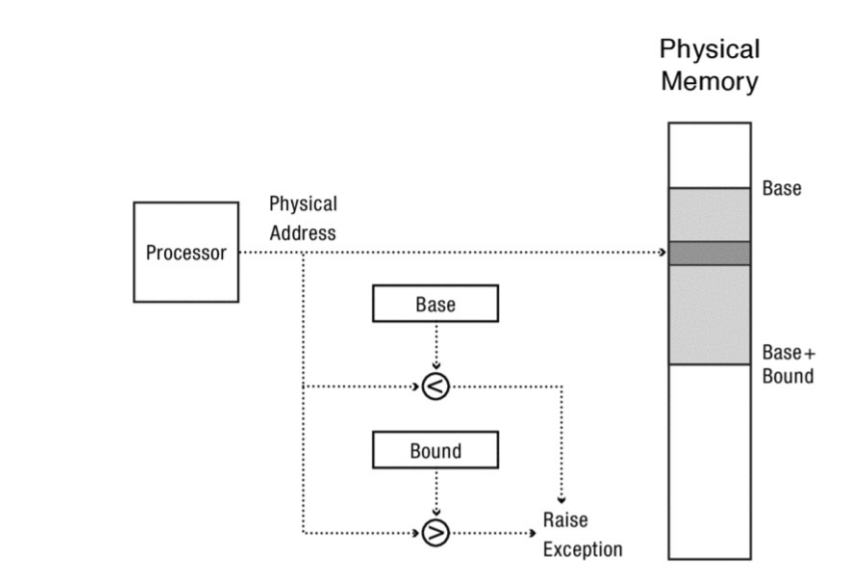
Programs are not allowed to change their privilege level by executing a special instruction, also called system call. Also, the app cannot be allowed to change the set of memory locations it can access.

Processor exception: The processor transfer control to an exception handler in the OS kernel. Usually the kernel will terminate the process / app from running after a privilege violation.

A process exception happens when an app attempts to access restricted memory or attempts to change its privilege levels.

1. **Memory protection**

Both the app process and the OS must be in the physical memory at the same time for an app to run. To make sharing safe, the application cannot read or write outside its own allocated memory space.



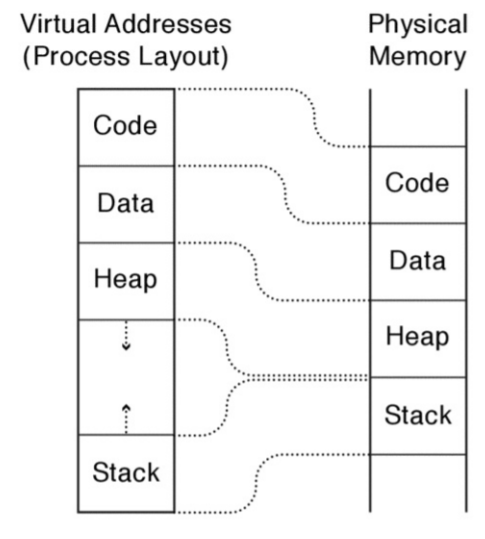
**Base and Bound**

Base and bound are used to prevent a user from accessing parts of physical memory that they are not supposed to access.

With this approach the processor has two extra registers, called base and bound. The base specifies the start of the memory space, and the bound specifies the end of it. These values can only be changed by the OS kernel. Each time the processor fetches an instruction it checks the address of the program counter to see if it is between the base and bound registers. The kernel executes without the base and bound registers.

Even though base and bound can provide protection, it cannot provide: *Expandable heap and stack, Memory sharing, Physical memory addresses, Memory fragmentation.*

For this reason, most modern processers use virtual addresses.

**Virtual addresses**

With virtual addresses every processes memory starts at the same place, 0. It thinks it has the entire machine for itself. The hardware translates these virtual addresses into physical memory locations.

Virtual addresses allow the stack and heap regions of a process to grow independently. To grow the heap, the operating system can move the heap in physical memory without changing the heap’s virtual address.

*Address space layout randomization*: Randomizing the virtual addresses to combat viruses.

1. **Timer interrupts**

The OS kernel needs to periodically regain control over the processor to prevent that apps have total control over the processor. If a program contained an infinite loop or we want to stop executing in the middle of the process, we need time interrupts in the processor. Also, for example if you are listening to music, the computer also needs to respond to user input therefore it must be able to regain control to switch to a new task.

This is achieved by a device called *hardware timer*. It can interrupt the processor after a given time delay or after a number of instructions have been executed. Each timer interrupts only one processor, therefore a multicore processor will have several timers.

The average human reaction time is a few hundred milliseconds, the OS might set the timer to expire every few milliseconds.

# 2.3 Types of Mode Transfer

Now when the user process is placed in a carefully constructed sandbox, the next question is how to safely transition from executing a user process to executing the kernel, and vice versa. This switching can happen up to thousands of times each second, therefore it is important that it is safe and fast.

**User to Kernel Mode**

Transitioning from kernel mode to user mode is just “undo”-ing or reversing this process.

Three reasons why the kernel should take control from a user process: *interrupts, processor exceptions, and system calls.*

1. **Interrupts**

Before the processor executes an instruction, it checks whether an interrupt has arrived. If there is an interrupt it will save current execution state and start executing at a specially designated interrupt handler in the kernel. Each interrupt has its own handler.

Interrupts are also used for I/O requests. For example, mouse device hardware triggers an interrupt every time the user moves or clicks on the mouse.

Interprocessor interrupts happens when a parallel program for example exits, and therefore the process on the other processor should be terminated as well.

1. **Processor exception**

A processor exception happens when a hardware event caused by user program behavior which causes a transfer of control to the kernel.

Examples:

* Process attempts to access memory outside its own memory region.
* Process attempts to perform privileged instructions.
* Divides an integer by zero.
* Accesses a word of memory with a non-aligned address.

In these situations, the system will halt the process and return an error code to the user.

1. **System calls**

User processes can also transition into the OS kernel voluntarily to request that the kernel perform an operation on the user’s behalf. A system call is any procedure provided by the kernel that can be called from user level.

**Kernel to user mode**

Just as there are several different types of transitions from user to kernel mode, there are several for kernel to user mode as well:

* New process
* Resume after an interrupt, processor exception, or system call.
* Switch to a different process.
* User-level upcall

# 2.4 Implementing Safe Mode Transfer

When switching from user to kernel mode or in the opposite direction, it is important to ensure that a buggy or malicious user program cannot corrupt the kernel. The processor must save its state and switch what it is doing. A similar example of the complexity of such an operation, is to change a cars transmission while it goes down the road at 100km/h.

At minimum, this common sequence must provide:

1. **Limited entry into the kernel**

When transferring control to the OS kernel, the hardware must ensure that the entry point into the kernel is one set up by the kernel. A user program should not be allowed to jump to arbitrary locations in the kernel.

1. **Atomic changes to processor state**

When in user mode, the program counter and stack point to memory locations in the user process. Memory protection prevent the user form accessing any memory outside the defined region. However, when in kernel mode, the program counter and stack point to the memory locations in the kernel. The memory protection is changed to allow the kernel to access its own data and user process. The transition is atomic.

1. **Transparent, restartable execution**

An interrupt can happen during any user process; therefore, it is important to store the state of the user program exactly as it was before the interrupt occurred. To the user this interrupt is invisible, except that the program temporarily slows down.

When an interrupt happens:

User process 🡪 Interrupt 🡪 Processor saves its current state to memory 🡪 changes to kernel mode 🡪 jumps to the interrupt or exception handler 🡪 steps are reversed when handler is finished.

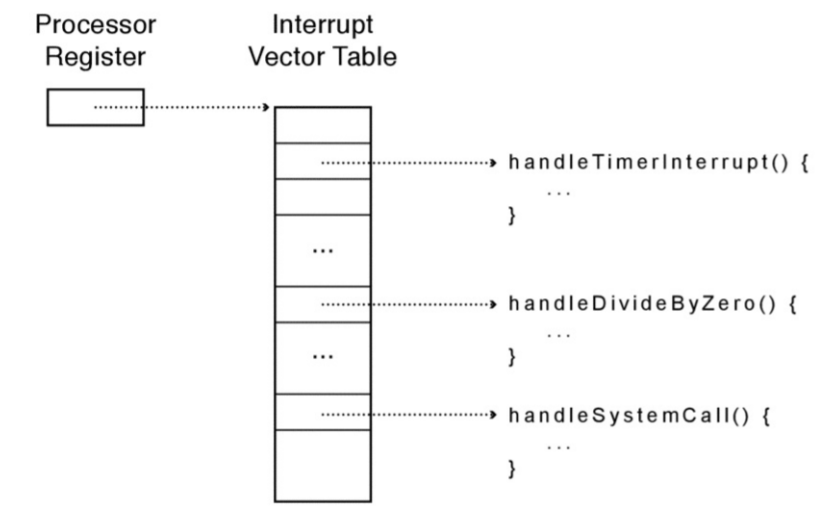
The processor state is restored from its saved location, with the interrupted program none the wiser.

**Interrupt Vector Table**

The processor has a special register that points to an area of kernel memory, called the interrupt vector table. This table holds an array of pointers, which points to the first instruction of a different handler procedure in the kernel, also called an interrupt handler.

Interrupt examples: hardware interrupts, processor exceptions and system calls.

Early computers sent the interrupt to “processor 0” and sent an interprocessor interrupt if the event required a change to what another processor was doing. However, in for example web servers, it is much more efficient to spread this to all processors, therefore interrupt routing is increasingly programmable. For example, disk I/O event can be sent directly to the processor that requested it.



**Interrupt stack**

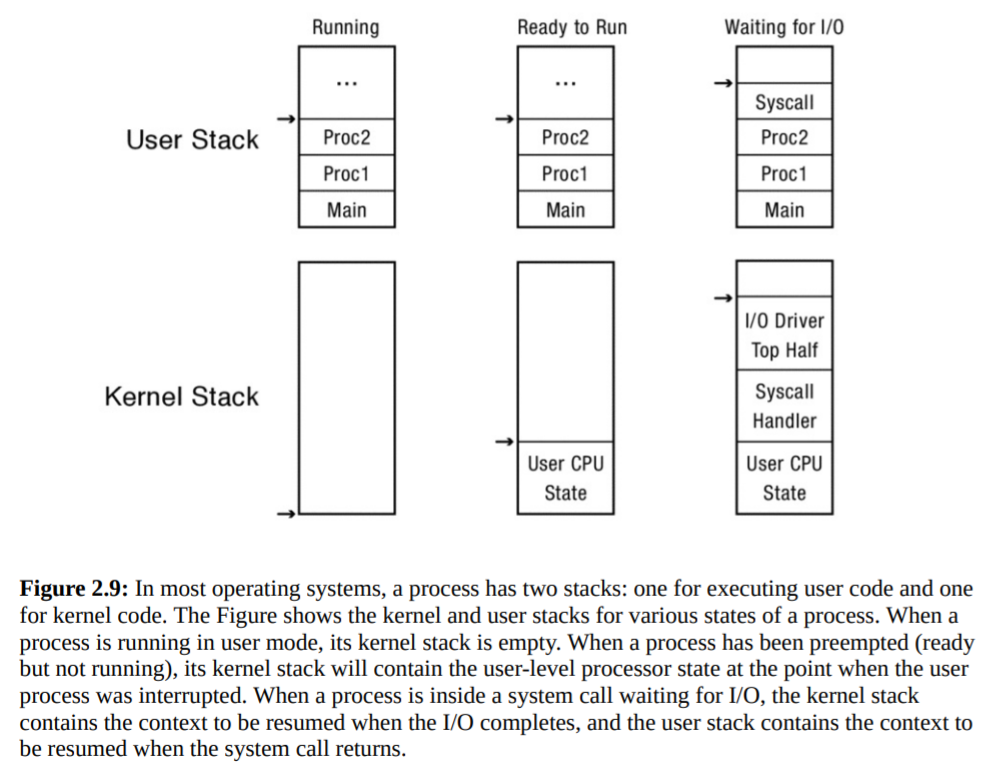
When an interrupt happens, the interrupted process state must be saved, therefore there is special privileged hardware register that points to a region of kernel memory called the *interrupt stack*. Right after an interrupt the hardware saves some of the interrupted processes registers onto the stack. Any remaining registers is saved by the kernel handler. After an interrupt this process is reversed.

There are two reasons to why the state is stored in the kernel stack rather in the user-level stack:

**Reliability**: User-level stack pointer might not be a valid memory address (e.g. if the program has a bug), but the kernel handler must continue to work properly.

**Security**: On a multiprocessor, other threads running in the same process can modify user memory during the system call. If the kernel handler stores its local variables on the user-level stack, the user program might be able to modify the kernel’s return address, potentially causing the kernel to jump to arbitrary code.

**Two Stacks per Process**



**Interrupt masking**

An interrupt could arrive when the processor is executing in either user or kernel level. If an interrupt arrives when the kernel is executing for example the interrupt handler, it could cause confusion. Therefore, the hardware provides a privileged instruction to temporarily defer delivery of an interrupt until it is safe to do so.

This instruction is called disable interrupts on x86 and many other processors. If an interrupt arrives during kernel execution the interrupt will be masked, not ignored, and unmasked once the execution is done. Multiple interrupts will be delivered in turn when interrupts are re-enabled.

**Hardware Support for Saving and Restoring Registers**

An interrupted processors registers must be saved so that the progress can be restarted exactly where it left off. The handler might change the values in those registers, therefore it must be saved before the handler runs.

The hardware saves the values for the stack pointer, program counter, and processor status word. Once the handler starts running, these values will be those of the handler, not those of the interrupted process.

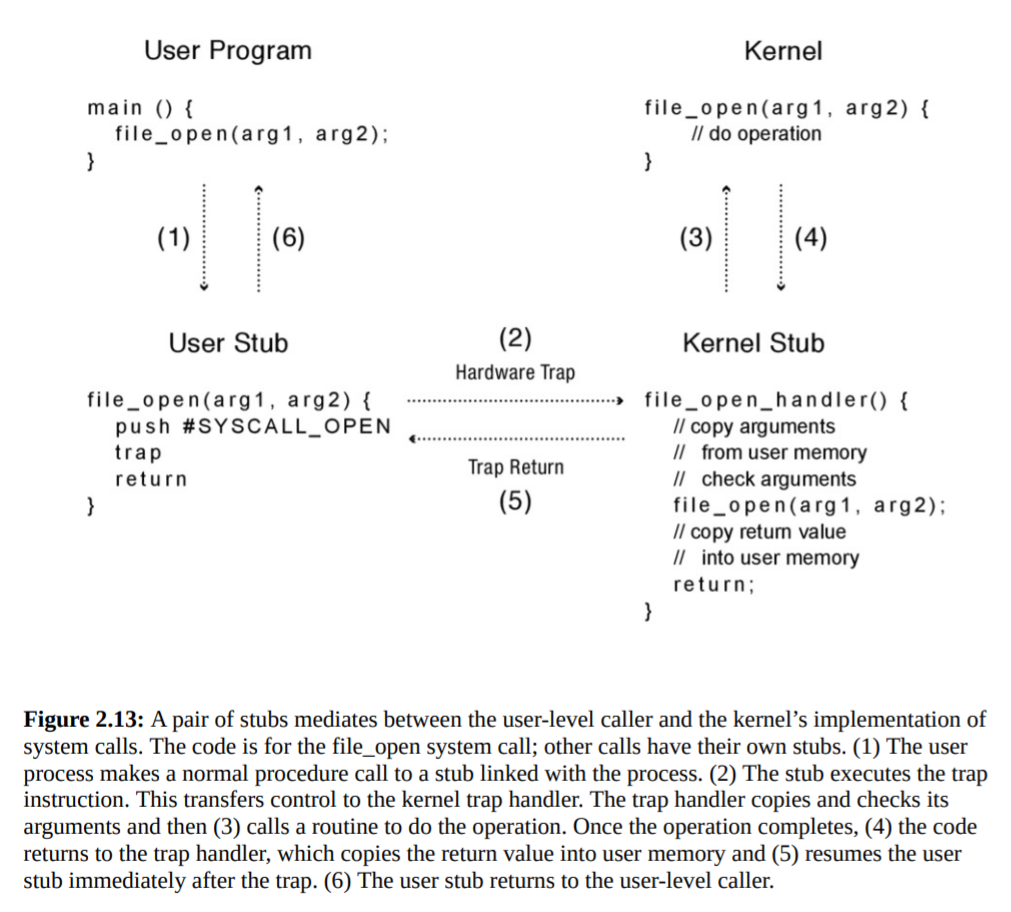
# 2.5 Putting it All Together: x86 Mode Transfer

How to implement an interrupt-triggered mode switch on the x86 architecture. Several OS on the x86 follow this principle, but the details differ.

# 2.6 Implementing Secure System Calls

Process execution happens in a restricted environment that the OS kernel constructs. If a process needs to perform an action outside of its protection domain (e.g. read from keyboard or write a disk block), it musk ask the OS to perform the action on its behalf, via a system call. A system call provides the illusion that the OS kernel is simply a set of library routines available to user programs.

Inside the kernel, a procedure implements each system call. This procedure behaves exactly as if the call was made from within the kernel but with one notable difference: the kernel must implement its system calls in a way that protects itself from all errors and attacks that might be launched by the misuse of the interface. Errors in an application program must not crash the kernel.



1. The user program calls the user stub in the normal way, oblivious to the fact the implementation of the procedure is in fact in the kernel.
2. The user stub fills in the code for the system call and executes the trap instruction.
3. The hardware transfers control to the kernel, vectoring to the system call handler. The handler acts as a stub on the kernel side, copying and checking arguments and then calling the kernel implementation of system call.
4. After the system call completes, it returns to the handler.
5. The handler returns to user level at the next instruction in the stub.
6. The stub returns to the caller.

The kernel stub has four tasks:

1. **Locate system call arguments**: arguments to a system call is stored in user memory, therefore the stub must check the address to verify it is a legal address within the user domain.
2. **Validate parameters:** To protect the kernel, it must check the format and content of its argument. It could be a file name which is corrupted, or it could be outside of the memory region of the app.
3. **Copy before check**: The parameters are checked before the necessary checks in the kernel. The reason for this is to prevent apps from modifying the parameter after the stub checks the value. This is called a time of check vs time of use (TOCTOU) attack.
4. **Copy back any results**: For the user program to access the results of the system call, the stub must copy the result from the kernel into user memory. Again, the kernel must first check the user address and convert it to a kernel address before performing the copy.

# 2.7 Starting a New Process

To start running at user level the kernel must:

* Allocate and initialize the process control block (PCB).
* Allocate memory for the process.
* Copy the program from disk into the newly allocated memory.
* Allocate a user-level stack for user-level execution.
* Allocate a kernel-level stack for handling system calls, interrupts and processor exceptions.

*In addition to this the kernel must do two things to run the program:*

1. **Copy arguments into user memory**: often when starting a program, the user may give it arguments, much like calling a procedure. These arguments are copied to the base of the user-level stack, and the users stack pointer is incremented so those addresses are not overwritten when the program starts running.
2. **Transfer control to user mode:** When we create the new process, we allocate a kernel stack to it, and we reserve room at the bottom of the kernel stack for the initial values of its user-space registers, program counter, stack pointer, and processor status word. To start the new program, we can then switch to the new stack and jump to the end of the interrupt handler. When the handler executes popad and iret, the processor “returns” to the start of the user program.

# 2.8 Implementing Upcalls

To allow apps to implement OS like functionality, we need to virtualize some part of the kernel. We call virtualized interrupts and exception *upcalls*.

Several uses for immediate event delivery with upcalls:

**Preemptive user-level threads**: An app may run multiple tasks, or threads, in a process. A user-level thread package can use a periodic timer upcall as a trigger to switch tasks, to share the processor more evenly among user-level tasks or to stop a runaway task, e.g., if a web browser needs to terminate an embedded third party script.

**Asynchronous I/O notification**: With asynchronous I/O we can make a system call and immediately return, so the processor can do other work in the meantime. When the I/O completes the notification can be sent via an upcall.

**User-level exception handling**: Sometimes an app has its own exception handling routines, e.g. to ensure the files are saved before the app shuts down. To do this the OS needs to inform the app when it receives a processor exception, so the app runtime can handle it, not the kernel.

**User-level resource allocation**: Sometimes an app can be resource adaptive, which means it is able to optimize its behavior to differing amounts of CPU time or memory. Therefore, the OS must inform the process when its allocation changes.