Memory isolation.

Processes have their own reserved quota in RAM. Draw a diagram. However, this reserved quota is decided by OS during runtime. Because the actual physical addresses for a process is only known during the fork/exec system call, the compiler cannot generate real addresses in statically generated code. The above limitation puts an unnecessary burden on the compiler to emit relocatable code. The hardware provides another abstraction called virtual address, which is uniform among all the processes and an additional hardware unit called MMU translates the virtual addresses to actual physical addresses during runtime. The compilers can generate code to access virtual addresses and MMU automatically translate them to the real physical address during memory access. E.g., if MMU is enabled instruction "movl $100, 100" stores 100 to virtual address 100. On execution of this instruction, MMU translates 100 to a real physical address and store 100 at that location.

Initially, OS allocates a contiguous RAM space of size “limit” for each process. The application does not know anything about the actual RAM addresses. Instead, a virtual address space from (0 - limit) is exposed to the applications. This makes the compiler job easier because they don’t have to worry about real physical addresses anymore.

To facilitate virtual to physical translation, every memory operand also contains a segment register. X86 has six segment registers namely cs, ds, es, ss, fs, gs. These segment registers contain a 16-bit value, whose higher 13-bit value, is indexed into an array of descriptors to get the base address of the segment. The base address is added to the virtual address to obtain the real physical address. The array of descriptors is called global descriptor table (GDT). A GDT entry contains the base and limit of a segment. Below is pseudocode of a memory access:

```
movl $100, %ds:100
---------------------------
index = ds >> 3;
base = gdt[index].base;
unsigned *phys = 100 + base;
assert (100 < gdt[index].limit);
*phys = 100;
```

Who creates the GDT?

OS

Where does GDT lives?

OS address space

How does hardware know the address of GDT?

GDTR register contains the address of GDT. “lgdt” instruction takes the address of GDT and set the GDTR register to the address of the GDT.

Why can applications not create their own GDT in memory and execute “lgdt” instruction to make hardware use their GDT?

Protection rings:

X86 has four protection rings (0,1,2,3). Most OS use only zero and three. Zero is most privileged, and OS runs in this protection ring. Three is least privileged, and user-programs execute in this ring. The only difference between these rings is that some instructions are only allowed in ring-0 or have a different meaning when executed in different protection rings. “lgdt” instruction is only allowed in ring-0. Ring-0 is also called the supervisor mode.

How does OS know the current privilege level (CPL)?
The last 2-bits of the cs registers contains the CPL. CPL zero is OS kernel, 3 is user-program.

What prevents from app to modify the GDT table itself.
GDT table lives in the OS memory.

Context Switch:
One way to use segmentation is to have dedicated GDT to every process. On every context switch, the OS loads the GDT corresponding to the target process.

Another way is to use the same GDT for all processes, but overwrite the GDT entries with the target process entries on every context switch.

How does OS access the entire memory when GDT is active?
OS segment selectors point to the GDT entries whose base and limit are set to 0 and (2^32-1) respectively.

Why can user applications not set their segment registers to use GDT entries for OS?
GDT entries also have permission filed that disallow non-supervisor access to these entries.

Traps

The x86 processor uses a table known as the *interrupt descriptor table* (IDT) to determine how to transfer control when a trap occurs. The x86 allows up to 256 different interrupt or exception entry points into the kernel, each with a different *interrupt vector*. A vector is a number between 0 and 256. An interrupt's vector is determined by the source of the interrupt: different devices, error conditions, and application requests to the kernel generate interrupts with different vectors. The CPU uses the vector as an index into the processor's IDT, which the kernel sets up in kernel-private memory of the kernel's choosing, much like the GDT. From the appropriate entry in this table the processor loads:

- the value to load into the instruction pointer (`EIP`) register, pointing to the kernel code designated to handle that type of exception.
- the value to load into the code segment (`CS`) register, which includes in bits 0-1 the privilege level at which the exception handler is to run.

Exceptions:

Exceptions are similar to the interrupts but they are internal to a CPU. IDT entries 0-31 are reserved for exceptions. Some examples of exceptions are divide-by-zero (when CPU tries to execute divide by zero), general protection fault (process tries to access a virtual address outside its limit). A unique vector number is assigned to each exception. e.g., zero for div-by-zero, 13 for general protection fault.

Entering and Returning from Trap Handlers

When an x86 processor takes a trap while in kernel mode, it first pushes a *trap frame* onto the kernel stack, to save the old values of certain registers before the trap handling mechanism modifies them. The processor then looks up the CS and EIP of the trap handler in the IDT, and transfers control to that instruction address. The following diagram illustrates the format of the basic kernel trap frame, defining the state of the kernel stack on entry to the trap handler:

```
+--------------------+ <---- old ESP
|     old EFLAGS     |     " - 4
|      old CS        |     " - 8
|      old EIP       |     " - 12
+--------------------+ <---- ESP
```

For certain types of x86 exceptions, in addition to the basic three 32-bit words above, the processor pushes onto the stack another word containing an *error code*. The page fault exception, number 14, is an important example. See the x86 manuals to determine for which exception numbers the processor pushes an error code, and what the error code means in that case. When the processor pushes an error code, the stack would look as follows at the beginning of the trap handler:

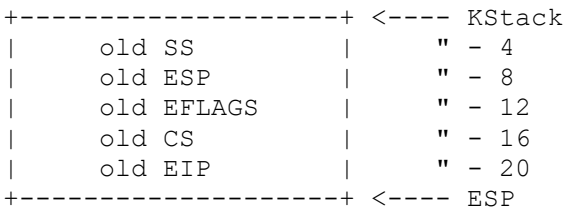
```
+--------------------+ <---- old ESP
|     old EFLAGS     |     " - 4
|      old CS        |     " - 8
|      old EIP       |     " - 12
|     error code     |     " - 16
+--------------------+ <---- ESP
```

The x86 processor provides a special instruction, `iret`, to return from trap handlers. It expects the kernel's stack to look like the *first* figure above, with ESP pointing to the old EIP. When the processor executes an `iret` instruction, pops the saved values of EIP, CS, and EFLAGS off the stack and back into the corresponding registers, and resumes instruction execution at the popped EIP.

Note that when returning from a trap, the processor doesn't actually know or care whether the "old" values it is popping off the stack are really the exact same values that it originally pushed onto the stack on entry to the trap handler. Think about what would happen - for better or worse - if the kernel trap handler changes these values during its execution.

Entering from User Mode.

User stack pointers cannot be trusted. For this reason, when a user-program is interrupted the CPU switches to the per-thread kernel stack before jumping to the target interrupt handler. Because CPU clobber the stack pointer, it must save them on the kernel stack in addition to user EIP. Similarly, during IRET when the CPU returns to user-mode, it also restores user-stack pointer.



The Task State Segment.

On an interrupt in user program the hardware automatically switches to the kernel stack and stack segment.

A structure called the *task state segment* (TSS) specifies the stack segment selector and address where the kernel stack lives. The processor pushes (on this new stack) `SS`, `ESP`, `EFLAGS`, `CS`, `EIP`, and an optional error code. Then it loads the `CS` and `EIP` from the interrupt descriptor, and sets the `ESP` and `SS` to refer to the new stack.

Combined, the IDT and TSS provide the kernel with a mechanism to ensure that traps are handled only by calling well-defined entry points in the kernel (the interrupt vectors in the IDT) and that trap handlers will have a well-defined, protected workspace (the stack pointers in the TSS). Exactly *where* these entry points and kernel stacks are located is up to the kernel.

Entering User Mode

What piece of register state in the processor actually defines which privilege level it is executing in at a given moment? Many architectures use a "kernel mode" flag in a control register of some kind, but x86 processors uses the low two bits of the CS register, effectively treating privilege level as a property of the currently running code segment.

We described above how the processor *leaves* user mode and enters the kernel via a trap, but how does the kernel *enter* user in the first place? Simple: the kernel "returns" to user mode - even if the processor has never been there before!

When the processor executes an `iret` instruction, it pops its standard trap frame off the stack starting with the old EIP, but it doesn't actually know or care whether *it* actually pushed that frame on the stack or if it got there some other way. Thus, the kernel can always *manufacture* a trap frame representing whatever user mode state it wants to load into the processor, and "return"

from it via `iret` to enter user mode. (There are other ways to switch to user mode on the x86, but this is the most general method.)

Software Interrupts

Now that your kernel has basic exception handling capabilities and can enter user mode, you will refine it to handle traps that user mode code may cause deliberately for various purposes; we refer to such traps as *software interrupts*.

- `T_BRKPT`: The breakpoint exception, interrupt vector 3, is normally used to allow debuggers to insert breakpoints in a program's code by temporarily replacing the relevant program instruction with the special 1-byte `int3` software interrupt instruction.
- `T_OFLOW`: The overflow exception, interrupt vector 4, allows software to generate a trap deliberately via the special `into` software interrupt instruction, if the overflow flag (FL_OF) is set when the instruction executes. The intent is for software to execute an `into` after an arithmetic operation that might overflow: if it doesn't, execution proceeds normally, but if it does, the overflow condition is immediately caught. This appealing idea has never really caught on in high-level languages, however, and is rarely used.

- `T_SYSCALL`: An interrupt vector number is reserved for SYSCALL.

We do not want user code to be able to invoke *just any* interrupt vector in the IDT deliberately via `int` instructions, however: that might allow user code to confuse the kernel into thinking that some special event has occurred when it has not. For this reason, all IDT descriptors have a *descriptor privilege level* indicating what privilege level is required for software to invoke that interrupt vector deliberately via a software interrupt instruction. Most of these vectors are normally set to "privileged" (ring 0), but we want the vectors used for software interrupts to be set so user mode code can invoke them.

Fragmentation:

One of the problems with segmentation is fragmentation. Because the process address space is contiguous, the OS cannot allocate extra physical resources for an application if the contiguous memory is occupied. In this case, the OS has to compact the physical RAM by relocating some processes (requires copying the entire process address space). Draw figure.

The process address space is limited to the available RAM. So, if you have 1 GB RAM, the processes can use at most 1 GB of virtual addresses.

Due to these limitations, the x86 hardware has an additional MMU called the paging hardware. In the paging scheme, the address space is not limited by the amount of RAM.