Descriptors

0.1 Segment/Interrupt Descriptors

0.1.1 Segment descriptor format

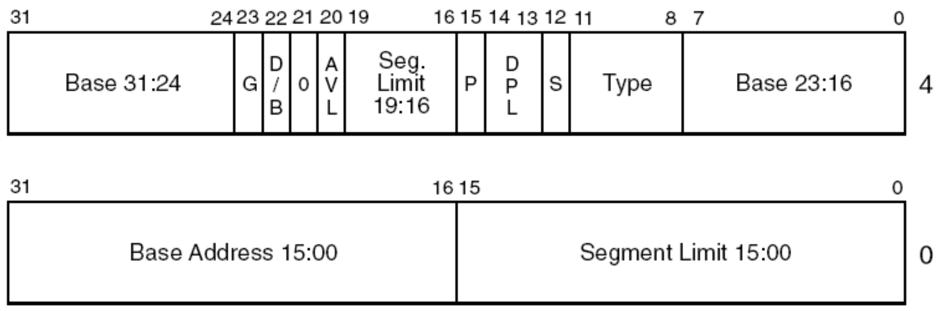

Figure 1: Segment descriptor format.

A quick note about how the descriptor is shown in Figure **??**: the top line of the figure is the upper 4 bytes of the 8-byte descriptor, and the bottom line is the lower 4 bytes. Here is the the descriptor as a 8-byte constant in C or GNU assembly:

```
0x(top line)(bottom line)
```

In effect, the pictures should look as shown in Figure **??**.

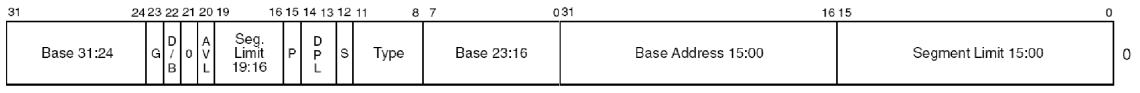

Figure 2: Single-line descriptor format.

Students (and TAs :)) have gotten confused in the past on the byte and bit orderings of the descriptor fields.

Description of fields:

- **Base 31:24**, **Base 23:16**, **Base 15:00**: The “Base” fields make up a 32-bit virtual base address that determines the start of the segment. You want these fields to be all set to 0 for a flat segment model.
- **Seg. Limit 19:16**, **Segment Limit 15:00**: A 20-bit segment limit that specifies the size of the segment. Its real meaning changes based on the value of the **G** field (below), but for MP4 you want it to be set to `0xFFFFF` to indicate a segment size of 4GB (in effect, no limit) for a flat segment model.
- **G**: Granularity. Determines the granularity of the segment limit field. If set to 0, the segment limit is taken to be a size in bytes. If set to 1, the segment limit is taken to be the number of 4K units in the segment.

  Example: If the Segment Limit field is `0xFFFFF`, and G is 0, the size of the segment is 1MB (1024K). If the Segment Limit field is `0xFFFFF`, and G is 1, the size of the segment is 4GB (1MB of 4K units, which is 4GB).
- **D/B**: Default operation size. Specifies 16-bit or 32-bit operations on this segment. Set this field to 1 for 32-bit operations always.

- **0**: Reserved, must always be set to 0. Make sure they explicitly zero out these bits and do not just assume that they are zero. Variables (especially stack-allocated local variables) in C code used to set up these descriptor entries contain garbage values, so some of these reserved bits may happen to be 1. If they load a descriptor with some reserved bits set to 1, they'll get some sort of exception (invalid descriptor exception).
- **AVL**: Available for the OS's own use. We don't use these bits at all.
- **P**: Present. Indicates that this segment descriptor is valid. Set this field to 1 for all segment descriptors you want to set up.
- **DPL**: Descriptor Privilege Level. Specifies the privilege level (0-3) necessary to access this segment. User-level code runs at privilege level 3, and kernel code runs at privilege level 0, so if you want a user- or kernel- accessible segment, the DPL of the segment should be set to 3. For a kernel-only segment, the DPL should be 0. A general protection exception is generated if code with a privilege level of 3 tries to access a segment with privilege level 0 (for example).
- **S**: Descriptor Type. If set to 0, this is a "system descriptor". System descriptors specify weird things like LDTs, TSSs, call gates, *etc.* If set to 1, the descriptor is for a code or data segment. For normal segments, this field should be a 1. In the MP4 code, there are a few descriptors that have this field set to 0 for the LDT and TSS, but those are in the given code.
- **Type**: Specifies code or data segment, and also the direction that the segment "grows". Bit 11 specifies code (1) or data (0). Bits 10-8 specify read/write permissions, accessed, and conforming/non-conforming or growth direction. Basically, a type of `0x2` specifies a read/write data segment, and a type of `0xA` specifies a executable code segment. Those are the only useful types for MP4. The full description is given on pg. 79 of volume 3 of the Intel manuals (section 3.4.3.1) if anyone is interested.

Segments that are already defined for you in the MP4 source:

- `KERNEL_CS`: A 32-bit segment with base 0, limit 4GB, DPL 0, segment type executable code. This segment should be selected by the CS register when executing in the kernel.
- `KERNEL_DS`: A 32-bit segment with base 0, limit 4GB, DPL 0, segment type read/write data. This segment should be selected by DS and SS when executing in the kernel.
- `USER_CS`: A 32-bit segment with base 0, limit 4GB, DPL 3, segment type executable code. This segment should be selected by the CS register when executing a user-level program.
- `USER_DS`: A 32-bit segment with base 0, limit 4GB, DPL 3, segment type read/write data. This segment should be selected by DS and SS when executing a user-level program.

The segment selectors (the values that go into CS, DS, and SS) are actually defined as `KERNEL_CS`, `KERNEL_DS`, *etc.* Those values contain the correct index values into the GDT to point to the segment descriptors.

0.1.2 Interrupt descriptor format

Description of fields:

- **Offset 31:16**, **Offset 15:0**: A 32-bit address of the interrupt handler to run when the interrupt corresponding to this descriptor is asserted (via a hardware interrupt, exception, or an `int` instruction). These fields should be set to the function address or label of your interrupt handler. The address is broken up into two fields, so they'll have to use shifting and masking operations to take a single 32-bit address and get the two 16-bit parts and put them in the right locations.
- **Segment Selector**: A 16-bit value that will get put into the CS register when the interrupt, exception, or `int` instruction is executed. Set it to `KERNEL_CS`.

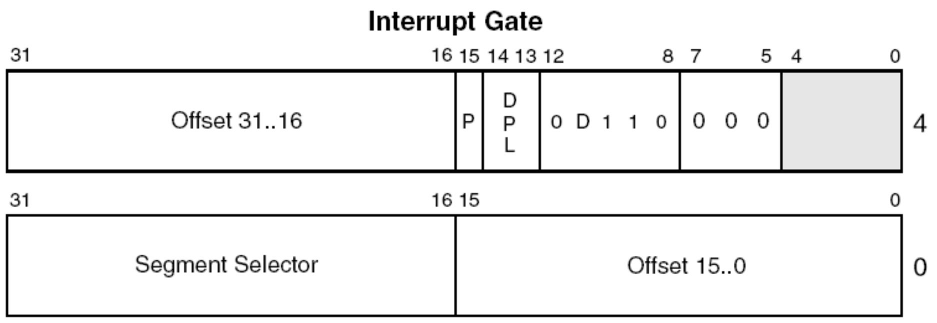

Figure 3: Interrupt descriptor format.

- **P**: Present bit, indicates that this is a valid interrupt descriptor. Set this bit to 1.
- **DPL**: Descriptor privilege level, same meaning as with the DPL in the segment descriptors. Hardware interrupt descriptors and exception descriptors should have their DPLs set to 0. The system call descriptor (index `0x80`) should have its DPL set to 3 so that user-level programs can execute system calls by executing `int $0x80`.
- **D**: Size. Should be set to 1 to indicate a 32-bit interrupt gate.

Trap gates vs. interrupt gates

The Intel manual has information on "trap gates" as well as interrupt gates. They are basically the same, but interrupt gates disable interrupts upon entry into the new code, whereas trap gates do not disable interrupts. For simplicity, use interrupt gates for **everything**, and then selectively enable interrupts in the actual code that executes the handler using a `cli` instruction. Disabling interrupts upon entry into the kernel is useful because you may need to do certain interrupt-unsafe things in the code before you re-enable interrupts, like touch a shared data structure. The system call entry point and the exception handlers should re-enable interrupts at some point.

Example code

Example code to set up IDT entry (have them talk through it):

```
void setup_idt(void* handler_address, int irq_num)
{
    uint32_t idt_upper;
    uint32_t idt_lower;

    /*          Set Offset 31:16                Present     DPL      */
    idt_upper = (handler_address & 0xffff0000) | (1 << 15) | (3 << 13)
            | 0x0E00; /* Size, other '1' bits */

    /*          Segment Selector   Offset 15:00          */
    idt_lower = (KERNEL_CS << 16) | handler_address & 0xffff

    /* Fill in the entry in the IDT */
    idt[32+irq_num] = ((unsigned long long)(idt_upper) << 32) | idt_lower;
}
```

Note the use of the shifting and masking operations to take the `handler_address` and split it up into two fields, as well as shifting `KERNEL_CS` to get it into the right place.

This code assumes that the `idt[]` array is declared as an array of 64-bit (8-byte) values. To set an 8-byte value in that array, we have to create an 8-byte value on the right side of the assignment. We have to cast `idt_upper` to an `unsigned long long` (a 64-bit type) **before** shifting left by 32 bits, and then ORing the lower 32 bits into place.

Finally, we set `irq[32+irq_num]`. The hardware interrupts start at IDT entry 32 because entries 0-31 are reserved for exceptions (see the Intel manual). So, IRQ 2 is really IDT entry 32+2=34.

0.1.3 TSS

Task state segment. It can be used to perform context switching in hardware. We're not going to use the hardware task support in MP4 *except* for the privilege stack switch abilities. When a privilege switch occurs (interrupt, exception, system call), we need to switch from the user-level stack to the kernel stack. The kernel stack pointer to switch to is specified in the ESP0 and SS0 fields of the TSS, so they need to set up those fields before they execute user-level code.

The TSS structure is already defined in the code as the variable `tss`. Just include `#include "x86_desc.h"` whenever you need to access the TSS structure. The fields are `tss.esp0` and `tss.ss0`.

0.2 Page Descriptors

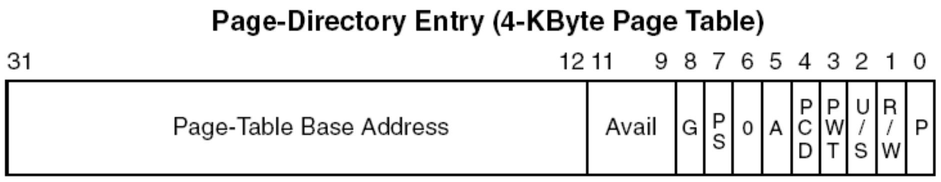

Figure 4: Page directory entry for a 4K page.

Description of fields:

- **Page-Table Base Address**: Upper 20 bits of the *physical* address of a 4K-aligned page table. In MP4, kernel physical and virtual addresses are the same, so the page table base address should be just a label or variable name from their code. All page tables must be 4K aligned, so the lower 12 bits of their addresses are 0.

- **Avail**: Available for our use, but we're not going to use these bits.

- **G**: Global bit. This bit controls TLB behavior for a particular page when the TLB is flushed (when CR3 is reloaded). If this bit is set (1), the page corresponding to this PDE/PTE is "global," and virtual→physical translations that use this page are visible to all processes. So, TLB entries associated with this page will not be cleared when CR3 is reloaded. If it is clear (0), the page corresponding to this PDE/PTE is a per-process page and the translations will be cleared when CR3 is reloaded. This bit should only be set if the page mapping is actually shared by all processes (this page directory entry exists in all processes' page directories). In MP4, set the bit for kernel page(s) only, since the kernel is mapped into every process's address space.

  This bit is *ignored* for 4K page directory entries, since there is still another translation hierarchy (the page tables). The bit is only relevant for 4K page table entries and 4M page directory entries.

- **PS**: Page size. If 0, this is a 4K page directory entry. If 1, this is a 4M page directory entry. We'll be using both in MP4 – kernel pages and program pages are 4M pages, and video memory is mapped using 4K pages.

- **A**: Accessed. The processor sets this bit to 1 when an access is made to a virtual address that corresponds to this entry. That is, when a process touches memory that this PDE maps, this bit gets set to 1. The bit and associated behavior can be used to implement some sort of LRU page replacement algorithm. We won't use it in MP4.
- **PCD**: Page cache disabled. If 0, the processor will not cache any memory that falls in this PDE's range. The bit should be 1 for program code and data pages (kernel pages, program pages). It should be 0 for video memory pages, since those pages contain memory-mapped I/O and should not be cached.
- **PWT**: Page write-through. If the page is being cached, this bit makes the caching write-through rather than writeback. If 0, the cache policy is writeback. If 1, the cache policy is write-through. We always want writeback, so this bit should always be 0.
- **U/S**: User/supervisor. If 1, memory mapped by this PDE/PTE is user-level (privilege levels 0-3 can access this memory). If 0, the memory is supervisor-only (privilege levels 0-2 can access this memory, but not privilege level 3). Since we run user-level code at privilege level 3, this bit allows us to mark pages or ranges of memory as kernel-only. This bit should be set to 1 for all user-level pages and memory ranges (for example, things that map user-level code and data), and 0 for kernel pages. The kernel's map of video memory at virtual address `0xB8000` – `0xC0000` (which is mapped to physical `0xB8000` – `0xC0000`) should be set to supervisor. When the user maps video memory via a `vidmap` system call, the PDEs and PTEs for their virtual address range (somewhere other than `0xB8000` – `0xC0000`, but still mapped to physical `0xB8000` – `0xC0000`) should be set to User (1).
- **R/W**: Read/write permission. If 1, the page(s) mapped by this PDE/PTE are read/write. If 0, page(s) are read-only. For our purposes, mark all pages read/write.
- **P**: Present. If 1, marks this PDE as valid, if 0, the page range associated with this PDE/PTE does not exist. All valid PDEs need to have their present bit set to 1.

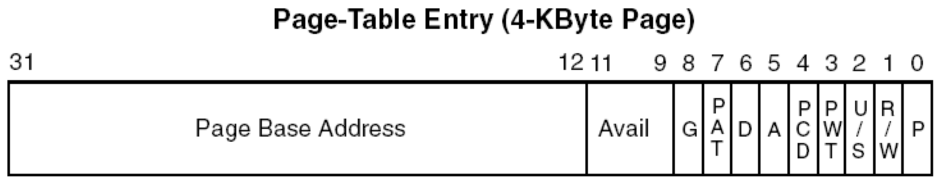

Figure 5: Page table entry for a 4K page.

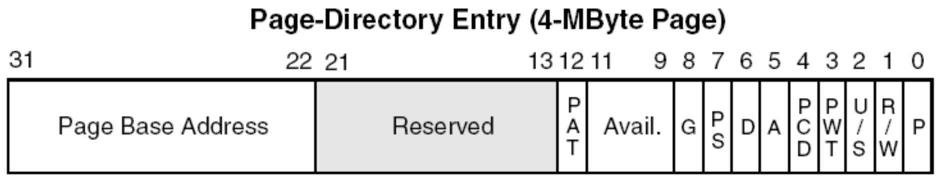

Figure 6: Page directory entry for a 4M page.

Fields not mentioned above:

- **PAT**: Page Attribute Table index. We're not going to use this field at all. Set it to 0.
- **D**: Dirty. The processor sets this bit to 1 if the page mapped by this PTE has been written to, and this bit remains 1 until it is cleared by software. The dirty bit is used in virtual memory systems to indicate that the page differs from what is in the swap disk. For example, when the physical page is going to be replaced and

used by a different process, the OS needs to know if it should write back the data in the page to the swap disk (or perhaps to the filesystem, *etc.*). The processor provides this support in hardware. We aren't going to use this bit at all, so set it to zero.

- **Reserved**: These bits *must* be set to 0, or they will get an invalid PDE exception.

0.2.1 Alignment

The page directories and page tables must be 4K-aligned. For example, the page directory must start at address 0x405000, not address 0x405d00. The way to force gcc to align variables is by using the gcc variable attribute extensions:

```
int some_variable __attribute__((aligned(4096)));
```

That line will force `some_variable` to be aligned at a 4096-byte (4KB) boundary.

In assembly, it's the .align directive:

```
.align 4096
label:
.long 0
```

Those lines will force ”label” to start at a 4K boundary.

0.2.2 Mapping video memory

Make sure you map video memory when setting up your paging. Video memory is at 0xB8000 - 0xC0000. It should be mapped using **4K pages**, NOT a single 4MB page that maps 0MB – 4MB. With a single 4MB page from 0MB – 4MB, the NULL address 0x0 is mapped to a valid range, and NULL pointer dereferences will not cause page fault exceptions. These NULL pointer bugs will then go undetected. Students **want** to catch NULL pointer dereferences – when developing, catching bad pointer dereferences is a good thing, as it will help find bugs that may otherwise manifest themselves in strange ways.

0.2.3 Allocating page tables

Because MP4 is simplified to only support 2 tasks, all the paging structures (page directories, page tables) for the two tasks can be statically allocated in a source file. A dynamically-allocated memory system is too complex to implement in 4 weeks, so this solution is best for this project.

0.3 Interrupts, Exceptions, System calls

0.3.1 The Hardware Context structure

Show figure from MP4 handout (Figure **??**). Talk about the convenience of this figure (the system call parameters are already on top). The bottom five elements always get pushed on by the processor. The 6th element (error code) sometimes gets pushed on by the processor depending on what exception occurred. Their interrupt handlers and the system call handler should push a dummy value in this slot to fill in the blank if no error code gets pushed (see the Intel manual for exactly which exceptions push an error code). Finally, every interrupt, exception, and system call handler should push their exception number on the stack. At this point, the first seven parameters are on the

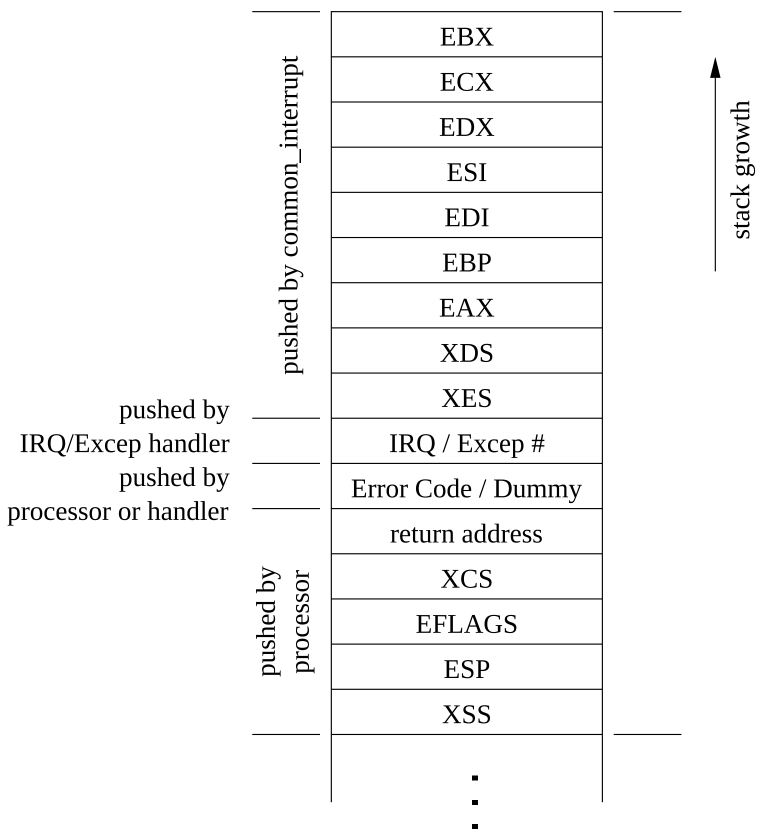

Figure 7: Hardware context structure (from MP4 handout).

stack. Then, the specific handlers should jump to common code that pushes the remaining elements onto the stack (the registers): specific interrupt handlers should jump to `common_interrupt`, specific exception handlers should jump to `common_exception`, and the system call handler should push the registers itself. Then, the assembly linkage is finished. Interrupts and exceptions should call C code that takes, as a parameter, a structure that exactly matches up with the parameters that have just been pushed. For example:

```
typedef struct {
    uint32_t ebx;
    uint32_t ecx;
    uint32_t edx;
    uint32_t esi;
    ...
} hwcontext_t;

void exception_handler(hwcontext_t hw_regs);
```

The C function appears as if it was called with a pass-by-value parameter of the `hwcontext_t` structure (pushed on the stack by the caller).

System calls need to validate that EAX holds a valid system call number (1-10). If the number is outside this range, they should put -1 into EAX and exit the kernel. If a valid system call number is in EAX, system calls should behave similarly to exceptions and interrupts (call a C function), but they should make a call through a system call table indexed on the EAX register. The syntax looks like this:

```
call    *syscall_table(,%eax,4)
```

This assembly code will call the function pointed to by `[syscalll_table+eax*4]`. We use the scaled addressing mode (by 4) because the function pointers in the system call table are 4 bytes each.

They should have common code to exit from the kernel. This code should restore all register values from the stack into the registers, remove the two dummy parameters (the interrupt number and the error code), and then do an `iret`. The `iret` will pop the last five parameters off the stack and return to executing in userspace.

System call exit code needs a little bit more work, because the EAX that was saved on the stack will be popped into the EAX register, overwriting whatever the current return value is. On return from system calls, their code should overwrite the EAX on the stack (in the hardware context structure) with the value that is in the EAX register, and then jump to the common return-from-interrupt code. The common code will put this value back into EAX, and the system call return value will behave properly.

An important thing to note is the behavior of `iret` when returning to userspace vs. returning to somewhere in the kernel. Hardware interrupts can happen when executing a user-level program or when executing a system call (or even another interrupt) in the kernel. If a hardware interrupt happens when executing user-level code, the hardware context structure will be exactly as seen here. If the interrupt occurs when executing in the kernel, no privilege switch will have occurred (because we were already running in privilege level 0), and so no stack switch will have occurred. Thus, the last two parameters in the hardware context structure will not be present. This is not a problem because the `iret` instruction figures out, based on the privilege level specified by lower 2 bits the CS field, what privilege level to return to, and thus if there are two extra parameters left on the stack (ESP and SS).

Signals

Signals should be delivered to a user-level process on return from the kernel to userspace. To see if a particular return is to user-level, they will have to check the lower 2 bits of the CS field for a privilege level of 3. If there is a pending signal and the return privilege level is 3, the signal should be delivered. If the return privilege level is not 3, the return is not to userspace (it's to another point in the kernel – perhaps this is a return from a hardware interrupt back into a system call). No signal should be delivered if the return privilege level is not 3 – when returning from privilege level 0 to privilege level 0, the ESP and SS fields will not be present (x86 does not push them on the stack at the privilege switch, and so they are not present on the return), and those values are necessary to figure out where to put the signal handler's stack frame and data.