Embedded X86 Programming: Protected Mode

By Jean Gareau

Intel has shipped millions of 80386, 80486, and Pentiums since 1986, and this figure is increasing rapidly. The x86 is expected to seriously affect the embedded systems market for the following reasons: applications can be developed on a PC (not necessarily on a target), both 16-bit and 32-bit programming are fully supported, a complete diversity of hardware is available, and GUI features—through Windows CE and 95—will become more accessible. [1]

Consequently, many existing RTOSs will likely be ported to this CPU—if not completely rewritten from scratch—to exploit the x86's capabilities. This CPU and its successors are armed with a battery of features that enables the implementation of the most advanced concepts in operating system design. These features also allow the writing of simpler embedded applications by providing 32-bit operations and various memory models that give applications large address space.

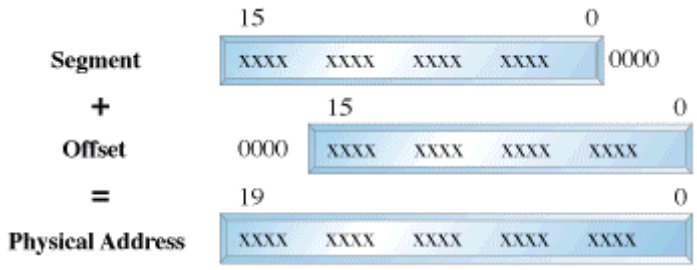
Development tools (compilers and linkers) also have some benefits, because popular memory models, such as the flat memory version, are simpler to support.

This article initiates a series presenting in-depth technical coverage of the most important features: protected mode (the subject of this article), segmentation, and paging. Functional examples are provided with each article to illustrate the concepts. To understand segmentation and paging concepts and how they can simplify embedded application development, the protected mode must first be explained in detail.

Complete examples that implement various kernel designs—fully documented and tested—can be downloaded from the *ESP* Web site at *www.embedded.com/code* . These examples demonstrate the concepts I'll explain in this series, including a port of mC/OS to protected mode. [2] The source code is provided, as well as ready-to-run executables and additional tools. These various implementations will provide a start to help you improve your applications or even implement your own system.

REVIEWING THE REAL MODE

Protected mode has its roots in the 8086 processor, the ancestor of the 32-bit 80386. The 8086, although a 16-bit CPU, provides a clever mechanism to access up to 1MB of physical (real) memory: real mode. This addressing mode relies on a combination of segment and offset registers to address bytes in memory (instruction or data). Each instruction uses one of the four segment registers available, either implicitly or explicitly. Address calculation is done by shifting a segment register by four (multiplying by 16) and adding one of the nine general registers, typically the one specified in the instruction (see Figure 1).



The result is a 20-bit address, providing 1MB of address space and using 16-bit registers. The carry bit (bit 20) is discarded.

Since a single segment register allows accessing 64K, multiple segments are required to access more memory. Most developers involved in Intel application development have heard of the various memory models that were popular not so long ago: tiny, small, medium, compact, large, and huge. These models proposed various segment combinations in order to overcome the 16-bit limitation when accessing code and data beyond 64K.

To push the 1MB limitation further, some complex schemes were introduced, such as the expanded and extended memory. These schemes helped, but they also increased memory management complexity and consequently, introduced overhead.

Compilers, linkers, and operating system loaders had the responsibility of assigning proper values to segment registers, to free the application developers from doing so. System programmers, writing programs mainly in assembly, were not so lucky and had

to cope with this complex scheme. The source of all this complexity was the infamous 64K limitation due to the 16-bit nature of the CPU.

INTRODUCING THE PROTECTED MODE

In 1986, with the advent of the Intel 80386, things really started to change. For one, this processor is a real 32-bit processor. The main advantages of 32-bit programming over 16-bit and 8-bit programming are speed and simplicity. Instructions themselves are usually faster and a single instruction (string and memory operations, arithmetic, and so forth) can work on 32 bits at the same speed as two 16-bit instructions or four 8-bit instructions. Reducing the number of instructions reduces program sizes and speeds up execution, because fewer instructions have to be fetched and decoded. Smaller size and faster execution are always welcome in embedded systems.

Second, this CPU's memory management unit (MMU) introduces a new addressing mode over real mode called protected mode. This mode offers a high degree of flexibility, making possible very large 4GB flat address space per task or per application (no segment) and up to 64 terabytes (that's 64,000,000,000,000 bytes) of virtual memory! This mode also adds some protection in order to run the software that needs it, such as Unix-like systems have. Advanced memory concepts and protection will be covered in subsequent articles.

In protected mode, the segment registers are indexes into special tables, all initialized and maintained by the operating system, but interpreted by the CPU.

There are three types of tables, all located either in RAM or ROM:

- The Global Descriptor Table, GDT: unique, always accessible
- The Local Descriptor Table, LDT: usually one per task. Zero, one, or many may be present in the system, but only one, if any, is active at all times
- The Interrupt Descriptor Table, IDT: used when interrupts are raised Each table contains a variable number of descriptors and an 8-byte data structure that describes a memory region with the following attributes. See figure 2
- The base address in memory (32-bit)
- The limit (20-bit), expressed either in 4K or 1-byte units
- Control bits: the granularity bit (limit's unit), present bit (useful with swapping), and two protection bits
- The descriptor type, one of the 16 supported, among them: executable, read-only code segment; data segment; stack segment; call, trap, or interrupt gate; task state segment, and others

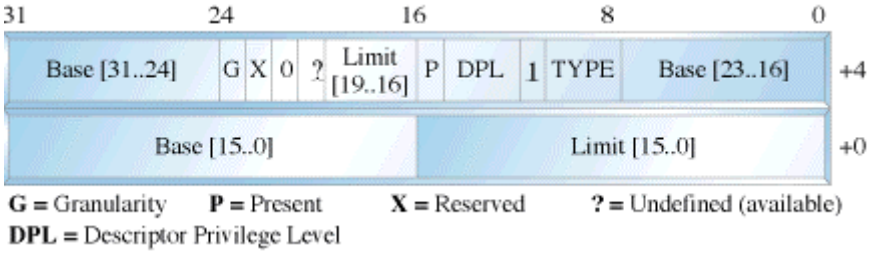


Figure 2

Segment registers are selectors (indexes) into either the GDT or the LDT (the IDT entries are only used when an interrupt is raised). A selector (such as a segment register) contains a 13-bit index, a 1-bit table identification (GDT/LDT), and a 2-bit protection level.

See Figure 3.

Figure 3

A logical address, as used by a program, is still the combination of a segment and a general register, or more precisely, a 16-bit selector and a 16-bit or 32-bit offset. The selector identifies a descriptor, which in turn provides a 32-bit base address, to which the

offset is added, forming a final linear 32-bit address as seen in Figure 4. This 32-bit addressing supports up to 4GB (2^{32}) of memory.

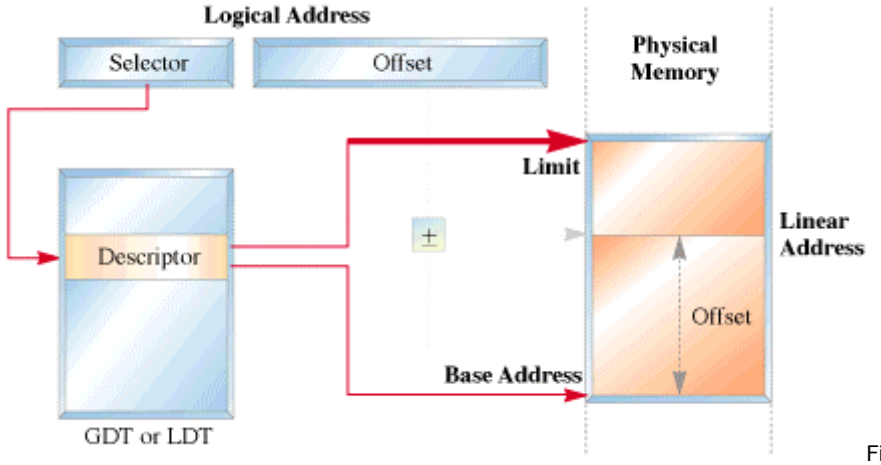


Figure 4

All memory accesses within the segment can be done with the offset only, simplifying program coding. This address calculation provides many advantages:

- Because segment registers cover up to 4GB individually, they don't have to be constantly reloaded, even with huge data structures, reducing complexity and increasing speed
- Offsets always start at zero, independently of the segment's location in physical memory, making it easier to debug—the addresses (offsets, for example) never change. Segments can be moved in physical memory without affecting the applications that use them
- An offset must be within the segment's limit; if it isn't, an exception is raised and the operating system, which typically catches it, may stop the faulty application. This feature prevents incorrect memory access, such as jumping outside the code segment or accessing out-of-segment data
- Segments are protected against undesired access, thanks to their descriptors. For instance, an application cannot write into a code segment, which is read-only. Another similar example is to prevent executing from a data segment
- This addressing mode provides a phenomenal virtual address space. Because a selector's index is a 13-bit value, the GDT and LDT tables are limited to 8,192 descriptors (2^{13}). One single descriptor can cover up to 4GB (with a base address of zero, a limit of 1MB (FFFFFh), and the granularity bit set, making the limit a 4K-unit value (4K x 1MB = 4GB)). Considering that the GDT and the active LDT together have a maximum of 16,384 descriptors, the total virtual address space is 64 terabytes (16K * 4GB). Although this is astronomical, one must realize that segmentation always produces a 32-bit linear address, limiting the physical address space to 4GB, still quite sufficient
- One embedded operating system may use a few segments, whereas another may use hundreds of them, as illustrated in Figure 5. In (a), an embedded application can reside in the operating system with a single segment covering 4GB. In (b), the same application might own its proper segments, all distinct from the operating system segments. These various approaches can be justified depending on the system constraints.

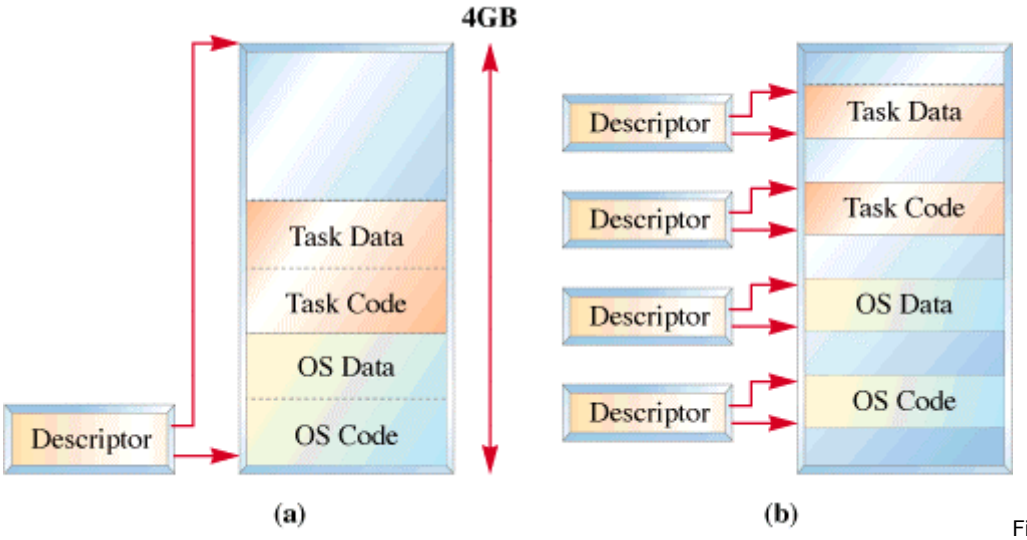


Figure 5

MIXING 16-BIT AND 32-BIT CODE

Protected mode is not synonymous with 32-bit. The Intel CPUs, in protected mode, support 16-bit and 32-bit segments. This makes them ideal CPUs to run legacy 16-bit applications as well as new 32-bit systems.

Most x86 assemblers support some directives to indicate whether a specific segment of code will be executed in 16-bit mode (the USE16 directive) or in 32-bit (USE32). The assembler will generate the appropriate code, but it's up to the operating system to load the task adequately—that is, to ensure the USE16 segments are run in 16-bit mode and USE32 segments are run in 32-bit mode.

By comparing disassembled 16-bit with 32-bit code, one notices many similarities, as shown in listing 1. For instance, the 16-bit instruction xor ax,ax (line 2) and the 32-bit instruction xor eax,eax (line 7) produce identical opcodes (33h C0h). How does the CPU make the difference between the 16-bit AX and 32-bit EAX registers? This has to do with the current segment mode. Whenever the code segment—the CS segment register—is loaded with a selector, the CPU loads the descriptor into an internal, inaccessible register (reserved for the CPU only) and analyzes its type. One bit in the descriptor determines whether this is a 16-bit or 32-bit segment. If this is a 16-bit segment, the opcodes 33h C0h mean xor ax,ax; if this is a 32-bit segment, they mean xor eax,eax. In a 16-bit code segment, opcodes work with 16-bit operands and addresses; in a 32-bit code segment, opcodes work on 32-bit operands and addresses. Consequently, the CPU has only one instruction set and it works for both 16-bit and 32-bit modes.

To provide maximum flexibility, the opcodes 66h and 67h respectively override the operand and address size. For example, in a 16-bit code segment, 66h 33h C0h (line 3) would work on 32-bit EAX whereas the same instruction, in a 32-bit code segment (line 8), would affect only the lower 16-bit AX.

This condition is made possible because the CPU really has one and only one bank of 32-bit general registers. It's the current CPU mode (either real mode, 16-bit protected mode, or 32-bit protected mode) that indicates whether 16-bit or 32-bit operands and addresses are used and how they are used. Typically, the size override opcodes are used in 16-bit code to access 32-bit values. They are rarely used in 32-bit application code.

Serious incompatibilities between 16-bit and 32-bit code still exist, especially regarding the address mode encoding (the 32-bit mode supports more addressing modes) and the addresses themselves (encoded over 16 or 32 bits). The 16-bit instruction lea ax,MyVar (line 2) has no direct opcode equivalent in a 32-bit segment, even with size override opcodes (lines 6 and 7).

Mixing 16-bit and 32-bit code is more than often an issue and should be avoided whenever possible. There is, however, one case in which this mix can't be avoided: upon starting up, the CPU starts running in real mode. A 32-bit system will have to switch into

32-bit protected mode and will have to mix at some point or another some 16-bit code (system initialization) and 32-bit code (rest of the system).
If both your assembler and linker accept 16-bit and 32-bit segments, you can simply put the real mode code into a USE16 segment and the 32-bit protected mode code into a USE32 segment. Once the 16-bit code is ready to go 32-bit, it simply jumps into the USE32 segment. The two segments don't have to be part of the same program, although having two programs doesn't always make things simpler. I prefer to keep the initialization in one single source file.
Again, this solution only works if your tools fully support a 16-bit and 32-bit code mix. Unfortunately, some recent tools only support flat memory model and do not fully support 16-bit code. For instance, some linkers will not permit a jump from a 16-bit segment into a 32-bit one. However, switching into 32-bit protected mode requires exactly that. Thus, the issue becomes how to write 16-bit instructions with tools that only support 32-bit programming.
The solution is to write the 16-bit code in 32-bit segments. Now this is not trivial, because mnemonics will be assembled as 32-bit instructions, and they will cause problems when they're executed in real mode. Using size override opcodes may not entirely resolve the issue because some instructions are simply incompatible. Simply put, 16-bit instructions cannot be written as such in a 32-bit segment. But there's still a solution.
By disassembling the real-mode instructions from a 16-bit, you can directly put the resulting opcodes in a 32-bit segment, using assembly directives such as DB, DW (which allow you to enter values in numerical form). This is really directly encoding 16-bit instructions. I agree that this isn't a high-tech solution, although the use of macros can maintain the code's readability. Also, the 16-bit portion typically consists of a few instructions and it affects little code overall. Furthermore, these few instructions are unlikely to change. This method is demonstrated a little bit further.
There are other alternatives. If you are designing a 16-bit system, you are free to stay in 16-bit as long as you want, by writing everything in a USE16 segment and either far jumping into a 16-bit segment or not jumping at all (since the CPU maintains a valid code segment after the protected mode is activated, as we will see later). This method obviously requires tools that support 16-bit programming.

ACTIVATING THE PROTECTED MODE
Now let's take a look at an example that shows how to activate protected mode. This example starts in real mode and switches into protected mode in order to execute 32-bit code in a flat memory model. The purpose of this example is to demonstrate basic protected mode concepts; if you're looking for ready-to-run examples, be sure to take a look at those at *www.embedded.com/code* .
This example assumes that after the CPU has been reset, some system tests have successfully run, the interrupts have been disabled, and the CPU is still running in real mode. The BIOS, if any, is ignored because it works only in real mode and is irrelevant in protected mode. The code can be run anywhere in the first 1MB of memory.
I used Microsoft's Macro Assembler (MASM) v. 6.11, the latest revision. Also, instead of using the provided linker, I used Microsoft Visual C++ v. 5.0's linker, which is the latest incremental linker from Microsoft. This linker generates COFF file executable, which is the standard under Windows NT, (MASM's linker outputs less popular OMF files). The incremental linker has one drawback: it doesn't fully support 16-bit segments. Consequent-ly, 16-bit instructions must be directly encoded, as I explained earlier.
The shortest way to execute 32-bit code is to load the GDT, activate the protected mode, and jump into a 32-bit segment. This order is not strict, though. For instance, you may first activate the protected mode, load the GDT, and make the jump. Either way is the same.
The GDT in Listing 3 is already constructed (lines 97 to 121). Even if you intend to dynamically add descriptors in it later, you can initially have it with a few static descriptors right from the start. In the example, the GDT occupies 24 bytes and contains three entries:

- Entry 0 is null. This entry cannot be referred to; segment registers can be initialized to zero (thus pointing to it), but using them raises an exception. This feature is aimed at identifying NULL far pointer references. Nevertheless, it could contain some data because the descriptor is never used by the CPU. In the example, it simply contains zero

- Entry 1 (Selector 08h) is used for kernel code, with a base of zero, a limit of FFFFFh, with the granularity bit set (making the limit 4GB), and the type set to executable, read-only code

- Entry 2 (Selector 10h) is used for kernel data, also with a base of zero, a limit of FFFFFh, granularity bit set but with a type of writable segment. This data segment overlaps the code segment. Together, they provide a flat memory address space.

The GDT is loaded by initializing the GDTR register with the GDT base address and size, both stored in a 6-byte data structure (line 61). The GDTR register is normally loaded by executing the 16-bit instruction lgdt fword ptr address. But this instruction cannot be written as such because the resulting 32-bit opcodes will not work in real mode (the address mode will be incorrect). To make things worse, addresses must be expressed as 32-bit values (a linker constraint), whereas 16-bit values are normally expected in real mode. To overcome this problem I used the 16-bit instructions mov ebx,address and lgdt fword ptr [bx], encoded in the LGDT32 macro (lines 17 to 25), and called from line 47. The address is specified as a 32-bit value, although only the lowest 16-bit portion is used. But above all, it properly loads the GDT register while the CPU is running in real mode.
With the GDT register set, protected mode is activated by setting bit #0 in the CR0 register (lines 49 to 51). CR0 is a control register that controls segmentation and paging, among other things. CR0 can only be read from or written to by using register operands (no memory nor immediate operands). The example uses the AX register, although the opcodes will use EAX when executed in real mode. As soon as the bit is set in CR0, protected mode kicks in and the CPU starts executing 16-bit instructions, but in protected mode (segment registers become indexes into a table).
The content of all segment registers is unknown at this point. However, it is guaranteed that they can still be used to access subsequent instructions or data. And immediately after the protected mode is activated, the CPU's instruction queue must be flushed because it contains pre-fetched real mode instructions, no longer valid in protected mode. The queue can be flushed by executing a jump to the next instruction (line 52).
The last thing to do in 16-bit is to switch to 32-bit. This step is achieved by loading the code segment register (CS) with a selector referring to a 32-bit executable code descriptor. The second entry in the GDT is such a descriptor. At line 57, the FJMP32 jump macro (lines 27 to 32) is executed with the selector 08h and the Start32 offset. Because the descriptor contains a base address of zero, the offset simply has to be the physical location of the first instruction to execute in 32-bit mode. In this case, this instruction is at Start32 (line 65). The entry into the 32-bit mode marks the end of the 16-bit code (as well as the tricky encoded instructions).
Once in 32-bit mode, the best thing to do is initialize the data registers (DS, ES, FS, GS) and the stack register (SS) (lines 73 to 78). Note that there are 16-bit and 32-bit stack segments: the size determines how many bytes (two or four) a normal push saves on the stack (the push and pop opcodes are identical in 16-bit and 32-bit). The example uses the third GDT entry as a combined data and 32-bit stack segment (selector 10h). Finally, ESP (the stack pointer) is set to an arbitrary top of stack.
And that's it—the code is running in 32-bit protected mode, in a flat address space of 4GB! A complete system would have to continue its initialization, such as loading and initializing the Interrupt Descriptor Table (IDT), set-up some hardware, and so on. These issues are beyond the scope of the article, although they are fully addressed in the examples that can be downloaded from the Web site. The next articles will explore segmentation (including protection) and paging in detail.

Listings

Listing 1
Sixteen-bit and 32-bit code segments. Opcodes are shown on the left.

```
1. _TEXT              SEGMENT PARA USE16 PUBLIC 'CODE'
2. 33 C0              xor     ax,ax
3. 66 33 C0           xor     eax,eax
4. _TEXT              ENDS
5.
6. _TEXT              SEGMENT PARA USE32 PUBLIC 'CODE'
7. 33 C0              xor     eax,eax
8. 66 33 C0           xor     ax,ax
9. _TEXT              ENDS
```

Listing 2
Although many instructions are identical in 16-bit and 32-bit, addressing modes and addresses always generate different opcodes, making the code incompatible.

```
1. _TEXT                            SEGMENT PARA USE16 PUBLIC 'CODE'
2. 8D 06 0024                 lea   ax,MyVar
3. _TEXT                            ENDS
4.
5. _TEXT                            SEGMENT PARA USE32 PUBLIC 'CODE'
6. 66 8D 05 00000024   lea    ax,MyVar
7. 8D 05 00000024             lea   eax,MyVar
8. _TEXT                            ENDS
```

Listing 3
Switching from 16-bit real mode to 32-bit protected mode.

```
1. ; ProtMode.asm
2. ; Copyright (C) 1998, Jean L. Gareau
3. ;
4. ; This program demonstrates how to switch from 16-bit real mode into
5. ; 32-bit protected mode. Some real mode instructions are implemented
6. ; with macros in order for them to use 32-bit operands.
7. ;
8. ; This program has been assembled with MASM 6.11:
9. ;    C:\>ML ProtMode32.asm
10.
11.             .386P                   ; Use 386+ privileged instructions
12.
13. ;---------------------------------------------------------------------;
14. ; Macros (to use 32-bit instructions while in real mode)
15. ;---------------------------------------------------------------------;
16.
17. LGDT32      MACRO   Addr                    ; 32-bit LGDT Macro in 16-bit
18.             DB              66h             ; 32-bit operand override
19.             DB              8Dh             ; lea (e)bx,Addr
20.             DB              1Eh
21.             DD              Addr
22.             DB              0Fh             ; lgdt fword ptr [bx]
23.             DB              01h
24.             DB              17h
25. ENDM
26.
27. FJMP32      MACRO   Selector,Offset  ; 32-bit Far Jump Macro in 16-bit

28.             DB              66h             ; 32-bit operand override
29.             DB              0EAh            ; far jump
30.             DD              Offset          ; 32-bit offset
31.             DW              Selector        ; 16-bit selector
32. ENDM
33.
34.             PUBLIC  _EntryPoint             ; The linker needs it.
35.
36. _TEXT       SEGMENT PARA USE32 PUBLIC 'CODE'
37.             ASSUME  CS:_TEXT
38.
39.             ORG             5000h   ; => Depends on code location.
40.
41. ;---------------------------------------------------------------------;
42. ; Entry Point. The CPU is executing in 16-bit real mode.
43. ;---------------------------------------------------------------------;
44.
45. _EntryPoint:
46.
47.             LGDT32  fword ptr GdtDesc       ; Load GDT descriptor
48.
49.             mov             eax,cr0     ; Get control register 0
50.             or              ax,1        ; Set PE bit (bit #0) in (e)ax
51.             mov             cr0,eax     ; Activate protected mode!
52.             jmp             $+2         ; To flush the instruction queue.
53.
54. ; The CPU is now executing in 16-bit protected mode. Make a far jump in
54b. ;  order to
55. ; load CS with a selector to a 32-bit executable code descriptor.
56.
```

```
57.             FJMP32  08h,Start32             ; Jump to Start32 (below)
58.
59. ; This point is never reached. Data follow.
60.
61. GdtDesc:                                    ; GDT descriptor
62.             dw              GDT_SIZE - 1    ; GDT limit
63.             dd              Gdt             ; GDT base address (below)
64.
65. Start32:
66.
67. ;---------------------------------------------------------------------;
68. ; The CPU is now executing in 32-bit protected mode.
69. ;---------------------------------------------------------------------;
70.
71. ; Initialize all segment registers to 10h (entry #2 in the GDT)
72.
73.             mov             ax,10h                  ; entry #2 in GDT
74.             mov             ds,ax                   ; ds = 10h
75.             mov             es,ax                   ; es = 10h
76.             mov             fs,ax                   ; fs = 10h
77.             mov             gs,ax                   ; gs = 10h
78.             mov             ss,ax                   ; ss = 10h
79.
80. ; Set the top of stack to allow stack operations.
81.
82.             mov             esp,8000h       ; arbitrary top of stack
8
84. ; Other initialization instructions come here.
85. ;           ...
86.
87. ; This point is never reached. Data follow.
88.
89. ;---------------------------------------------------------------------;
90. ; GDT
91. ;---------------------------------------------------------------------;
92.
93. ; Global Descriptor Table (GDT) (faster accessed if aligned on 4).
94.
95.             ALIGN   4
96.
97. Gdt:
98.
99. ; GDT[0]: Null entry, never used.
100.
101.            dd              0
102.            dd              0
103.
104.;GDT[1]:Executable, read-only code, base address of 0, limit of FFFFFh,
105. ;  granularity bit (G) set (making the limit 4GB)
106.
107.            dw              0FFFFh                  ; Limit[15..0]
108.            dw              0000h                   ; Base[15..0]
109.            db              00h                     ; Base[23..16]
110.            db              10011010b ;P(1) DPL(00) S(1) 1 C(0) R(1) A(0)
111.            db              11001111b       ; G(1) D(1) 0 0 Limit[19..16]
112.            db              00h                     ; Base[31..24]
113.
114. ; GDT[2]: Writable data segment, covering the save address space than
GDT[1].
115.
116.            dw              0FFFFh                  ; Limit[15..0]
```

```
117.            dw              0000h                   ; Base[15..0]
118.            db              00h                     ; Base[23..16]
119.            db              10010010b ;P(1) DPL(00) S(1) 0 E(0) W(1) A(0)
120.            db              11001111b       ; G(1) B(1) 0 0 Limit[19..16]
121.            db              00h                     ; Base[31..24]
122.
123. GDT_SIZE   EQU             $ - offset Gdt          ; Size, in bytes
124.
125. _TEXT      ENDS
126.            END
```

Listings by:
Jean Gareau graduated from the Polytechnic School of the University of Montreal in 1992 with an M.S. in electrical engineering. He has been involved since 1989 in the development of operating systems, system tools, and large commercial applications. He can be reached at jeangareau@yahoo. com.

REFERENCES

1. Ganssle, Jack G., "Future Challenges," *Embedded Systems Programming 1997 Buyer's Guide* , p. 7.
2. Labrosse, Jean J. m *C/OS The Real-Time Kernel* . Lawrence, KS: R&D Publications, Fourth Printing, 1992.

Other Sources

80386 Programmer's Reference Manual , Order Number 230985-001, Chandler, AZ: Intel Corp., 1987.
Reference: Microsoft MASM 6.11 , Document Number DB35749-1292, Redmond, WA: Microsoft Corp., 1992.
Programmer's Guide: Microsoft MASM 6.11 , Document Number DB35747-1292, Redmond, WA: Microsoft Corp., 1992.

Advanced Embedded X86 Programming: Protection and Segmentation

In this part of the tutorial the author explains how protection and segmentation are implemented in protected mode.

This article is the second in a series of three describing protected mode features of the 386 up to the Pentium. Because this family of CPUs is extremely popular and is affecting the embedded industry, RTOSes and embedded applications can be updated to take advantage of its 32-bit programming capabilities and larger, simpler memory models. Development tools also obtain some benefits because popular memory models, such as the flat memory model, are simpler to support. The previous article introduced the 32-bit protected mode, in which each segment register is an index to a table of descriptors, each descriptor describing a memory segment by its base address, limit, type, and protection fields. A 32-bit linear address is produced from a segment/offset logical address by adding the offset to the base address found in the descriptor that's pointed to by the segment. That article also demonstrated how to switch from real mode to protected mode.

This article explains how protection and segmentation are implemented in protected mode to design robust and reliable systems. These qualities are becoming more important as some embedded systems now include multiple tasks that run concurrently, such as an embedded Java Virtual Machine that supports concurrent Java applets. Whereas the lack of protection allows tasks to access, corrupt, and destroy anything from other tasks' data to the operating system (OS) internal structures, protection permits the building of robust and powerful applications by restricting them to their own code and data. Another advantage offered by protection is the ability for the OS to remove a faulty task (such as copying memory outside its address space—a typical problem with memcpy() in C). For example, when a task commits a fault, the CPU can notify the OS, which can then halt the task before any damage is done; the other tasks continue undisturbed. Protection also simplifies task development—when an error is detected, protection helps to pinpoint the exact problem where it happened, so developers don't have to search for it. In other words, protection simplifies building and deploying safe and secure embedded tasks that can work together but not harm each other.
Complete examples, fully documented and tested, that implement various kernel designs can be downloaded from *www.embedded.com/code* . These examples demonstrate the concepts I've explained in this series, including a port of mC/OS to the protected mode. [1] The source code is provided as ready-to-run executables and additional tools. These examples provide an excellent start to help you improve your applications or even implement your own system.

WHAT PROTECTION?
The protection we're talking about here is three-fold: 1) to prevent a task from executing privileged instructions, such as setting or clearing the interrupt flag; 2) to prevent a task from accessing another task's code and data; and 3) to prevent tasks from calling privileged kernel code in an unordered fashion or corrupting privileged kernel data. This protection doesn't deal with user authentication, since this concept is implemented in the OS itself, not in the CPU. All protection features are implemented in the CPU and activated by the OS, freeing application developers from having to worry about protection.
In the x86, protection is a feature associated with segments, and automatically kicks in when the CPU executes in protected mode.
Whenever a segment register is referred to, the CPU accesses the related descriptor and analyzes its control bits. If the operation doesn't concur with these bits, the processor raises an exception, which is typically caught by the OS—resulting in the application

being shut down. Examples of this would be to write in a code segment (code segments cannot normally be altered), jump into a data segment (data isn't executable), and so on.
Another protection check involves checking the offset used in the address calculation against the segment's limit. If an operation tries to address beyond the limit, an exception is raised. The most common example is the use of an incorrect pointer or an invalid jump. This limit check feature is useful because it constrains a task to its own segments. Many tasks might be in memory at once, but if they all have their own separate segments, they can't see each other, and therefore can't alter one another.
Each task typically requires at least two descriptors, one for code access and one for data access. Code segments are read-only and can't be used to modify data—hence the need for a second segment for data access. The descriptors normally don't cover the same address space (i.e., they don't overlap), in order to maintain the initial protection (not to overwrite code, and so forth). Through its own descriptors, a task is restricted to its code and data. The flat memory model's segments *do* overlap, but this model is normally used in conjunction with paging, the subject of part three of this series of articles.
The previous article explained that descriptors were kept in either the global descriptor table (GDT) or local descriptor table (LDT). Let's forget for a moment about the LDT and suppose that all the code and data descriptors of all tasks are kept in the GDT. One problem that arises at this point is that any task can load another task's data descriptor from the GDT and alter this other task's data segment. The problem exists because the GDT is unique and a task can theoretically access any descriptors in it. Providing distinct address space per task isn't sufficient in itself; the solution comes through the use of the LDTs, task state segments (TSSes), privilege levels, and gate descriptors, all explained in the following sections.
LDTs, like the GDT and the interrupt descriptor table (IDT), are built by the OS, not by the task. A given LDT usually contains a task's code and data descriptors, and is built when the task is loaded in memory. If the task is going to be in ROM, the LDT can be hard-coded and ROMable. Although a system may have more than one LDT, only one is active at a time (see Figure 6), pointed to by the LDTR register. Note that no LDTs might be active at all, if none are referred to.

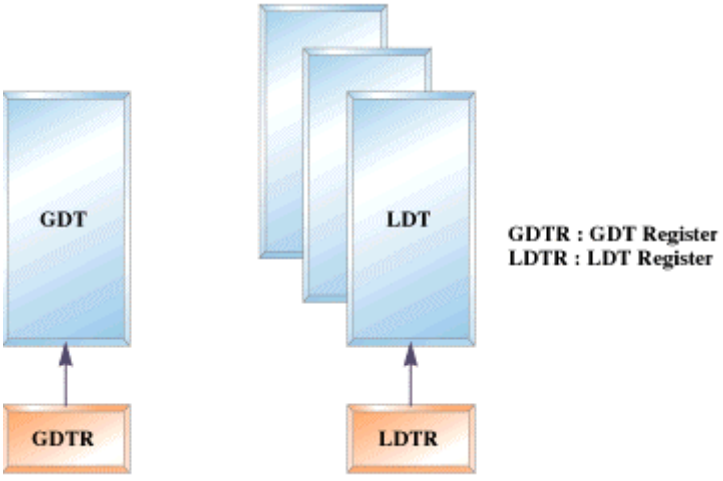


Figure 6

An LDT is described by a descriptor whose base address is the LDT address in memory. The limit is the LDT size, as it's useful to limit the number of entries in it,
In a system where each task has its own LDT, you can keep the LDT selector in the task control block (or a similar structure) to identify this LDT as the current one when the task is selected to run. Such a system also provides a good deal of protection, because each task sees either the GDT or its LDT, but not the other tasks' LDTs. Consequently, a task can't alter or even look at another task's code and data—a must with segmentation.
If a system is designed to provide such isolation among the tasks, the OS must provide primitives to transfer information among tasks (to send and receive messages, for

instance); this is something tasks can't do by themselves because they're restricted to their own address space.

PRIVILEGE LEVELS

Despite the LDTs, a task can still refer to any GDT descriptor and alter some data. To prevent this from happening, privilege levels are introduced. The Intel CPUs support four levels: 0 (most privileged) to 3 (least privileged). Level 0 allows the execution of privileged instructions (such as set or clear interrupts, accessing I/O ports, loading the GDT, LDT, or IDT, and the like), whereas this is strictly forbidden in any other level. Operating systems must run at level 0 to have no restriction at all, whereas you must decide whether device drivers, OS extensions, and application tasks run at either 0, 1, 2, or 3. Using level 0 for all system software and 3 for application software is very common, while simply ignoring levels 1 and 2. Such tasks can neither execute privilege instructions nor alter the state of the system.

A privilege level is always associated with current executing code. When entering the protected mode, the current privilege level (CPL) is 0 (the highest) because OS code is expected to be running. All descriptors contain a descriptor privilege level (DPL), which is a two-bit value (zero to three) identifying the privilege of the related segment. The DPL has a different meaning depending on the segment type (code or data).

When the code segment register—the CS register—is loaded with a valid code selector (via a jump, a call, or returning from a function or an interrupt), the CPU examines the descriptor and the DPL becomes the CPL, as seen in Figure 7.

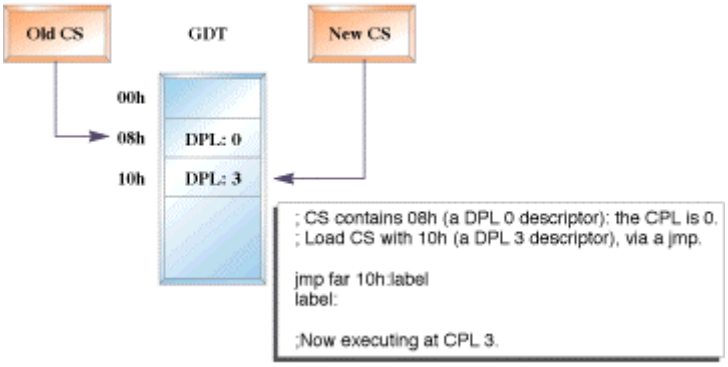


Figure 7

For instance, once the OS initialization (running at CPL 0) is completed, the first task is executed by loading CS with a selector referring to the tasks' code descriptor with a DPL of 3; consequently, the task starts running at CPL 3. But there is a trick: when loading CS, the CPL can never become more privileged; it has to stay at the same or a lesser privilege. Thus, a CPL 3 task cannot load CS with a selector referring to a descriptor with a DPL of 0, 1, or 2—doing so will raise an exception. The OS would typically catch that exception, analyze it, delete the faulty task, and reschedule another one.

When a data segment register is referred to, the DPL of the related descriptor indicates the minimum CPL required to access it—the CPL must have the same or a higher privilege than the DPL. Thus, a CPL 2 task cannot refer to a data descriptor carrying a DPL of 0 or 1; it can only access data descriptors with a DPL of 2 or 3.

The OS, by carefully setting the DPL of all GDT and LDT descriptors, prevents nonauthorized tasks from accessing sensitive or protected code and data. For instance, all OS descriptors are set with a DPL of 0, whereas all task descriptors are marked with a DPL of 3. Consequently, tasks cannot directly access OS code and data. As far as the applications are concerned, they only see their code and data, nothing else.

TASK STATE SEGMENTS

Before we explore privilege levels in detail, we must introduce task state segments (TSS). A TSS is a placeholder for all the registers of a task when that task doesn't run. Like LDTs, only one TSS is active at all times and it is interpreted at some point by the CPU. It is also described by a descriptor that indicates the base address, the size (which may vary because extra data can be stored in each of them), the protection, and the type, which in this case is a TSS. The TR register contains the selector of the active TSS. Having one TSS per task in a segmented system is common. In this case, TSS descriptors are usually kept in the GDT, with a DPL of 0 to prevent a task (with a CPL of 1, 2, or 3) from accessing them. A task's TSS selector can also be stored in a task control block for quick reference. The operating system can indicate which TSS is active by executing the ltr instruction (load task register).

TSSes are special: a far jump to a TSS selector (the offset is ignored) makes a complete context switch from the current task to the task referred to by the selected TSS. That switch not only saves and reloads the task registers, but it manages the segment registers, the current LDT selector, the current TSS selector, and so on, all with a single instruction. Keeping all LDT and TSS descriptors in the GDT is best, to ensure their accessibility during the switch.

MIXING PRIVILEGE LEVELS

Maximum protection is achieved by mixing privilege levels: the OS is given full privileges, while the tasks receive no special privileges. Luckily, the x86 has much to offer in order to mix privileges.

If you want to execute higher-privileged code, such as directly calling an OS function at CPL 0 from a CPL 3 task, you can use call gates (see Figure 8).

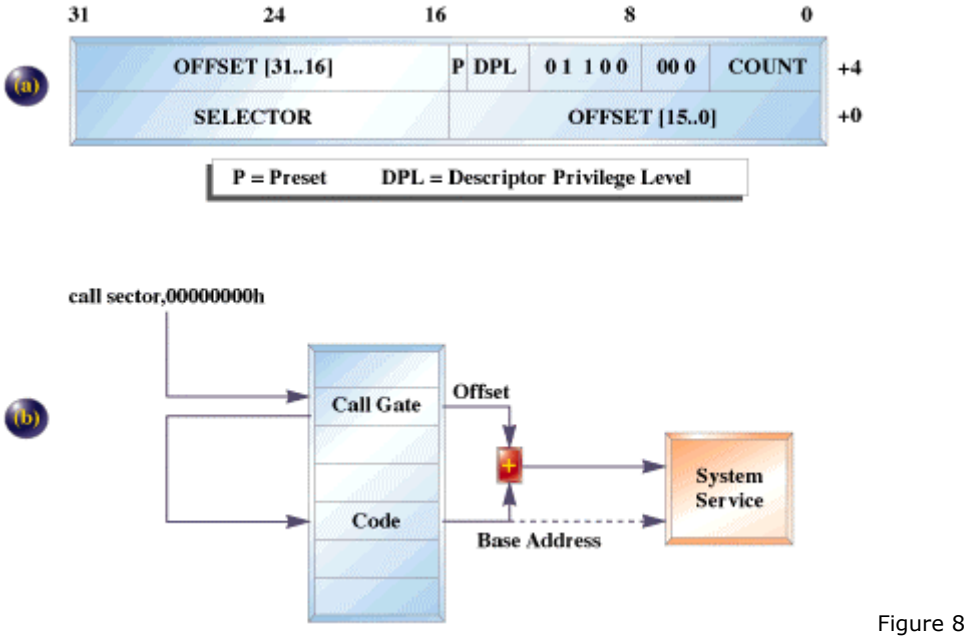


Figure 8

Call gates are a special type of descriptor and can reside in the GDT (making them sharable among all tasks) or a task's LDT (making them private to that task). A call gate is no more than an indirect, controlled call to a more privileged function, typically an OS service. The call gate contains a code selector and the address of the function to call within that selector. The code selector usually has a higher privilege, allowing the system service to run with adequate privileges. Call gates are initialized and maintained by the OSes, but used by the tasks.

Call gates are accessed via a far call (a selector/offset combination), though only the selector is meaningful (the offset is discarded). Call gates can be hidden in a normal function (such as open()), making them "invisible" to the application programmers.

The CPU ensures that the current task has enough privilege to use the call gate. For instance, if the call gate has a DPL of 2, only tasks executing at CPL 0, 1, or 2 can use it;

tasks running at CPL 3 are excluded. But it is common to set all call gates' DPL to 3, to make them available to all tasks. Also, the target code segment's DPL must have the same or a higher privilege than the CPL. For example, if a task running at CPL 2 uses a call gate that refers to a segment at CPL 3, a fault is triggered. Call gates are only used to increase privilege levels, not to decrease them; otherwise, upon returning, there would be an uncontrolled privilege increase (which would be disastrous if the return address would have been altered by the task). For that same reason, when the system service terminates, various checks are performed to ensure that the control is returned to a code segment of the same or lesser privilege. Note that executing a far call to a less-privileged segment is possible, as long as it is a conforming segment. A segment is conforming when a special control bit is set in its descriptor. Such a segment conforms to its caller in that it executes under the caller's CPL. For instance, if the current task runs at a CPL of 2, and calls a function in a conforming segment of DPL 3, that function will also run at CPL 2. Thus, calling a conforming segment doesn't alter the calling task's CPL. Conforming segments are a useful way to implement system libraries callable by any task, regardless of its privilege. In fact, there is no other way to call less-privileged segments. However, conforming segements are still quite rare, since libraries (such as the C library) are usually bound to applications at link time, not run time.
But a call gate isn't enough to ensure a successful execution—enough stack space must be provided for the OS service to run. Because the calling task may have very little stack space, the call gate will perform a stack switch if the privilege level is increased. The TSS is important here because, in addition to the task's registers, it holds stack pointers for privilege levels 0, 1, and 2, all initialized by the OS. Here's an example of how it works: a CPL 3 task uses a call gate to execute a system service at DPL 0; the task's TSS (the current TSS) is looked up to get the privilege 0 stack pointer (selector/offset), and this value becomes the effective stack. The original task's stack pointer (selector/offset) is pushed on the new stack to have a link back to it, as shown in Figure 9.

Figure 9

In addition to the stack switch, up to 32 double-word parameters can be copied from the task's stack to the new stack, which is a convenient feature. The number of parameters per call gate is fixed—call gates do not support a variable number of parameters.
The alternative to using call gates is to use a trap interface, which consists of calling OS functions by raising software interrupts. A trap interface involves the interrupt table (IDT), which can contain three types of descriptors:

- Interrupt gate, which refers to a specific function, usually in the kernel
- Trap gate, which is similar to an interrupt gate
- Task gate, which points to a TSS descriptor

Like call gates, invoking a method through the IDT is a way to increase the CPL. Each IDT descriptor, like any other descriptor, contains a DPL, usually set to 0 by the OS. Such a DPL prevents unprivileged tasks from directly triggering the interrupt (via the int instruction). Note that some IDT descriptors may have a lower DPL, making them callable by the tasks, to implement system calls. These software interrupts or traps may also be hidden in a library function (such as close()), to hide them from the application programmers.

Whenever a hardware interrupt is raised or a valid software interrupt is encountered, the related descriptor of the IDT is analyzed and the CPL set to the descriptor's DPL (as seen in Figure 10).

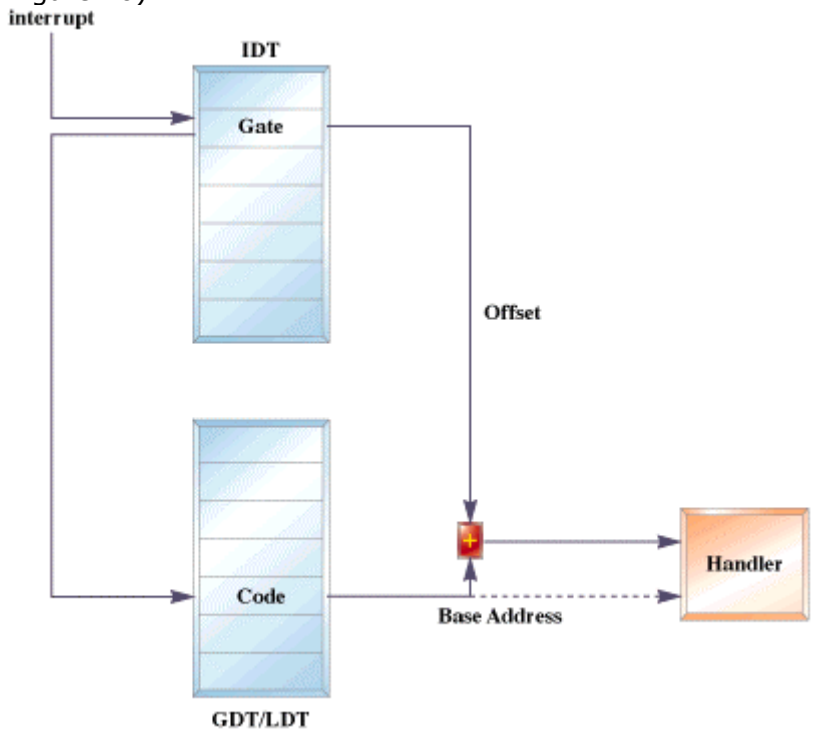


Figure 10

For an interrupt gate, execution starts in the address found in the descriptor (typically an interrupt handler in the OS) with the interrupts disabled; when the handler terminates, extra protection checks occur to ensure a proper return to the caller. Trap gates are almost identical—the same processing happens, but with the interrupts enabled. With a task gate, a context switch occurs (as described earlier with the TSS). Note that task gates in the IDT are not a convenient way to implement multitask, because context switches normally occur under OS conditions (time slice expired, system calls that makes a higher-privileged task ready, and the like), rather than raised interrupts. Nevertheless, they can be a useful way to invoke special tasks, such as a debugger.

The CPU invokes the handlers described by the interrupt or trap gates in a way similar to call gates: if the privilege is increased, a stack switch occurs using the TSS to get the new stack pointer. As opposed to a call gate, though, no parameter can be copied. If parameters are needed, they can be passed via the registers or they may be traced back via the task's stack pointer, which is on the new stack.

Choosing between call gates vs. interrupt/trap gates to execute an OS service depends on how many system calls you have and whether they require some arguments or not. If you have many system calls, an interrupt/trap gate is better because it offers a single entry point in the kernel; however, a register must be set to identify the service called. On the other hand, because a call gate refers to one function, having many system calls implies many call gates. Since high-end systems might have hundreds of calls, call gates might be tough to maintain. Moreover, if they are placed in the LDT (to prevent some specific tasks from using it), things can get very complicated.

If the task passes parameters, call gates allow you to transfer them into the more privileged stack, from where the system call can access them as local parameters (easy). Via the interrupt/trap interface, you might have to trace back the calling stack (which is just annoying).

If you prefer call gates (because of the fixed parameter transfer facility) and you have hundreds of system calls, only a few gates with specific numbers of parameters (one, two, four, and so forth) may be all it takes. All system calls that have one parameter go through the call gate with one parameter, and so on. Each call requires a

register/parameter to identify the service because many services converge toward the same call gates.
As I mentioned earlier, call gates can only transfer a fixed number of parameters. If some services have a variable number of parameters, an argument count and an argument pointer must be passed instead.
Call gates require far pointers, whereas traps or interrupts are simply triggered via one instruction (no pointer nor segment register at all), which is faster.
A word of caution: whenever privilege checks are involved by the CPU, execution cycles dramatically increase. Here are some examples on a 386 (for reference, the fastest instruction, excluding lock, requires two cycles):

- Operations on descriptors, such as lsl (load segment limit), lar (load access right byte), and the like usually take more than 10 cycles. Directly accessing the tables where they reside might be a faster way to get the information
- Loading a segment register in itself takes at least 18 cycles, compared to two with general registers. (This extra time has to do with the internal validation of the descriptor.) Remember that within a task, you should load a segment register only if the new value is different (the comparison instruction itself only takes two cycles)
- Loading the current LDT—the lldt instruction—takes 20 cycles
- Loading the task register (setting the current TSS)—the ltr instruction—takes at least 23 cycles
- A call/interrupt/trap gate to a higher-privilege descriptor takes a minimum of 90 cycles (and it increases with the number of parameters for a call gate)
- And the worst case: a task switch through a TSS takes more than 300 cycles! TSS task switches are only useful if all registers—especially the segment registers—must be reloaded with new values

IMPLEMENTATION EXAMPLES
These four features (LDT, TSS, privilege levels, and the variety of gates) can be used in a multiple of ways to satisfy specific needs. [2] Following are some examples of implementing protection in an OS, from the easiest to the hardest.
Case 1. The OS and a single task run at privilege 0 (see Figure 11).

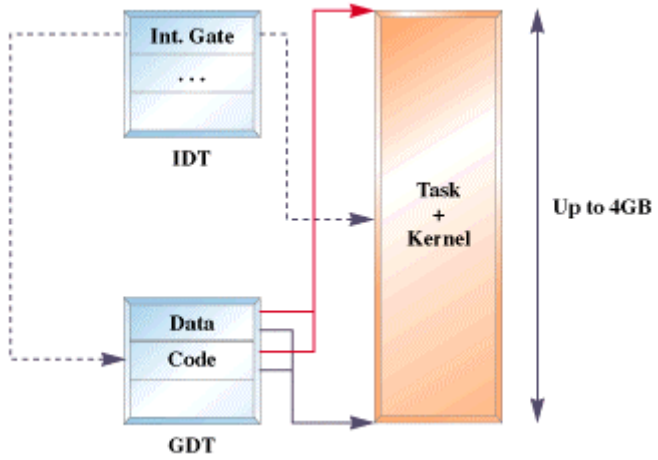


Figure 11

This is ideal for a simple, real-time dedicated 32-bit controller and is easy to achieve, such as with a fuel-air mixture analyzer that needs 32-bit registers to perform calculations with a certain precision.

- The OS and the task form one combined image
- Two entries are required in the GDT (in addition to the first entry): one code descriptor (zero to 4GB, DPL 0), one data descriptor (zero to 4GB, DPL 0). Segment registers always refer to these code and data descriptors
- Hardware interrupts are implemented via interrupt gates, all DPL 0, which call interrupt handlers

- No call gate nor task gate is used; the system services can be called directly
- A TSS isn't required because there is only one task and no privilege transition

Case 2. The OS and many tasks run at level 0. This model would be best for a multitask, real-time kernel, such as mC/OS. This case is ideal for a breaking system that needs multiple tasks to simultaneously control hydraulic systems, breaking force, collect statistical data, and the like. Such a system has a similar architecture to the previous case.

- Because all tasks have the highest privilege, they can share one single segment, so there is no need to use LDTs. Thus, only one code and one data descriptor (CPL 0, zero to 4GB) are required in the GDT
- Interrupts are implemented via interrupt gates (DPL 0)
- TSSes aren't required because no privilege transition exists and segment registers don't change. Task switches are done by saving application registers on the stack, switching stack, and restoring registers from the new stack

Case 3. The OS runs at level 0 and many tasks run at level 3 (see Figure 12). This

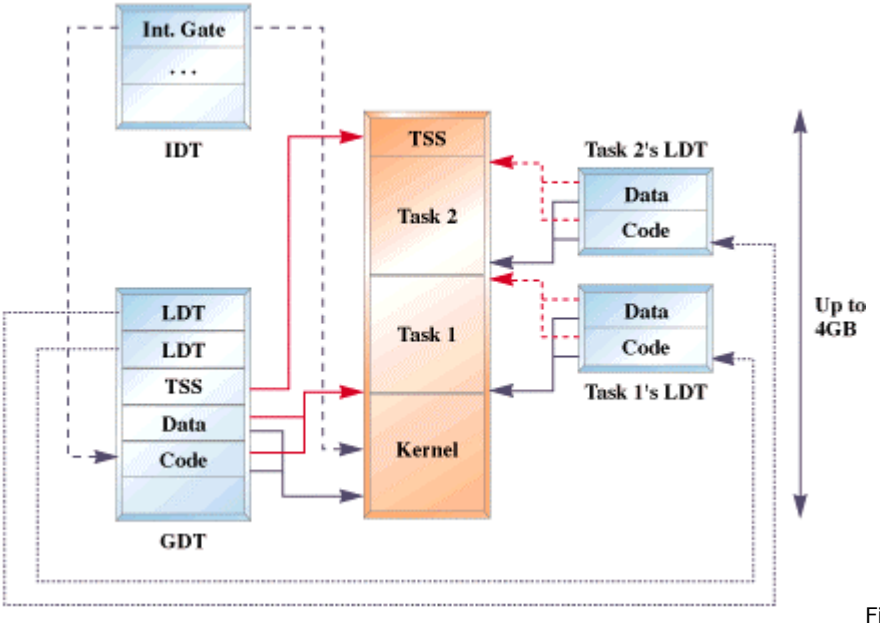


Figure 12

model would be best for a simple system running untrusted tasks, such as an embedded Java Virtual Machine that supports unknown Java applets.

- All descriptors in the GDT have a DPL of 0 to prevent tasks from using them directly
- The GDT has one code and data descriptor (CPL 0, zero to 4GB) for kernel use only
- Each task runs in its own address space, and needs its private LDT with two entries: one for the code and one for the data. For a flat memory model, the code and data segment of a given task may overlap; for better protection, they may be distinct. In the latter case, each task must be built apart (not linked with the kernel) using a small memory model, and loaded in order to be run (a loader is required). Tasks run at privilege 3 (which prevents accessing kernel code and data directly). LDTs prevents tasks from seeing each other. LDT descriptors reside in the GDT
- The IDT contains descriptors referring to interrupt handlers (to maintain interrupts disabled when the handler is called) in the kernel, at DPL 0, to ensure that kernel code always runs at 0
- TSSes can't be avoided because protection transition requires a stack switch, which is done from the current TSS. TSS descriptors reside in the GDT If a flat memory model is retained, system services may be callable through interrupts (to avoid the far system calls required with call gates)
- Message-based systems usually have few system calls (send, receive), which may equally be called via interrupts or call gates, passing parameters via registers

Case 4. The OS runs at level 0, system libraries at level 1, device drivers at level 2, and many tasks at level 3, each of them with multiple segments. This case is a complicated variant of Case 3 (more descriptors, more privilege levels) and it requires more effort to implement it. This case would be best for a high-end system rather than an embedded one. Compared to Case 3:

- System libraries are accessed through call gates that reside in the GDT, making them available to all tasks. Application libraries may hide these call gates from the tasks
- Devices drivers have their code, and data segments too, at DPL 2 in the GDT (if they are public) or in their own LDT (if they are accessible by the kernel only)
- Each task has its own TSS, and for this case, switching via task state segments might be justifiable, since all registers must be changed

A system such as the one in Case 4 is hardly justifiable, since it can be simplified and rendered more powerful by using paging—which is the subject of the next article.

Jean Gareau received an M.S. in electrical engineering from the Polytechnic School of the University of Montreal. Since 1989, he has been involved in the development of operating systems, system tools, and large commercial applications. He can be reached at jeangareau@yahoo. com.

REFERENCES

1. mC/OS is a portable, ROMable, preemptive, real-time, multitasking kernel for microprocessors, and it's free!
2. So many possible combinationsactually exist that this extra flexibility is a difficulty in itself.

BIBLIOGRAPHY

Intel Corp., *80386 Programmer's Reference Manual* . Order Number 230985-001, 1987.
Labrosse Jean J. m *C/OS The Real-Time Kernel* , Fourth Printing. Lawrence, KS: R&D Publications, 1992.

Advanced Embedded x86 Programming: Paging

by JEAN GAREAU

This article is the third and final in a series describing protected-mode features of the Intel x86 family, from the 80386 through the Pentium. RTOSes, embedded applications, and development tools can be updated to take advantage of the x86's 32-bit programming capabilities and larger, simpler memory models.

The first two articles in this series introduced 32-bit programming on the 80386 and its successors, switching into protected mode, and implementing protection and segmentation in protected mode. Let's quickly review these features.

Each segment register is an index to a table of descriptors, each of which describes a segment of memory by a base address, a limit, a type, and some protection fields. A linear address is produced from a segment/offset register combination by adding the offset to the base address found in the descriptor (which is pointed to by a segment register). Among the descriptor's protection fields is the descriptor privilege level (DPL), which sets the current task's current privilege level (CPL). Only a CPL of 0 (the highest level) gives full privileges to execute protected instructions (set or clear the interrupts, and so forth). Applications usually have a CPL of 3 (the lowest level), which gives them no privilege.

Segmentation offers flexibility and protection but it presents various constraints: it can seriously increase the complexity when many memory segments are used; switching segment registers increases execution time; each segment has a limited, static address space; and development tools must also support segmented memory models. An alternative is to preserve the segmentation's main advantages (virtual memory and protection), but replace all segments by a per-task flat and flexible address space. Paging, the feature that allows just that, is the subject of this article.

Paging is ideal for a multitasking embedded application whose tasks may require a very large address space or share a lot of data. It better fits systems with a few megabytes of ROM and RAM. Some recent high-end embedded devices fall into that category; for instance, a Web appliance (such as a TV Web device) may run many instances of the same browser. Paging allows the code of all these browsers to be shared instead of being uselessly duplicated, freeing precious memory. These browsers may temporarily need large address space to load pages with a lot of text, images, and sound. Paging allows you to store and access these megabytes of data easily through a flat address space.

You can download examples that implement various kernel designs from www.embedded.com/code.htm. These examples demonstrate the concepts explained in this series, including a port of mC/OS to protected mode. The source code is provided as well as ready-to-run executables and additional tools.

A QUICK TOUR OF PAGING

A system that implements paging breaks a task into a multitude of small pages, as illustrated in Figure 13. The size of a page typically ranges from 512 bytes to 8K, and is CPU-dependent. Each task is under the illusion that it has a huge flat address space, composed of hundreds of thousands of pages; however, only the pages in use have to be in physical memory. The other pages reside on disks or can even remain compressed elsewhere-in flash memory, for instance. As an application requires more stack or data memory, physical pages are dynamically and transparently allocated in RAM by the operating system (OS).

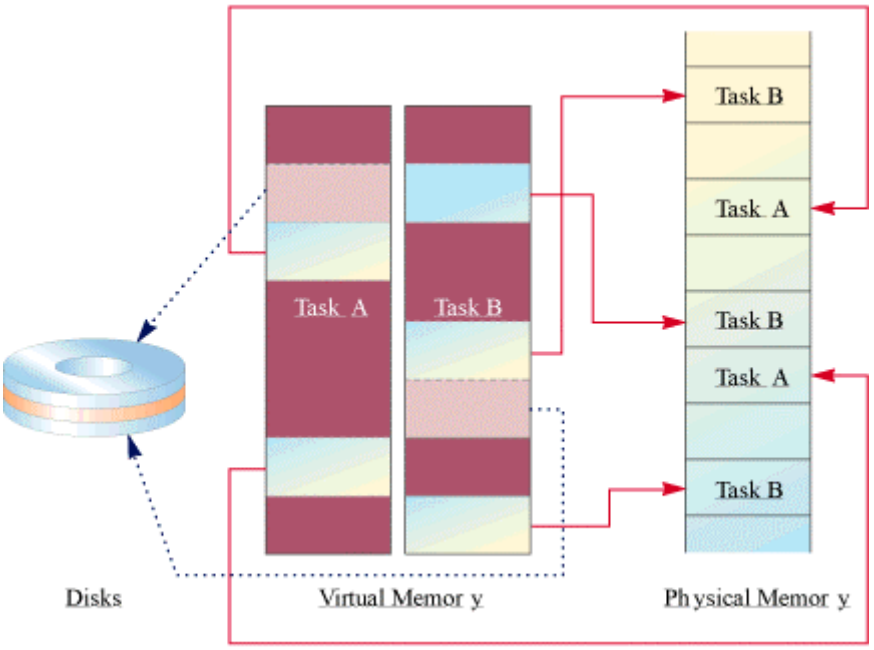


Figure 13

Paging offers many advantages, the first of which is a simple memory model. Each task has a large and uniform address space (no more segmented memory model, such as small, large, and so on). Segmentation can be ignored, simplifying application development. Development tools are also simplified because a flat memory model is much easier to handle than a segmented one.

Paging also offers a smaller task footprint. The physical space that must be committed for a task is directly proportional to the number of pages it needs, unlike segments that require a fixed amount of memory. Pages can reside anywhere in memory and do not need to be contiguous, optimizing the use of the physical memory and rendering the tasks' physical location irrelevant. Only the OS has to keep an eye on their physical location.

Another advantage offered by paging is efficient memory allocation. Pages can be allocated and deallocated on the fly, quickly expanding or shrinking task stacks or heaps. Pages can also be shared among tasks, and then can be replaced on the fly. For instance, a task in ROM can be partially or totally updated by adding new pages in flash memory and reorganizing the task's address space to use the new pages.

To summarize, paging is ideal to support large applications with multiple tasks. On the other hand, it imposes significant memory overhead on small systems and isn't trivial to implement.

A CLOSER LOOK

Under paging, each task is broken up into a series of fixed-size pages (See Figure 13). These pages can reside anywhere in memory and do not have to be contiguous.1 Only the pages that are accessed have to be in memory. For instance, if an application executes just a few functions, only the pages containing these functions need to be loaded in memory; the other pages may stay on a disk.

In a simple paging system, an address is broken in two: the left portion is a number that indicates a page in memory, whereas the right portion is considered an offset within the page (see Figure 14a).

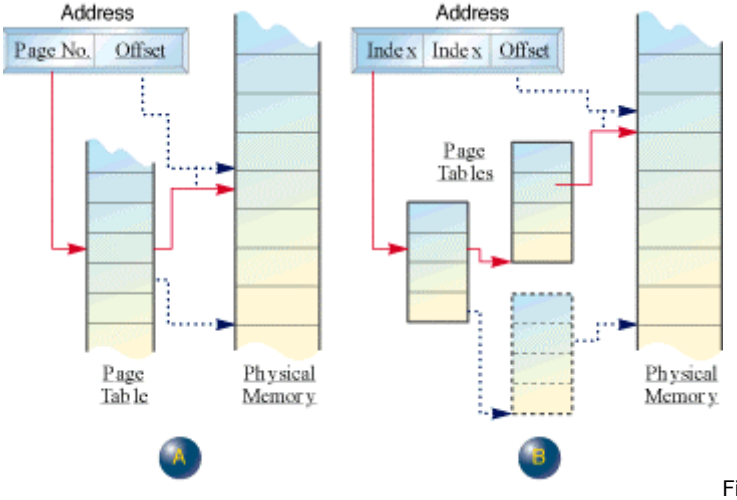


Figure 14

The OS uses internal tables to map the page number to its physical location. Under that scheme, pages may reside anywhere in memory. A simple paging system such as the one I just described requires a huge page table for a 32-bit system, since the page table has to cover the entire address space of 4GB for each task. Such a huge page table would span many contiguous pages itself. To prevent this situation, this huge page table is fragmented in smaller page tables, resulting in a three-level page table hierarchy, as seen in Figure 14b. Large systems typically use a three- or four-level hierarchy, and accordingly split any address into three or four indexes. These systems are more complex, but they allow anything to be split into fixed-size pages.

Pages and page tables are constructed by the OS as applications are loaded in memory. The code and data are loaded into pages called page frames, whose addresses are stored into page tables by the OS. Because a task sees the page frames through its page tables, it cannot see (or alter) another task's page frames. As a task requires memory, more pages are transparently allocated by the system; at the end, a task only uses the pages it needs, and no more. Tasks are never involved in page management; it's the operating system's business.

Paging can be implemented only if the underlying CPU supports it. Alternatively, a CPU may rely on an external memory management unit (MMU) instead. The hardware is involved because each address must be translated into a page whenever code or data is accessed. Needless to say, the translation logic must be optimized, given all the translations that occur when tasks run. Caching is extremely important at that level for performance considerations.

PAGING ON THE x86

The x86 uses a three-level hierarchy, as shown in Figure 14b. As I've explained, any logical address (segment/offset pair used in the task) is translated into a 32-bit linear address through segmentation. When paging is enabled, the linear address is broken up into three components: a 10-bit directory index, a 10-bit page index and a 12-bit offset (see Figure 15).

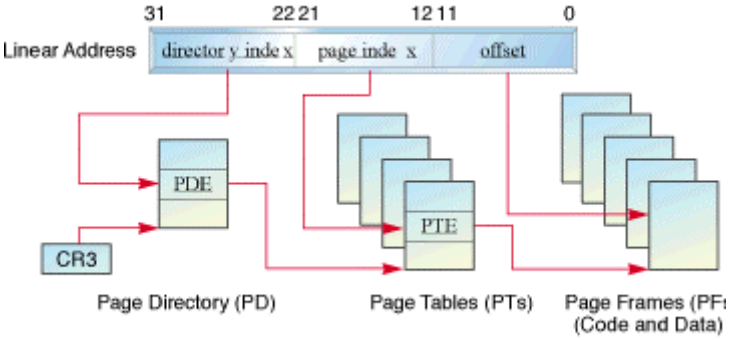


Figure 15

The OS must create and initialize, for each task, a page directory (PD) and at least one page table (PT). Only one page directory may be active at a time, indicated by the CR3 register. The 4K page directory contains 1,024 (210) four-byte entries, called page directory entries (PDEs). The linear address' 10-bit directory index is an index to this table, to a specific PDE. This PDE in turn contains the address of a page table, which is very similar to a page directory: it contains 1,024 four-byte entries, called page table entries (PTEs). The linear address' 10-bit page index is an index into this page table, to a specific PTE. This PTE points to a page frame (PF), also 4K, which contains task code or data. The linear address' 12-bit offset is an offset into this page. At the end, a 32-bit address points to a byte in a specific page.

Note that a page size on an x86 is always 4K and a page entry is always 32 bits wide (20 for the page address and 12 control bits). A page address is obtained by taking the 20 address bits in the page entry and adding 12 zero bits. Consequently, the pages are always aligned on a 4K (212) boundary.

Let's see how a hypothetical OS could create and manage those pages on an x86. We'll start with an over-simplified example: a paged system with one task. This example isn't realistic because paging would add more memory overhead than if we didn't use it, but explaining paging concepts will be easier.

Let's assume the OS is running and that a 32K task is ready to be loaded and executed. The OS starts by creating the page directory. Even if in this case one entry will be used, an entire page (4K) must be allocated. The address of that page is stored in the CR3 register. Then the OS identifies the first page of code that will be executed, based on the task's entry point (usually written somewhere in the executable image-the .EXE). The OS also creates a page table that points to that page of code, and stores the address of that page table in the proper page directory entry (see Figure 16).

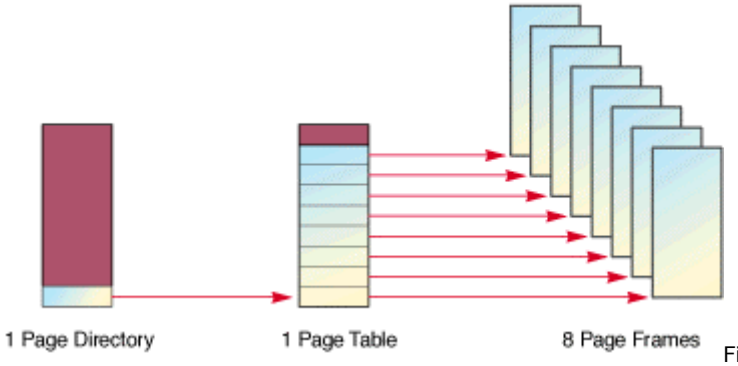


Figure 16

Then control is given to the task, which sends the address of the first instruction to execute on the address bus. The CPU translates this address by splitting it into the page directory index, page table index, and page frame offset. Because the OS carefully prepared the page directory and the proper page table, the instruction is located, fetched, and executed. The execution continues with the next instruction and so on. Not that magic, huh?

You may wonder what happens when execution goes beyond that unique page of code in memory. After all, the task has a size of 32K, so it has more code than a single page. Each entry in the page directory and page tables contains a present bit, managed by the OS, initially set to zero. The bit is set to one if the entry contains a valid address of a page in memory; it contains zero if the entry hasn't been initialized or is no longer valid. Let's say the task jumps 8K ahead (for example, two pages ahead). The target instruction's address is decoded by the CPU, but this time either the page directory entry or the page table entry will have its present bit set to zero, since the target code is not already loaded. That condition automatically raises a page fault exception. The OS reacts by analyzing the address, only to realize that it's valid but the page isn't in memory. Fair enough, the proper code page is loaded in memory, the page table and page directory are updated, and the instruction is restarted; this time, it succeeds. The same scenario is repeated for other pages of code or data access. This method is called demand paging, because pages are loaded as the task requires it-on demand.

Now let's execute another task. Again the OS prepares a page directory and a page table, different from those allocated for the first task. A page of code is loaded in memory and the second task starts executing its own code. The OS here makes sure that the page directory and page tables of that second task only point to code and data pages of that task. By doing so, the second task only sees its own code and data, and never sees (or alters) the first task's code and data.

As tasks' pages are allocated, the OS eventually runs out of physical memory. What happens when a page fault is triggered because a task wants to execute unloaded code? The OS needs to discard unused pages. As a task executes, some of the executed code will in fact never be executed again. The OS can't predict if some pages of code will ever be used again or not, but it can guess which pages are least likely to be required again. These pages are deallocated-the present bit of the page table entries pointing to them is reset to zero. The locations of these pages become available in order to load other pages that are in demand. The OS tries to reduce page faults, which can incur a significant source of overhead. Imagine if an interrupt is triggered and the handler isn't already paged in memory-the delay for loading the page could simply be unacceptable. A solution in this case is to make sure that interrupt handlers are always in memory and never deallocated. But invariably, numerous page faults are to be expected, as execution is unpredictable.

By carefully allocating and deallocating pages, the OS can keep in memory the pages required by the tasks. By keeping a few pages of each task, the OS is able to run many tasks, even if the total size of these tasks far exceeds the available physical memory. Paging gives each task 4GB of private virtual memory, although only a fraction of that address space is resident in memory at any moment. But from each task's standpoint, there is 4GB of memory to play with (although I've yet to see any embedded task taking advantage of all that virtual space).

Quite interestingly, for a given task, only one register is involved in the address translation: CR3, which points to the page directory. When a task switch occurs, only CR3 needs to be reloaded with the next task's page directory address, and here it goes in its own, private address space. Segmentation isn't disabled under paging, but by always using descriptors with a base address of zero and a limit of 4GB, one can safely forget about it, as segments never have to be changed.

Another plus for paging is the possibility of easily sharing pages. Let's say the same task is run twice (two instances). A good example is a task that monitors an analog device; in a system with two analog devices, two tasks may be required (one per device). Since the code is the same, the two tasks may share the code. This sharing is simply achieved by loading the code once in memory, and having both tasks' page tables pointing to the

same pages (see Figure 17). The larger the shared code, the bigger the gain. Shared libraries are also good candidates for code sharing. Writable data is not sharable (although it can be shared until it is modified).
All in all, despite the features directly implemented in the CPU, the challenge is in designing how the system will manage pages, which pages should be deallocated, how many pages a single task could be allowed in memory at once, which pages could be anticipated and pre-allocated to speed task execution, and so forth. The CPU gives you the tools, but you really have to prepare a good system design beforehand.

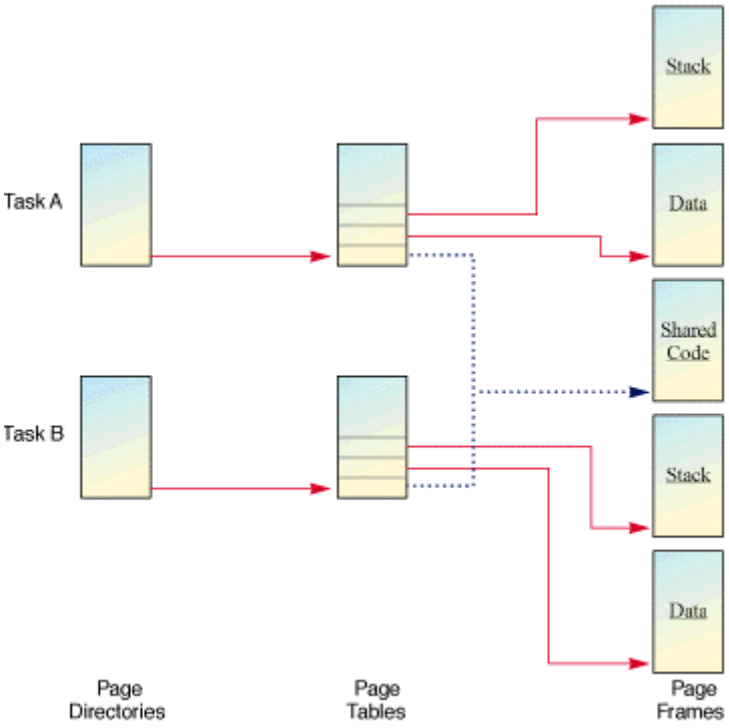


Figure 17

TASK ADDRESS SPACE LAYOUT
Although each task has a virtual address space of 4GB, partitioning this memory for various uses is important. The OS must reserve some of that space for itself-we'll see why in a moment-and 2GB (the upper portion of each address space) is commonly reserved for the system, leaving 2GB for the task. The task's code usually starts at the bottom of the address space, followed by the data. The stack usually starts at the end of the 2GB (below the space reserved by the OS) and grows downward. The heap (for dynamically allocated data) sits between the data and the stack. Other combinations are acceptable, depending on the system's needs. The important concept is that all that space is virtual; when the task starts to execute or access data, physical pages are allocated one by one, as required. Address space layout is a concern for OSes, compilers, and linkers, but not for application developers.
Because each application only sees its own 4GB, kernel services (invoked by the application or an interrupt) must be mapped within that range as well. In fact, the kernel must be mapped in all tasks to be equally accessible. System calls can be implemented as call or interrupt gates, as long as they point to the proper handler in the address space of each task.
When a system call is invoked, the kernel begins executing within the context of the task being interrupted or making the call. If some arguments are passed in the call (even pointers), they can be used as is to access task's data.

MAPPING THE OPERATING SYSTEM

Upon initialization, the OS must build an initial page directory and page table to activate the paging. These could actually belong to a permanent monitor, debugger, or simply the idle loop. The operating system's code and data are the page frames (if the OS is entirely loaded). The OS can map itself from linear address 0, but also from, say, address F0000000h (see Figure 18).

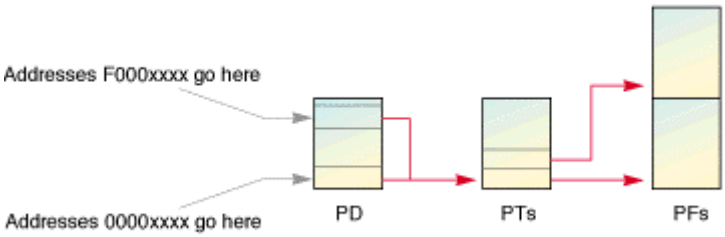


Figure 18

All call gates and interrupt handlers are set to addresses above F0000000h, which lead to their real locations in physical memory.

When the first task is built, the upper part of its page directory is mapped to all system page tables, mapping the OS into its address space, as shown in Figure 18. When that task executes, an interrupt or a call gate will jump somewhere into the OS.

There is no rule regarding whether the OS should be mapped in the bottom or the top of the task address space. However, many OSes map themselves at the high end of all address spaces, leaving the application at the bottom (smaller addresses are more human-readable). But you may well implement an alternate design, depending on your needs. Following are certain issues to consider.

Use flat segments (base address of zero, limit of 4GB). Unless you have extraordinary constraints, you can forget about segmentation by keeping it that way.

The OS must be able to access all physical memory while paging is enabled. Because the kernel executes in the context of the interrupted task (whichever it is), the entire physical memory must be mapped in all tasks. In the previous example, mapping the kernel from F0000000h gives access up to 256MB of RAM. In order to support, say 512MB of RAM, the OS would have to be mapped at E0000000h.

Shared libraries, if you intend to support them, can be mapped in the kernel. Since the kernel is mapped into all tasks, the shared libraries will be too. Dynamic linking is easier if each library resides at the same address in each task. In the previous example, the space between C0000000h and F0000000 (768MB) is a good placeholder.

Reserving address space for system usage reduces the address space of all tasks. Some systems keep 2GB of address space (the upper half) for themselves, no matter what, to give OS designers ample room to implement features in future releases without changing the architecture.

PROTECTION REVISITED

Protection also exists at the paging level, in addition to the protection already present in the segmentation (which is always enabled). Page directory and page table entries have two protection bits: read-only, and privilege required to access the page; either CPL 0 (supervisor mode) or above zero (user mode). The operating system's page entries are always marked supervisor mode, whereas task's page entries are marked user mode. If a task with a CPL of one, two, or three tries to access any pages marked supervisor, even to read only, an exception will be raised (and the OS may destroy that task).

The CPL of each task is still dictated by the code segment's DPL. A simple yet efficient design involves using a DPL of three with task's descriptors and zero with the OS. Thus, combining protection features from segmentation and paging provides an effective shield over the system resources.

ACTIVATING PAGING

Let's review a code example that demonstrates how to activate paging on an x86. Paging is activated once the CPU executes in protected mode with full privileges (CPL 0). If the protected mode is turned off, so will be the paging. The next example starts in real mode, with the interrupts disabled. It then enables protected mode and switches into 32-bit (as presented in the first article of this series). It then activates paging and maps the kernel at the end of its address space (F000xxxx).

The example starts its execution at a low physical address (0000xxxx) and will end up somewhere above F000xxxx. An address issue exists here, regarding a single application running at 0000xxxx and F000xxxx: if a directive such as ORG F000xxxx appears in the program, most linkers will try to fill the gap between the instruction before the directive and the instruction that follows (in this case, almost 4GB). Such a directive can obviously not be used. The only way to resolve the problem is to set the application base address to F0000000h using a linker option (most recent linkers have such an option), and to bring all "pre-paging" instructions into low addresses by subtracting F0000000h from them, or using relative addressing. This action affects only a few instructions.

One page directory and one page table are required before activating paging. Both are pre-allocated in the example (the page directory is at line 49 and the page table at line 51); they could have been allocated dynamically if dynamic location were available. The only requirement is that the pages must be aligned on a 4K boundary; their physical location isn't important.

The example is loaded at physical address 0, so the page table is initialized to cover the physical page frames from physical address 0. The entire page table is initialized, covering up to 4MB of physical memory (lines 108-116), although only a few entries will be required in the example. The page table address is stored in the first page directory entry (lines 101-103), making virtual address equal to physical address when paging is enabled.4

Addresses that start with F000 result in a page directory index of 960. In order to map the code at address F0000000h, the page table address is also stored in page directory entry 960 (lines 105-106). The page directory has two entries referring to the same page table, as shown in Figure 17. Finally, the CR3 register is set to the address of the page directory (lines 121-122). The kernel could be mapped at another location simply by properly initializing the page directory. For instance, if the kernel is to be mapped at E0000000h, PDE 896 instead of 960 must point to the first page table. The program would also have to be linked with a base address of E0000000h.

Paging is then enabled by setting bit 31 in CR0 (lines 126-128). From that point, all instructions are decoded using the paging translation. The translation then maps virtual addresses to physical addresses. The instruction queue must be flushed in order to prevent any problems with the pre-fetched, pre-paging instructions (line 129).

The next step is to switch into the high end of the address space. A jump is simulated by PUSHing the address (as is) of the next instruction and RETurning to it (lines 133 to 134). A relative jump cannot be used because the assembler doesn't know that half of this code is running at 0000xxxx and the other half at F000xxxx.

From that point, the rest of the OS is initialized. All trap, interrupt, and call gates must point to functions in the high address space (F000xxxx). Finally, if a task was created, its page directory's entry 960 would have to be mapped to the system page table. Thus, any reference by any gates to the addresses in that range would properly end up in the OS.

AN IMPLEMENTATION EXAMPLE

The real issue that arises when implementing paging has to do with the task address space layout, and finding the proper balance between system space and task space, where the various components (task, system services, shared libraries, and the like) will be mapped, what kind of protection is required, and so forth. If swapping is supported, you'll have to identify the task working set (how many pages at once in memory), the page replacement strategy (what page to remove if there is some memory contingency), and so on.

Here is an implementation of a multithreaded, multitasking OS (such as an embedded Java Virtual Machine running large applets, an embedded Web server, or a TV Web device), illustrated in Figure 19. Some tasks are considered untrusted and use their own flat address space. All threads of a task share the same address space.

Segmentation:

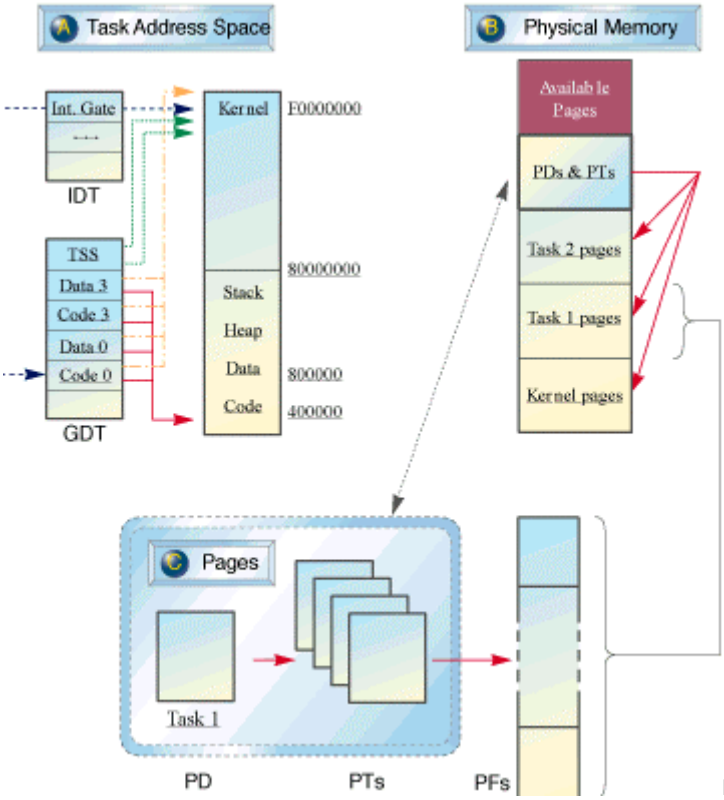


Figure 19

- The GDT contains one code and data descriptor for the OS (DPL 0, 0GB to 4GB) and one code and data descriptor for the tasks (DPL 3, 0GB to 4GB). Without privileges, tasks cannot execute privileged instructions
- System calls are provided by either call or trap gates, both of them using the kernel code selector and relevant service addresses. Interrupt descriptors also use the kernel code selector. The kernel selector allows them to run at CPL 0
- No LDT is required because isolation is obtained through paging
- One TSS is required because of the privilege transition. Task switches can be done by saving registers on the stack and switching the stack, instead of using the TSS, because segment registers never change. The TSS descriptor is in the GDT

Paging:

- Each task has its own page directory, which is shared among all its threads (hence all threads of a task share the very same address space)
- Code and data use different page tables, to potentially share code page tables with other instances; the stack pointer is set at 80000000h and grows downward
- The task's upper-half page directory entries are all marked "supervisor." The tasks cannot access any operating system's code or data. This 2GB area is reserved for the OS, the shared libraries, and hardware maps (the video buffer, for instance)

This series of articles has demonstrated the multiple features of the x86: native 32-bit programming, virtual memory with segmentation and paging, multitask support, and protection. These features exist to provide maximum flexibility to embedded developers, allowing them to design and implement a myriad of OS types, ranging from a simple segmented kernel with no overhead to an advanced page-demand, multitasking, and multithreaded system with full-task protection and shared-memory capabilities.
If you intend to develop your own OS, I would recommend as the most important step getting the proper documentation (such as the x86 programming manuals) for your processor. A few books about OS implementation on the x86 are also available. You'll be able to find enough examples and ideas to start building your customized embedded OS.

Listing: Paging

```
1. ; Paging.asm
2. ; Copyright (C) 1997, Jean L. Gareau
3. ;
4. ; This program demonstrates how to enable paging in protected mode.
5. ; A flat memory model and a simplified segment definition are used.
6. ;
7. ; This program has been assembled with MASM 6.11:
8. ;       C:\>ML ProtMode32.asm
9. ;
10.; When linked, it must have a base address of BASE (F0000000h in this
10b; example),
11.; which is where this code is to be mapped in its address space.
12.
13. BASE   EQU        0F0000000h            ; Base address (virtual)
14.
15.        .386P                ; Use 386+ privileged instructions
16.
17. ;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<-;
18. ; Macros (to use 32-bit instructions while in real mode) ;
19. ;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<-;
20.
21. LGDT32 MACRO      Addr                  ; 32-bit LGDT Macro in 16-bit
22.                   DB         66h        ; 32-bit operand override
23.                   DB         8Dh        ; lea (e)bx,Addr
24.                   DB         1Eh
25.                   DD         Addr
26.                   DB         0Fh        ; lgdt fword ptr [bx]
27.                   DB         01h
28.                   DB         17h
29. ENDM
30.
31. FJMP32 MACRO      Selector,Offset       ; 32-bit Far Jump Macro in 16-
bit
32.                   DB         66h        ; 32-bit operand override
33.                   DB         0EAh       ; far jump
34.                   DD         Offset     ; 32-bit offset
35.                   DW         Selector   ; 16-bit selector
36. ENDM
37.
38.                   PUBLIC     _EntryPoint ;The linker needs it.
39.

40. _TEXT  SEGMENT PARA USE32 PUBLIC ŒCODE'
41.                   ASSUME     CS:_TEXT
42.
43. ;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<-;
44. ; Page Directory and Page Table.        ;
45. ;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<-;
46.
47.                   ORG        3000h      ; => Depends on code location.<=
48.
49. PD:
50.        dd         1024 DUP(0)  ; Page Directory: all entries at 0.
51. PT:
52.        dd         1024 DUP (0) ; Page Table  : all entries at 0.
53.
```

```
;‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹-;
; Entry Point. The CPU is executing in 16-bit real mode. ;
;‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹-;

       ORG        5000h      ; => Depends on code location. <=

_EntryPoint:

       LGDT32     GdtDesc - BASE        ; Load GDT descriptor

       mov        eax,cr0               ; Get control register 0
       or         ax,1                  ; Set PE bit (bit #0) in (e)ax
       mov        cr0,eax               ; Activate protected mode!
       jmp        $+2                   ; Flush the instruction queue.

; The CPU is now executing in 16-bit protected mode. Make a far jump in
; order to
; load CS with a selector to a 32-bit executable code descriptor.

       FJMP32     08h,Start32 - BASE    ; Jump to Start32 (below)

; This point is never reached. Data follow.

GdtDesc:                                ; GDT descriptor
       dw         GDT_SIZE - 1          ; GDT limit
       dd         Gdt                   ; GDT base address (below)

;‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹-;
; The CPU is now executing in 32-bit protected mode ;
;‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹‹-;

Start32:

; Initialize all segment registers to 10h (entry #2 in the GDT)

       mov        ax,10h                ; entry #2 in GDT
       mov        ds,ax                 ; ds = 10h
       mov        es,ax                 ; es = 10h
       mov        fs,ax                 ; fs = 10h
       mov        gs,ax                 ; gs = 10h
       mov        ss,ax                 ; ss = 10h

; Set the top of stack to allow stack operations.

       mov        esp,8000h             ; arbitrary top of stack

; Store the PT address into PDE 0 and 960.

       mov        eax,offset Pd - BASE             ; eax  = &PD
       mov        ebx,offset Pt - BASE + 3         ; ebx  = &PT | 3
       mov        [eax],ebx             ; PD[0] = &PT

       mov        eax,offset Pd - BASE + 960 * 4   ; eax = &PDE[960]
       mov        [eax],ebx             ; PD[960] = &PT

 ; Initialize the PT to cover the first 4 MB of physical memory.

       mov        edi,offset Pt - BASE  ; edi = &PT
       mov        eax,3                 ; Address 0, bit p & r/w set
       mov        ecx,1024              ; 1024 entries
 InitPt:
```

```
114.       stosd                           ; Write one entry
115.       add         eax,1000h           ; Next page address
116.       loop        InitPt              ; Loop
117.
118. ; Turn on paging by:
119. ;      1) setting the PD address into CR3 and
120.
121.       mov         eax,offset Pd - BASE  ; eax = &PD
122.       mov         cr3,eax             ; cr3 = &PD
123.
124. ;     2) setting CR0's PG bit.
125.
126.       mov         eax,cr0
127.       or          eax,80000000h       ; Set PG bit
128.       mov         cr0,eax             ; Paging is on!
129.       jmp         $+2                 ; Flush the instruction queue.
130.
131. ; Let's now jump to F000xxxx.
132.
133.       push        offset PagingMode   ; Keep full address (F000xxxxh)
134.       ret                             ; Jump at Paging Mode (below)
135.
136. PagingMode:
137.
138. ;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<-;
139. ; -> Paging is now enabled, executing code at F000xxxxh <- ;
140. ;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<-;
141.
142. ; Other initialization instructions come here.
143. ;                ...
144.
145. ; This point is never reached. Data follow.
146.
147. ;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<-;
148. ; GDT                                        ;
149. ;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<-;
150.
151. ; Global Descriptor Table (GDT) (faster accessed if aligned on 4).
152.
153.       ALIGN       4
154.
155. Gdt:
156.
157. ; GDT[0]: Null entry, never used.
158.
159.       dd          0
160.       dd          0
161.
162. ; GDT[1]: Executable, read-only code, base address of 0, limit of
163. ; FFFFFh, granularity bit (G) set (making the limit 4GB)
164.
165.       dw          0FFFFh              ; Limit[15..0]
166.       dw          0000h               ; Base[15..0]
167.       db          00h                 ; Base[23..16]
168.       db          10011010b  ; P(1) DPL(00) S(1) 1 C(0) R(1) A(0)
169.       db          11001111b           ; G(1) D(1) 0 0 Limit[19..16]
170.       db          00h                 ; Base[31..24]
171.
172. ; GDT[2]: Writable data segment, covering the save address space than
173. ; GDT[1].
174.       dw          0FFFFh              ; Limit[15..0]
```

```
175.       dw          0000h                ; Base[15..0]
176.       db          00h                  ; Base[23..16]
177.       db          10010010b  ; P(1) DPL(00) S(1) 0 E(0) W(1) A(0)
178.       db          11001111b            ; G(1) B(1) 0 0 Limit[19..16]
179.       db          00h                  ; Base[31..24]
180.
181. GDT_SIZE          EQU        $ - offset Gdt       ; Size, in bytes
182.
183. _TEXT             ENDS
184.                   END
```

Jean Gareau received an M.S. in electrical engineering from the Polytechnic School of the University of Montreal. Since 1989, he has been involved in the development of operating systems, system tools, and large commercial applications. He can be reached at jeangareau@yahoo.com.

REFERENCES

1. In some cases, pages must be contiguous. DMA devices, for instance, require contiguous memory.
2. Page directory can be shared if tasks can share the same address space. This solution is ideal for threads. Whereas distinct tasks have their own address space, multiple threads of a given task all use the same page directory and consequently, the same address space.
3. Microsoft's MS-LINK 5.0 generates code that always starts at 1000h above the base address.
4. This is called identity-mapping, where virtual and physical addresses are the same.

BIBLIOGRAPHY

80386 Programmer's Reference Manual. Intel Corp., 1987. Order Number 230985-001.

Labrosse, Jean J., mC/OS The Real-Time Kernel, Fourth Printing. Lawrence, KS: R&D Publications, 1992.

McKusick, et al. The Design and Implementation of the 4.4BSD Unix Operating System. Reading, MA: Addison Wesley, 1996.

Reference: Microsoft MASM 6.11. Redmond, WA: Microsoft Corp., 1992, Document No. DB35749-1292.

Silberschatz, A., et al. Operating System Concepts, Fifth Edition. Reading, MA: Addison Wesley, 1997.

Tanenbaum, A. Modern Operating Systems. Englewood Cliffs, NJ: Prentice-Hall, 1992.

Tanenbaum, A. Operating Systems: Design and Implementation. Englewood Cliffs, NJ: Prentice-Hall, 1997.

Turley, Jim. Advanced 80386 Programming Techniques. New York: McGraw-Hill, 1988.