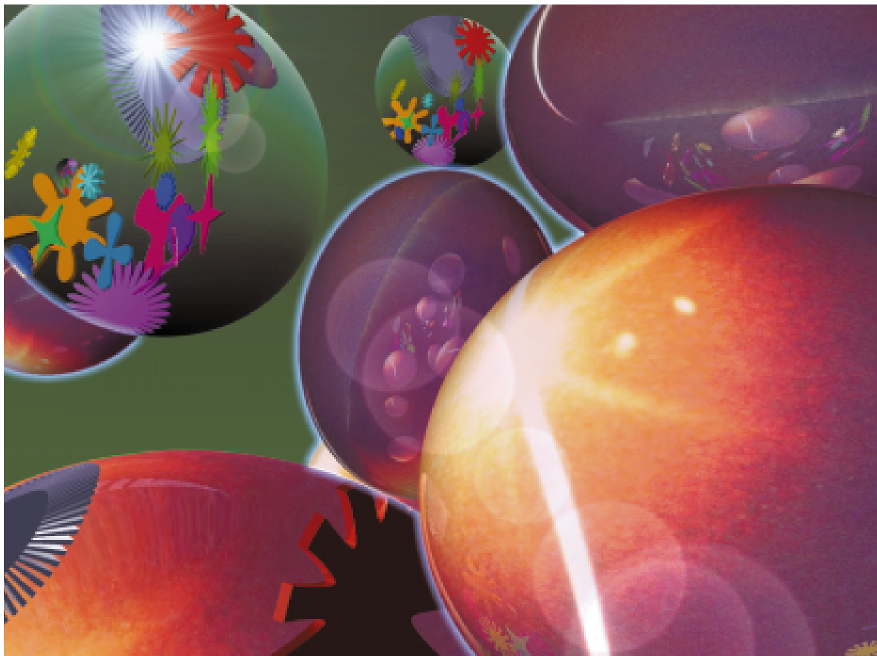
by JEAN GAREAU

Advanced Embedded x86 Programming: Paging



Rupert Adley

Look to paging for applications whose tasks require a huge address space or share a lot of data.

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

Paging allows the code of multiple browsers to be shared instead of being uselessly duplicated, freeing precious memory.

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*. These examples demonstrate the concepts explained in this series, including a port of μC/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 1. 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).

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.

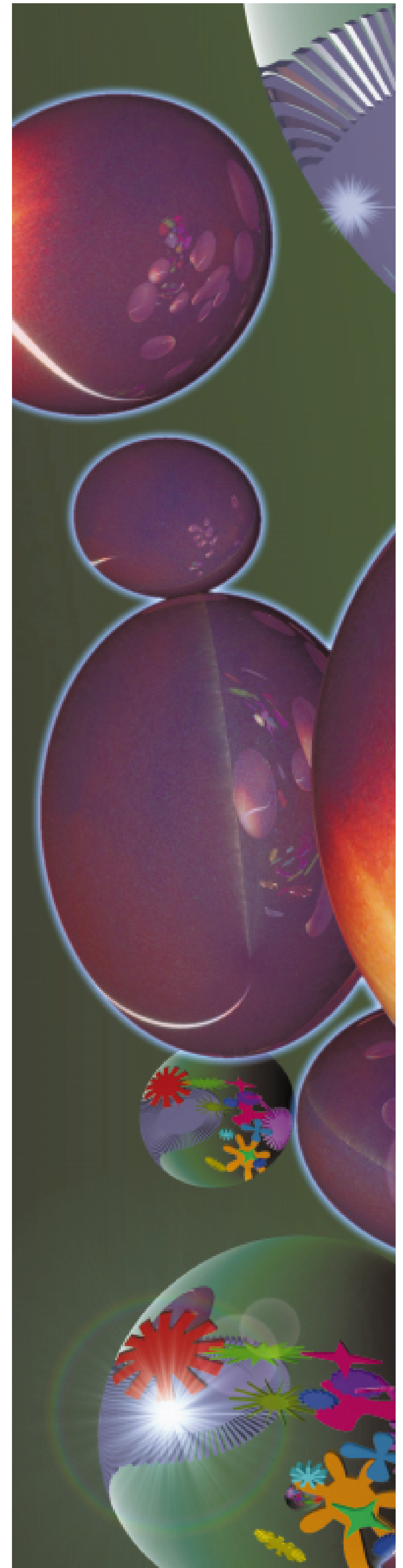
Paging also offers a *smaller task footprint.* The physical space that must be committed for a task is directly proportional to the number of pages it needs, unlike segments that require a fixed amount of memory. Pages can reside anywhere in memory and do not need to be contiguous, optimizing the use 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 1). 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 2a). 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 2b. 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.

FIGURE 1

Paging breaks tasks into a series of noncontiguous pages, anywhere in memory. Only the pages that are required have to be in memory; the others rest on a disk.

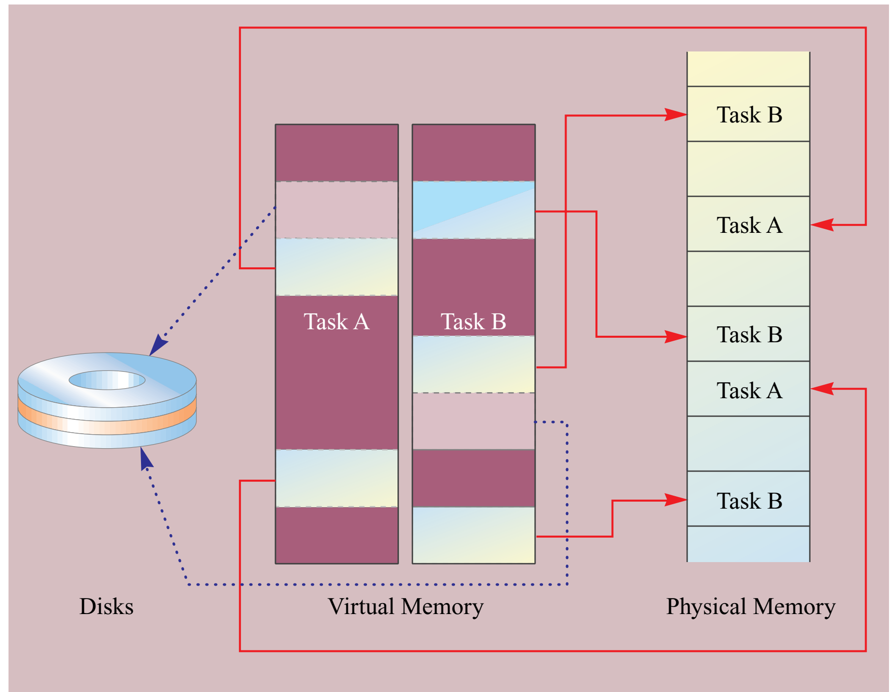
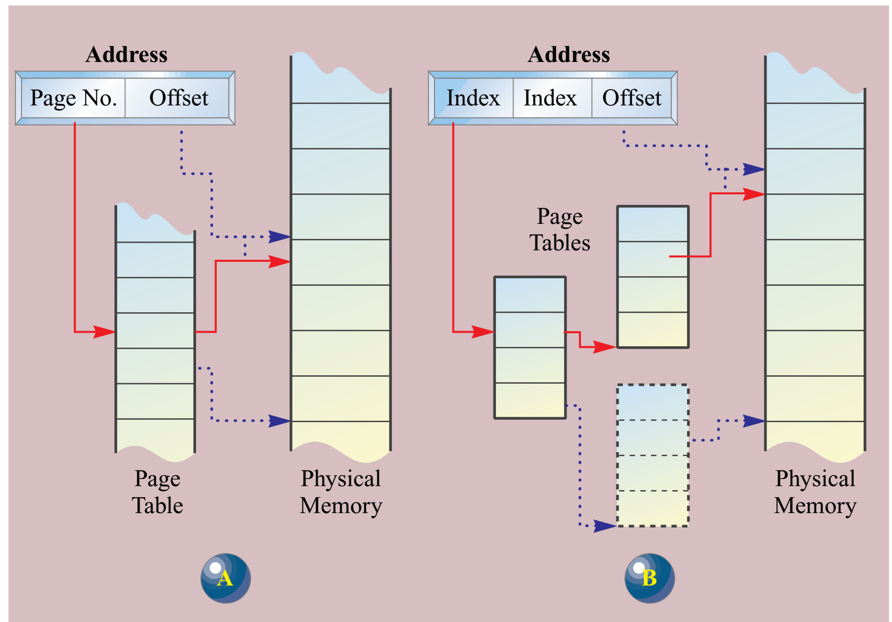


FIGURE 2

Paging can be implemented via a two-level (a) or a three-level (b) page table hierarchy. Some systems use even more levels.



PAGING ON THE X86

The x86 uses a three-level hierarchy, as shown in Figure 2b. 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 3).

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 (2^{10}) 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 (2^{12}) 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 4).

FIGURE 3
Paging address translation. A linear address is broken up into three components that identify a unique page frame, which contains some of the task's code and data.

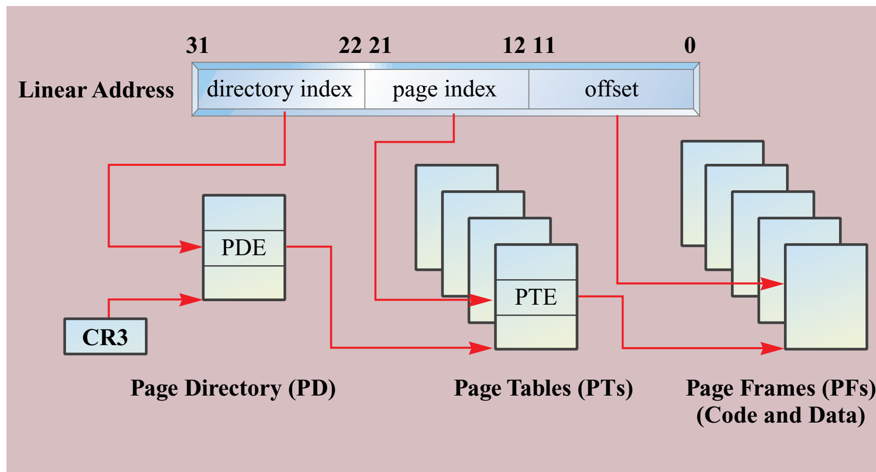
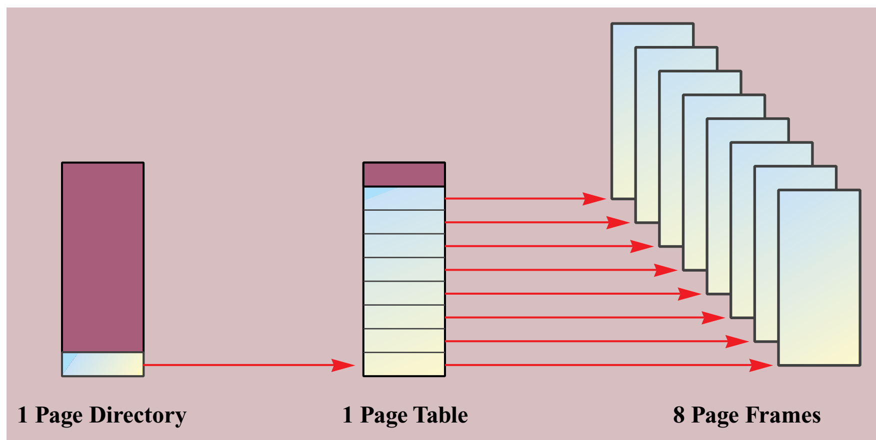


FIGURE 4
Creating system page tables for a 32K task. Grayed regions contain unused entries.



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.

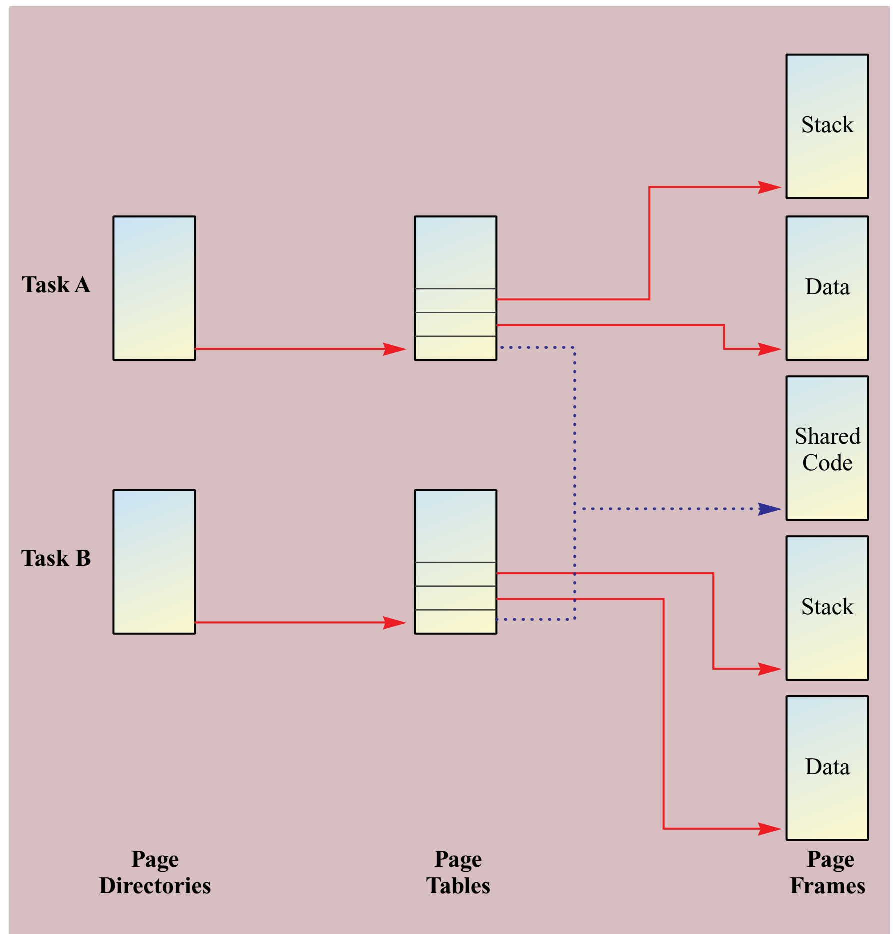
By carefully allocating and deallocating pages, the OS can keep in memory the pages required by the tasks. By keeping a few pages of each task, the OS is able to run many tasks, even if the total size of these tasks far exceeds the available physical memory. Paging gives each task 4GB of private virtual 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

FIGURE 5

Two tasks sharing a single code page. Each task has private page directory, page table, stack, and data pages.



is simply achieved by loading the code once in memory, and having both tasks' page tables pointing to the same pages (see Figure 5). 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 6

The operating system maps itself at zero and F0000000h. Two entries in the PD point to the same PT and consequently, to the same physical memory.

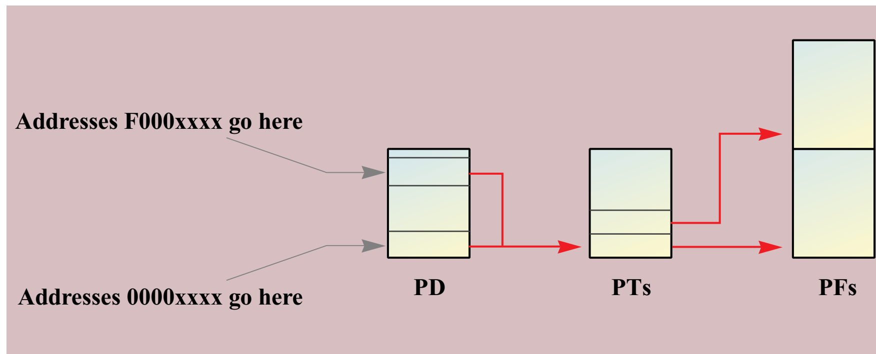
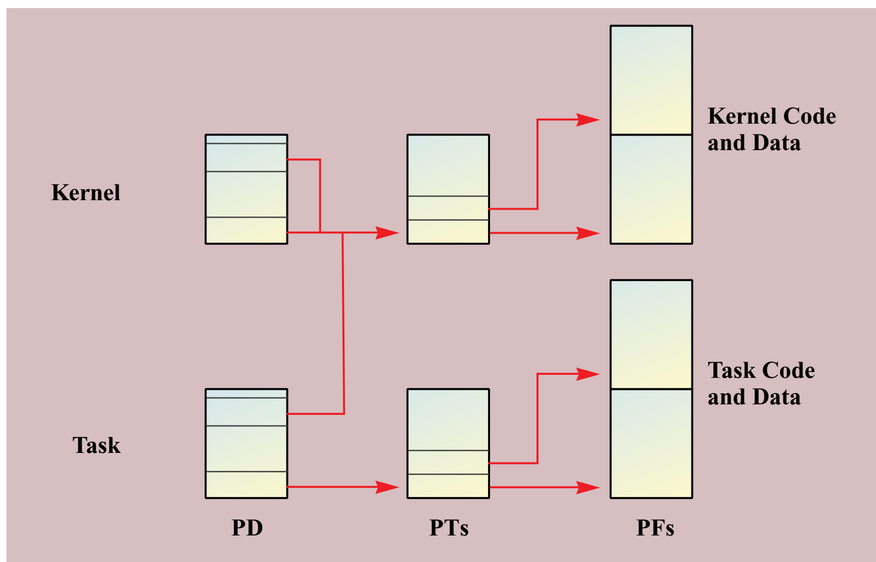


FIGURE 7

By having a task's PDEs pointing to a kernel's PTs, the kernel gets mapped into the task's address space. If an interrupt is raised while the task is running, it can directly jump into the operating system code.



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 6). 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 7. 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 6. 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 prefetched, pre-paging instructions (line 129).

The next step is to switch into the high end of the address space. A jump is simulated by `PUSH`ing the address (as is) of the next instruction and `RET`urning 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.

LISTING 1

Activating paging can be done with a few instructions.

```
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 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 'CODE'
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.
54. ;------------------------------------------------------------------------;
55. ; Entry Point. The CPU is executing in 16-bit real mode.                ;
```

Embedded x86 Programming: Paging

LISTING 1 (cont.)
Activating paging can be done with a few instructions.

```
56. ;---------------------------------------------------------------------------;
57.
58.                 ORG       5000h               ; => Depends on code location. <=
59.
60. _EntryPoint:
61.
62.                 LGDT32    GdtDesc - BASE      ; Load GDT descriptor
63.
64.                 mov       eax,cr0             ; Get control register 0
65.                 or        ax,1                ; Set PE bit (bit #0) in (e)ax
66.                 mov       cr0,eax             ; Activate protected mode!
67.                 jmp       $+2                 ; Flush the instruction queue.
68.
69. ; The CPU is now executing in 16-bit protected mode. Make a far jump in order to
70. ; load CS with a selector to a 32-bit executable code descriptor.
71.
72.                 FJMP32    08h,Start32 - BASE  ; Jump to Start32 (below)
73.
74. ; This point is never reached. Data follow.
75.
76. GdtDesc:                                      ; GDT descriptor
77.                 dw        GDT_SIZE - 1        ; GDT limit
78.                 dd        Gdt                 ; GDT base address (below)
79.
80. ;---------------------------------------------------------------------------;
81. ; The CPU is now executing in 32-bit protected mode.                        ;
82. ;---------------------------------------------------------------------------;
83.
84. Start32:
85.
86. ; Initialize all segment registers to 10h (entry #2 in the GDT)
87.
88.                 mov       ax,10h              ; entry #2 in GDT
89.                 mov       ds,ax               ; ds = 10h
90.                 mov       es,ax               ; es = 10h
91.                 mov       fs,ax               ; fs = 10h
92.                 mov       gs,ax               ; gs = 10h
93.                 mov       ss,ax               ; ss = 10h
94.
95. ; Set the top of stack to allow stack operations.
96.
97.                 mov       esp,8000h           ; arbitrary top of stack
98.
99. ; Store the PT address into PDE 0 and 960.
100.
101.                mov       eax,offset Pd - BASE; eax  = &PD
102.                mov       ebx,offset Pt - BASE + 3              ; ebx  = &PT | 3
103.                mov       [eax],ebx           ; PD[0] = &PT
104.
105.                mov       eax,offset Pd - BASE + 960 * 4; eax = &PDE[960]
106.                mov       [eax],ebx           ; PD[960] = &PT
107.
108. ; Initialize the PT to cover the first 4 MB of physical memory.
109.
110.                mov       edi,offset Pt - BASE; edi = &PT
```

LISTING 1 (cont.)

Activating paging can be done with a few instructions.

```
111.            mov      eax,3              ; Address 0, bit p & r/w set
112.            mov      ecx,1024           ; 1024 entries
113. 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 FFFFFh,
163. ;   granularity bit (G) set (making the limit 4GB)
164.
```

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 8. Some tasks are considered untrusted and use their own flat address space. All threads of a task share the same address space.

Segmentation:

- 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 demon-

LISTING 1 (cont.)

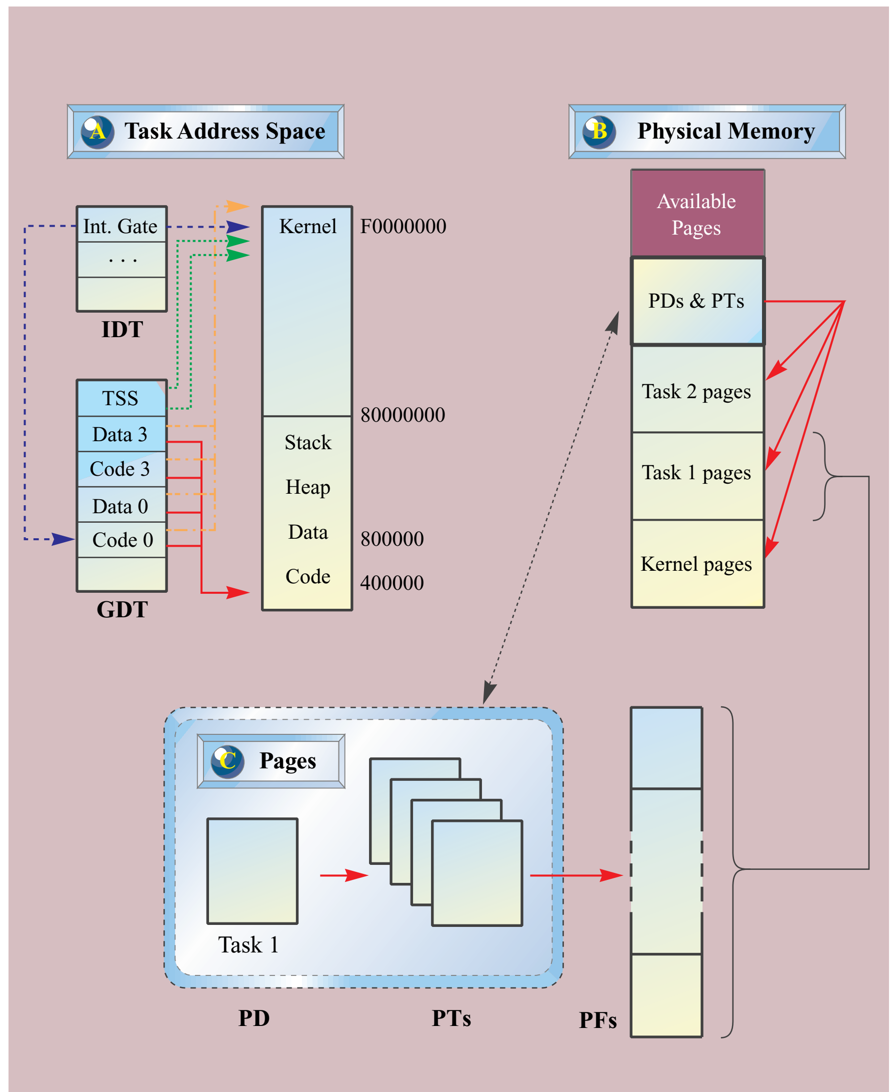
Activating paging can be done with a few instructions.

```
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 GDT[1].
173.
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
```

strated 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. **ESP**

FIGURE 8
A paged, multitasking, multithreaded OS. The GDT entries are always used to cover a 4GB address space (a). The physical memory contains the pages for the kernel's code and data, the tasks' code and data, and all PDs and PTs (b). Each task has its own page directory (PD) and page tables (PTs), which translate all linear address into page frames (PFs), all in physical memory (c).



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., *µC/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.