FEATURE

MMUs: Maximizing Mission-critical Utility

High-availability, mission-critical applications in data and telecommunications need to employ the most robust and reliable software possible. Maintaining software quality across applications with millions of lines produced by hundreds of programmers is the main challenge faced by software engineering.

By William Weinberg

Employing hardward-based memory management is one way of enhancing the reliability of applications in the field. Although this technology is often associated with large, multiuser systems, even microcomputers in embedded systems can benefit from it, since as much as 50% of the silicon in popular embedded CPUs like the PowerPC and Intel Pentium families is dedicated to a memory management unit (MMU). Too often, this valuable resource is misunderstood by developers and goes unused in applications.

This article explains memory management techniques, including protection, virtual addressing, and paging/swapping, and examines how each applies to real applications, with special emphasis on reliability and performance benefits.

Protection

Despite popular images of lone programmers hacking away into the night, most software projects of any magnitude come from teams of software engineers working together. Larger applications, especially those found in telecommunications and data communications, often employ hundreds of programmers generating millions of lines of code. The methodologies of modern software engineering concentrate on the disciplines needed to manage such large projects, but reality dictates that in such large groups of engineers, competence will vary greatly; moreover, even the best programmer cannot always produce bug-free code.

Nonprogrammers envision software bugs involving program logic — miscalculations, omitted program steps, and erased key data — but the bugs that make software engineers

Segment-based address translation is quite straightforward: The processor emits an address that must pass through the MMU to arrive on the system bus.

lose sleep are those involving program *structure*.

Modern software divides computer memory up into regions, principally for program code and several types of data. Without an MMU, these divisions are "soft," enforced only by the way that development tools lay out memory and by how programmers follow the layout discipline. Large multiprocess or multithreaded programs further divide these gross regions, giving each process, task, or thread of execution its own chunks of memory for code and data. Among the hardest bugs to find are those that violate this neat structure — when programs unintentionally modify their own code regions, accidentally corrupt data structures, or violate the code or data regions of other threads, or even the operating system kernel itself.

Hardware-based memory protection offers greater security and robust application code to solitary or group programmers by isolating the various regions of a program and using the integrated hardware MMUs on modern microprocessors. The MMU erects "walls" around code and data by defining memory segments for each and restricting access to those segments. Code segments are read-only, disallowing accidental self-modification. Data segment access is restricted to the current thread of execution or read-only data, and may be protected in the same manner as code.

When a program attempts to write over itself or violates data access restrictions, an exception or trap occurs, the executing program is interrupted, and the offending program location can be easily located and repaired. In multiprocess systems, one errant thread will be stopped from corrupting the entire application, and may be restarted or replaced without interrupting mission-critical operation.

Virtual addressing

Computer memory, while contiguous and amorphous at the device level, often gets laid out in computer systems on the basis of hardware requirements such as vector tables, memory-mapped I/O, and ROM/RAM address boundaries. It is the job of the operating system to make this "lumpy" collection of resources and address spaces appear "flat" and accommodating to programs, so that most software need not be customized to run on a given computer, thereby rendering it "portable."

The method by which an operating system can accomplish this leveling process is to give each program or process in a system its own virtual address space via the MMU. Then, each process can behave as though the entire computer memory space is at its complete disposal. And with memory protection (as above), the process does not risk corrupting the memory space of other processes in the system. The operating system maps or translates all program memory accesses, from the program's vision of memory (logical view) to the available collection of system memory resources (physical view) through the MMU, in a manner completely transparent to the process and its programmer. Combining virtual address translation with segmented memory protection yields the following benefits:

- Simplified generic programming models offer more straightforward development and porting.
- Better allocation and use of system resources can reduce system costs.
- Complete or selective isolation among system processes simplifies debugging, and when deployed, enhances reliability.

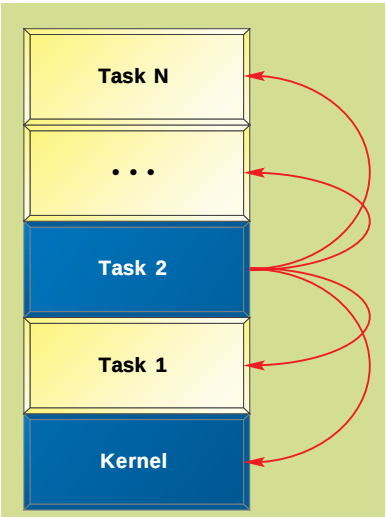

FIGURE 1: *Errant task violating memory divisions.*

How does it work?

Segment-based address translation is quite straightforward: The processor emits an address that must pass through the MMU to arrive on the system bus. The MMU "decodes" the address and determines if it lies within the bounds of a known logical address segment. If it does, and if the memory operation is allowed for the logical address in question, then an offset is calculated from the base of the logical address block, and that same offset is added to the base of the physical address block per the translation in the MMU. If that address exists in available physical memory, then it is passed to the system bus and the memory access cycle proceeds apace. If not, or if the operation is not permitted (e.g., writing into a read-only space), then the MMU interrupts the processor with an address exception. This entire process usually occurs in one CPU cycle.

Paged memory management and disk swapping

One of the first applications of memory management was to expand the available memory address space in the venerable mini-computer. In the days before the advent of semiconductor memory, main computer memory was built out of tiny ferrous rings (cores) strung together on grids of wire; core memory was difficult to

manufacture and expensive, so a system with even 4 kbytes was considered "high end." Even after the introduction of semiconductor RAM, memory pricing remained prohibitively high and made memory size the principal limit on application development.

As is the rule in software development, programs quickly grew to fill available memory (and to exceed it), and so a trade-off between expensive core (and later RAM) and less-expensive disk space emerged. Programs can only run from on-line RAM, so rotating media extended available RAM space as follows: Programs execute in the swap area. When execution reaches the boundary of this memory, the program memory fetch "falls off the edge," an exception or trap occurs, and the OS fetches the next block of executable code from disk. When execution jumps to code multiple blocks away in address space, the OS must calculate the RAM-to-disk mapping and retrieve the appropriate disk block.

The same basic procedure applies when data space is extended through swapping, but with the added necessity of writing modified blocks of data back to disk to ensure integrity.

Paged/swapped memory management use continues today; is present in Unix servers and workstations, Windows, Windows NT, OS/2, and Macintosh desktop machines; and is an available and desirable option in real-time operating system (RTOS) environments.

Even with the advent of cheap computer memory and huge address spaces, applications still grow to exceed the available resources that multimegabyte systems may need to run applications with gigabyte code and data requirements. Even in diskless systems, or systems with sufficient RAM and ROM that do not need swapping, the paged memory paradigm proves useful.

The benefits of using paged memory management are numerous:

- Reduced fragmentation of logical program memory (although physical memory is randomly fragmented, with no performance impact).
- Elimination of performance-intensive "free-chain" memory allocation.

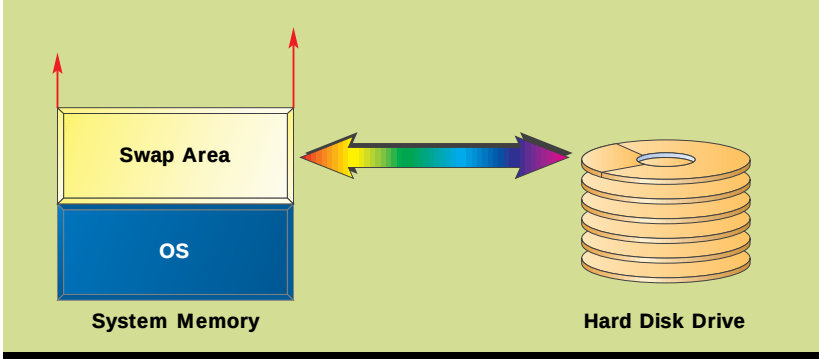

FIGURE 2: *Swapping to disk.*

- Simplified extension of growing memory regions (stack and heap) through on-demand page allocation prevents system crashes from unforeseen memory requirements.
- Uniform, efficient use of memory resources.
- Finer control over memory protection and memory sharing among processes offers more flexible system design, higher portability, and better reliability.

Paged memory management is also a key part of the Portable Operating System Interface for UniX (POSIX) process definition standard for open systems.

How it works

Due to the higher resolution of address translation, paged memory management is somewhat more complex than segmented models. As with segmented translation schemes, the MMU stands between the CPU and system memory, and the result of the logical-to-physical translation is comparable. For a 32-bit machine (e.g., the x86 in protected mode) with a 4k (4,096 bytes) page size, translation proceeds as follows (Figure 3): The least significant 12 bits of the address are taken as an offset into the page in question (12 bits yield 2^{12} addresses, or 4,096 bytes). The remaining 20 bits are divided into two groups of 10 bits. The first 10 bits can address up to 1k (1,024 bytes) translation tables, while the second 10 bits address up to 1k entries in each table. Since each entry represents 4k of real memory, this scheme handles a total address space of 4 Gigabytes, where each 4k page may be tagged as read-write, read-only, swappable, or with other salient characteristics. Small modifications in such a scheme can enable even larger addressable memory sizes.

Performance considerations

Unfortunately, there is no such thing as a truly free lunch. Using protection, virtual addressing, paging, and swapping induces a finite performance burden and carries some system overhead:

- *Address translation.* The process of address translation induces a small delay in the transmission of memory address information onto the system bus. This delay is never more than one CPU clock cycle and applies uniformly to all program code and data accesses.
- *Page table lookup.* In paged memory-management systems, the data structures (page tables) that support the address translation often do not fit into the on-chip MMU itself and end up in system memory. Memory access time can then slow address translation by multiples of system-memory bus cycle time, potentially degrading performance. All modern MMU architectures, however, support a translation caching scheme with "translation look-aside buffers" (TLBs) that keep the most frequently or most recently used mappings on-chip and, therefore, eliminate translation delays.
- *Context switch overhead.* When the operating system hands over control from one process to another, it must reprogram the MMU to adjust address translation to reflect the virtual address map for the new process. In a segmented MMU architecture, this switch can involve quite extensive manipulation of

MMU register sets and multiple address calculations, increasing context switch times (latencies) and degrading real-time response.

If an RTOS thoroughly integrates a paged-memory management scheme (initializing the complex of paged logical-to-physical address mappings at start-up or process load time), the impact on context switching is truly minimal, optimally involving the reloading of a single register to point to a new mapping structure, and the flushing of the MMU TLBs.

- *Swapping to disk.* The only significant performance impact in a memory-managed system comes from the use of disk-based page swapping. If a page of virtual memory is not present in physical RAM, then the operating system must fetch it from disk storage, inducing potentially long delays and reducing predictability of system performance. This effect can be minimized with the use of a fast, deterministic real-time file system and caching of the most frequently and recently used pages. It can be avoided entirely for key sections of code and data by "locking" them into physical memory.
- *Choosing hardware and software.* Most available CPUs used in communication applications today do offer integrated MMUs, including PowerPC, Intel 386/486 and Pentium, SPARC, and high-end 68000 (030/040/060). Highly integrated microcontrollers with peripheral sets that appeal to communication applications (like the Motorola QUICC devices) formerly evolved around 16-bit processor cores without MMUs. Semiconductor manufacturers, however, have recognized the need for memory management, and now offer the same complement of powerful devices built around 32-bit processors with integrated MMUs (like the MPC860 or 386EX).

Not all embedded real-time software, however, utilizes key MMU hardware, and, therefore, denies applications the benefits of memory management. In fact, many embedded RTOSes, kernels, and executives, ignore the presence of the MMU entirely, presenting only a flat memory model. This approach is common with RTOS software that covers a broad spectrum of target CPUs (often over a dozen), and, thus, supports the least common denominator of functionality among them. The 32-bit and 64-bit applications that you develop today with such a system inherit the limitations of decade-old hardware built into the kernel. If you and your customers are paying for the silicon that implements the MMU, shouldn't its presence offer more than just added power dissipation?

Memory management checklist

If your application development involves significant amounts of code, large development teams, high availability, and real-time response, you should ask the following questions about the OS that will support the design:

- Does the OS offer per process/task memory protection? If not, what recovery mechanisms exist for crashes after deployment?
- Is the OS also fully preemptive? If not, how can you prevent errant processes from bringing down the whole system?
- Is an available memory management facility segmented or paged? Is swapping available?
- Is an offered memory protection scheme deployable, or just present as a debugging tool? Remember, you deploy the kernel and your application, not the tools.
- Does an available memory management scheme significantly impact system performance? If so, what are the performance/reliability tradeoffs involved?
- Does the OS offer POSIX/Unix or other open standards compliance, both in terms of memory management models and general programming interface? If not, how is portability impacted?

More options

In the past, developers of embedded systems had few options for deploying high-availability/mission-critical applications. Code and team sizes were smaller, tools and kernels were almost always proprietary, and the CPUs employed in designs offered only small, flat address models. Exhaustive testing (and sometimes a wish and a prayer) produced acceptably reliable applications.

Today's projects, involving huge, international development teams deploying gargantuan executables of dozens of megabytes, combine diverse systems units in an open-systems "buy, not build" environment. This trend towards commercial off-the-shelf components, combined with microprocessors offering the performance of yesterday's supercomputers, confronts developers and integrators with greater reliability challenges. Memory management is a key tool to bringing the many pieces of a modern application together, and ensuring that they actually perform as one in the field.

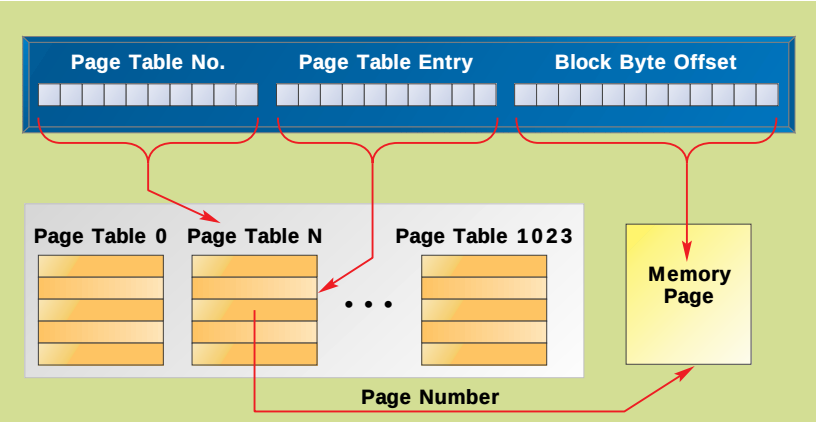

FIGURE 3: *How paged memory management works.*

William Weinberg is senior technologist for Lynx Real-Time Systems, where he focuses on embedded internetworking and real-time performance issues. Weinberg is a graduate of the University of California at Berkeley, and also attended the Universities of Rome and Pittsburgh for graduate studies. He can be reached at william@lynx.com.

Posted to LYNX REAL-TIME SOLUTIONS, INC web site with permission from COMMUNICATION SYSTEMS DESIGN, October 1997
©1997 MILLER FREEMAN, INC. All Rights Reserved.