

ANDREW S.
TANENBAUM
HERBERT
BOS

MODERN OPERATING SYSTEMS

Fourth Edition



MODERN OPERATING SYSTEMS

FOURTH EDITION

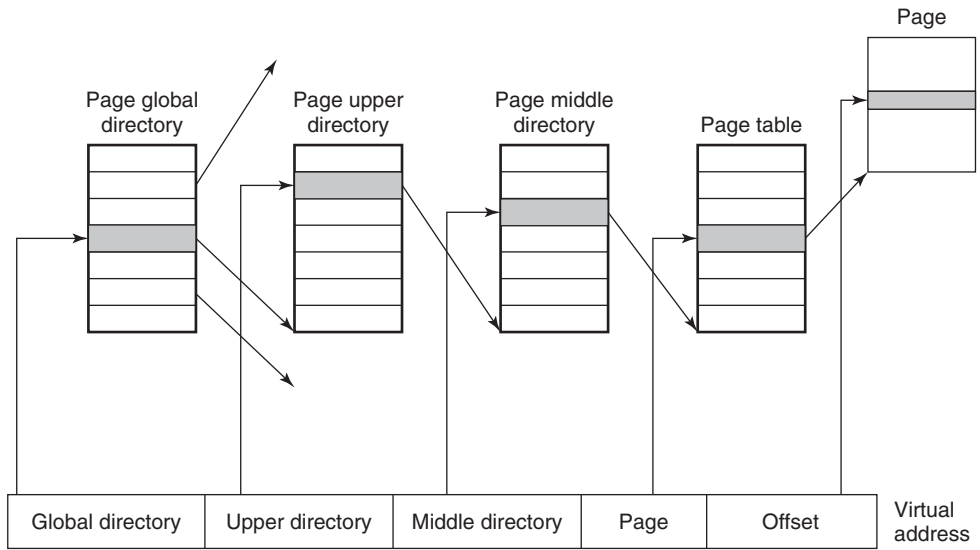


Figure 10-16. Linux uses four-level page tables.

Physical memory is used for various purposes. The kernel itself is fully hard-wired; no part of it is ever paged out. The rest of memory is available for user pages, the paging cache, and other purposes. The page cache holds pages containing file blocks that have recently been read or have been read in advance in expectation of being used in the near future, or pages of file blocks which need to be written to disk, such as those which have been created from user-mode processes which have been swapped out to disk. It is dynamic in size and competes for the same pool of pages as the user processes. The paging cache is not really a separate cache, but simply the set of user pages that are no longer needed and are waiting around to be paged out. If a page in the paging cache is reused before it is evicted from memory, it can be reclaimed quickly.

In addition, Linux supports dynamically loaded modules, most commonly device drivers. These can be of arbitrary size and each one must be allocated a contiguous piece of kernel memory. As a direct consequence of these requirements, Linux manages physical memory in such a way that it can acquire an arbitrary-sized piece of memory at will. The algorithm it uses is known as the buddy algorithm and is described below.

Memory-Allocation Mechanisms

Linux supports several mechanisms for memory allocation. The main mechanism for allocating new page frames of physical memory is the **page allocator**, which operates using the well-known **buddy algorithm**.

The basic idea for managing a chunk of memory is as follows. Initially memory consists of a single contiguous piece, 64 pages in the simple example of Fig. 10-17(a). When a request for memory comes in, it is first rounded up to a power of 2, say eight pages. The full memory chunk is then divided in half, as shown in (b). Since each of these pieces is still too large, the lower piece is divided in half again (c) and again (d). Now we have a chunk of the correct size, so it is allocated to the caller, as shown shaded in (d).

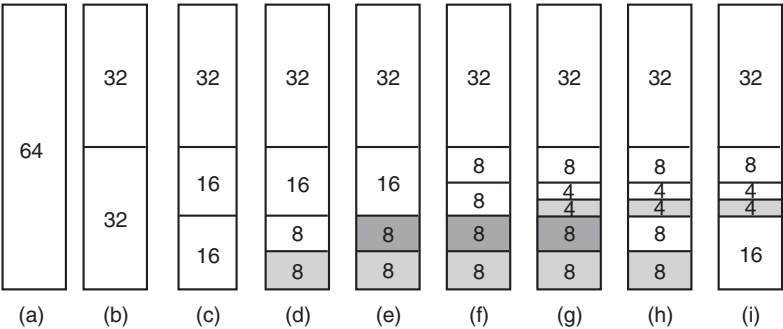


Figure 10-17. Operation of the buddy algorithm.

Now suppose that a second request comes in for eight pages. This can be satisfied directly now (e). At this point a third request comes in for four pages. The smallest available chunk is split (f) and half of it is claimed (g). Next, the second of the 8-page chunks is released (h). Finally, the other eight-page chunk is released. Since the two adjacent just-freed eight-page chunks came from the same 16-page chunk, they are merged to get the 16-page chunk back (i).

Linux manages memory using the buddy algorithm, with the additional feature of having an array in which the first element is the head of a list of blocks of size 1 unit, the second element is the head of a list of blocks of size 2 units, the next element points to the 4-unit blocks, and so on. In this way, any power-of-2 block can be found quickly.

This algorithm leads to considerable internal fragmentation because if you want a 65-page chunk, you have to ask for and get a 128-page chunk.

To alleviate this problem, Linux has a second memory allocation, the **slab allocator**, which takes chunks using the buddy algorithm but then carves slabs (smaller units) from them and manages the smaller units separately.

Since the kernel frequently creates and destroys objects of certain type (e.g., *task_struct*), it relies on so-called **object caches**. These caches consist of pointers to one or more slab which can store a number of objects of the same type. Each of the slabs may be full, partially full, or empty.

For instance, when the kernel needs to allocate a new process descriptor, that is, a new *task_struct*, it looks in the object cache for task structures, and first tries to find a partially full slab and allocate a new *task_struct* object there. If no such

slab is available, it looks through the list of empty slabs. Finally, if necessary, it will allocate a new slab, place the new task structure there, and link this slab with the task-structure object cache. The `kmalloc` kernel service, which allocates physically contiguous memory regions in the kernel address space, is in fact built on top of the slab and object cache interface described here.

A third memory allocator, `vmalloc`, is also available and is used when the requested memory need be contiguous only in virtual space, not in physical memory. In practice, this is true for most of the requested memory. One exception consists of devices, which live on the other side of the memory bus and the memory management unit, and therefore do not understand virtual addresses. However, the use of `vmalloc` results in some performance degradation, and it is used primarily for allocating large amounts of contiguous virtual address space, such as for dynamically inserting kernel modules. All these memory allocators are derived from those in System V.

Virtual Address-Space Representation

The virtual address space is divided into homogeneous, contiguous, page-aligned areas or regions. That is to say, each area consists of a run of consecutive pages with the same protection and paging properties. The text segment and mapped files are examples of areas (see Fig. 10-13). There can be holes in the virtual address space between the areas. Any memory reference to a hole results in a fatal page fault. The page size is fixed, for example, 4 KB for the Pentium and 8 KB for the Alpha. Starting with the Pentium, support for page frames of 4 MB was added. On recent 64-bit architectures, Linux can support **huge pages** of 2 MB or 1 GB each. In addition, in a **PAE (Physical Address Extension)** mode, which is used on certain 32-bit architectures to increase the process address space beyond 4 GB, page sizes of 2 MB are supported.

Each area is described in the kernel by a `vm_area_struct` entry. All the `vm_area_structs` for a process are linked together in a list sorted on virtual address so that all the pages can be found. When the list gets too long (more than 32 entries), a tree is created to speed up searching it. The `vm_area_struct` entry lists the area's properties. These properties include the protection mode (e.g., read only or read/write), whether it is pinned in memory (not pageable), and which direction it grows in (up for data segments, down for stacks).

The `vm_area_struct` also records whether the area is private to the process or shared with one or more other processes. After a fork, Linux makes a copy of the area list for the child process, but sets up the parent and child to point to the same page tables. The areas are marked as read/write, but the pages themselves are marked as read only. If either process tries to write on a page, a protection fault occurs and the kernel sees that the area is logically writable but the page is not writeable, so it gives the process a copy of the page and marks it read/write. This mechanism is how copy on write is implemented.