第 15 章　进一步关于内存管理的主题

15.1　简　　介

本章讨论 3 个重要主题。第一个是 Mach 虚存体系结构，它有些特有的特征，比如由用户级任务完成内存管理的许多功能。第二个是关于多处理机系统中的快表一致性问题。第三个是如何高效而又正确地使用虚地址缓存的问题。

15.2　Mach 的内存管理设计

Mach 操作系统是由卡内基·梅隆大学在 20 世纪 80 年代中期开发的，它的内存体系结构几乎与 14 章中介绍的 SunOS/SVR4 VM 设计同时发展起来的。尽管这两个系统使用不同的术语来描述它们的基本抽象概念，它们在目标、设计以及实现方面有着许多相似之处。许多 SVR4 VM 的对象和函数在 Mach 中都可以找到几乎完全相同的等价物。在下面的小节中，我们将讨论 Mach 2.5 的 VM 子系统设计。Mach3.0 中这部分的变化较小，而且这些变化不涉及我们所讲述的范围。

由于 Mach 和 SVR4 在内存体系结构方面的相似性，我们不准备就 Mach VM 设计的各个方面都展开论述，我们只对它们进行比较分析，并侧重于 Mach 系统中那些 SVR4 不具备的特征。

15.2.1　设计目标

与 SVR4 相似，Mach VM 的设计也是不满于 4.3BSD 内存结构的限制而产生的。4.3BSD 的内存结构受 VAX 机硬件的影响很深，很难移植。而且其功能也非常原始，局限于对分页的支持，缺少除只读正文段共享外的其他内存共享机制,而且 4.3BSD 的内存管理不能扩展到分布式环境中。尽管 Mach 提供了与 4.3BSD UNIX 之间的全二进制兼容，但 Mach 的目标是提供更为丰富功能集，这包括：

- 相关或无关进程间 copy-on-write 和读写式内存共享。
- 内存映射文件访问。
- 支持大容量，稀疏地址空间。
- 不同机器的进程间共享内存。
- 用户可控页面替换策略。

Mach 也将所有机器相关代码分离到一个很小的 pmap 层。这使得在一个新结构上移植 Mach 非常容易，只需重写 pmap 层。其余部分的代码都是机器无关的，而且不以任何特定的 MMU 结构为模型。

Mach VM 设计中的一个重要目标就是将许多 VM 的功能移出内核。这出自 Mach 设计的初衷，Mach 采用微内核结构，由用户级任务提供许多附加的内核功能。Mach VM 也将诸如换页这样的功能移到外部(用户级)任务。

最后，Mach 将内存管理和进程间通信(IPC)子系统集成在一起。这有两个优点，虚存设施利用 Mach IPC(见 6.9 节)的位置无关性可以将其透明地扩展到分布式环境中。15.5.1 节给出了一个例子，一个用户级程序如何为不同机器上的进程提供共享内存。反过来，IPC 子系统可以利用 VM 子系统支持的 copy-on-write 共享实现大消息的快速传递。

以后的论述中将时常山现 Mach 中的 5 个基本抽象概念，即任务，线程、端口、消息和内存对象。任务是一组资源的集合，包括一个地址空间和若干端口，每个任务中可以运行一个或多个线程。一个线程是程序中的一个控制点，它是任务中可运行、可调度的实体。Mach 用只有一个线程的任务替代 UNIX 中的进程。Mach 的任务和线程在 3.7 节中有述。一个端口是

一个受保护的消息队列。一个端口可以有许多任务持有其发送权，但只有一个任务拥有其接收权。一个消息是一组有类型数据的集合。它的大小可以从几个字节到整个地址空间。Mach 的端口和消息在 6.4 节中有述。内存对象是虚存页面的后备存储，本章论述这个概念。

15.2.2 编程接口

Maeh 有许多与内存管理相关的操作，分为四个基本类别[Rash 88]：

·内存分配 用户可以调用 vm_allocate 或 vm_map 分配一个或多个虚存页，前者是全 0 页面，后者的后备存储是某个特定的内存对象(例如，一个文件)。分配时并不立即占用资源，直到首次访问它们时，Mach 才分配物理页面。调用 vm_deallocate 释放虚存页。

·保护 Mach 支持页面上的读，写，执行权限，但它们的实现受硬件限制。许多 MMU 不支持执行权限。在这种系统中，系统允许在任何可读页面上执行程序。每个页面都有一个当前和最大保护权限。最大保护权限被设置后，只能降低该值。当前保护权限不能超过最大保护权限，调用 vm_protect 更改两种类型的保护权限。

·继承 每个页面都有一个继承值，它决定当任务调用 task_create 生成新任务时如何处理页面。该属性可取如下 3 个值之一(图 15-1)：

VM_INHERIT_NONE 子任务不继承该页面，页面不在其地址空间中出现。

VM_INHERIT_SHARE 父、子任务共享该页面。两个任务访问页面的同一个副本，任何修改对双方都是可见的。

VM_INHERIT_COPY 子任务需要自己的页面副本。Mach 用 copy-on-write 机制实现该功能，仅当子进程或父进程首次修改页面时才复制数据。它是所有新分配页面的缺省继承值。

调用 vm_inherit 修改一组页面的继承值。要特别注意，继承属性与当前任务间如何共享该页面无关。例如，任务 A 分配了一个页面，其继承值为 VM_INHERIT_SHARE，然后生成子任务 B。任务 B 将其继承值改为 VM_INHERIT_COPY，然后生成子任务 C。任务 A 和任务 B 以读写方式共享页面,而任务 B 与任务 C 则采用 copy-on-write 方式共享(见图 15-2)。这一点与 SVR4 不同，在 SVR4 中继承的方式与页面当前映射方式(共享或私有)一致。

·其他 通过 vm_read 和 vm_write 调用，一个任务可以读写其他任务的页面。这主要用于调试器和性能分析工具。调用 vm_regions 返回地址空间中所有内存区的信息，调用 vm_statistics 返回有关虚存子系统的统计数据。

图 15-2 同一页面上的各种共享方式

15.2.3 基本抽象概念

Mach 采用面向对象方法设计，通过一套定义好的接口访问这些基本抽象概念。图 15-3 所示是较为重要的内存管理原语的高层视图。最上层的对象是地址映射图，用 struct vm_map 描述，保存有一个地址映射图表项的双向链表和一个 hint 指针，该指针指向最近发生失效的表项。每一个地址映射图表项(struct vm_map_entry)对应一片连续的虚存空间，其中的页面具有相同的保护权限和继承属性，由相同的内存对象管理。

通过 vm 对象提供的接门访问内存对象的页面。内存对象[Youn 87]是一组字节的抽象表示，其上可支持诸如读、写等若干操作。这些操作是由页面的数据管理器或 pager 执行的。pager 是一个任务(用户级或内核级)，它管理一个或多个内存对象。文件，数据库，网络共享内存服务器(15.5.1 节)等都是内存对象。

内存对象用对象的 pager 持有的端口表示(pager 对这个端口有接收权)。vm 对象对该端口有一个引用，即发送权，并通过它与内存对象通信，15.4.3 节中将详细介绍。vm 对象维护一个链表，表中包括所有内存对象的驻留页面。该链表加速了诸如释放对象，清除或刷新所有内存对象页面之类的操作。

每个内存对象都唯一地与一个 vm 对象相关联。如果两个任务都将同一个内存对象映射到自己的地址空间，它们共享 vm 对象，详见 15.3 节。vm 对象为实现共享维护一个引用计数。

与 SVR4 之间的相似程度是惊人的。vm_map 对应 struct as，vm_map_entry 对应 structseg，pager 的作用与段驱动程序相似(除了 pager 是一个独立的任务)，而 vm 对象和内存对象一起说明文件之类的特定数据源。一个重要的不同之处是它没有每页保护数组。如果用

图 15-3　描述 Mach 地址空间的对象

户更改了内存区(这里内存区指地址映射图表项)的一个页面子集的保护权限，Mach 就要将内存区分裂为两个①映射到同一内存对象的内存区。与此相似，其他操作则可能将相邻的内存区合并起来。

还有两个重要数据结构——驻留页面表和 pmap。驻留页面表是一个数组(struct vm_page[])，每一项对应一个物理页面。物理页面的大小是硬件页面大小的 2 的整次幂。物理内存被用作内存对象内容的缓存。它们的名字空间用<object，offset>偶对表示，它指明页面所在的内存对象及其起始偏移。该表中的页面都在如下 3 个列表中：

·内存对象列表链接所有属于同一对象的页面，加速释放对象以及 copy-on-write 操作。

·分页守护进程维护一组内存分配队列。每个页面都在如下 3 个队列之一—活动，失活，空闲。

·object/offset 哈希链，用于在内存中快速查找页面。

vm_page[]数组与 SVR4 中的 page[]数组十分相似。最后，每个任务有一个机器相关的 pmap 结构(与 SVR4 的 HAT 层相似)，说明由硬件定义的虚址到实址的地址转换映射表。该数据结构对系统的其他部分不可见，而且只能通过一些接口访问。pmap 接口只假设硬件有一个简单的分页 MMU，如下是 pmap 层支持的一些函数：

·pmap_create()在系统使用一个新地址空间时调用。它创建一个新的 pmap 结构并返回其地址。

·pmap_reference()和 pmap_destroy()对 pmap 对象上的引用计数增 1 或减 1。

·pmap_enter()和 pmap_remove()加入或删除一个地址转换。

·pmap_remove_all()删除页面的所有地址转换。由于一个页面可能由多个任务共享(或映射到一个任务地址空间的多处)，它可能有多个地址转换。

·pmap_copy_on_write()将页面的所有地址转移的保护权限降低为只读。

·pmap_activate()和 pmap_deactivate()在上下文切换期间调用，更换处理器的活动 pmap。

15.3　共享内存设施

Mach 支持相关或无关任务间的读写和 copy-on-write 方式共享。在 task_create 期间，任务从其父任务处继承内存。这样一个任务就与其后代共享它的内存区，无关任务通过映射同一内存对象共享内存。本节中论述这些设施。

15.3.1　copy-on-write 共享

task_create 在复制标记为 VM_INHERIT_COPY 的内存区时使用 copy-on-write 方式共享页面。子任务须有自己单独的内存区副本，但内核并不立即复制它们，并利用这一操作来优化内存操作。当父任务或子任务修改内存区的页面时，它们会触发一个内核陷阱，此时，内核生成页面的一个副本，并且修改地址映射图，每个任务都引用自己的那个副本。这样，只有那些父任务或子任务修改的页面才需要复制，内核节省了大量的工作。

内核在实现这种共享时，使两个任务中相应的 vm_map_entry 引用同一个 vm_object。同时在 vm_map_entry 中设置一个标志，说明是 copy-on-write 共享。最后，调用 pmap_copy_on_write()对父任务和子进程的内存区进行写保护，以后就可以捕获父进程或子进程的写操作了。图 15-4 给出一种典型的情况，任务 A 创建了任务 B，它们都以 copy-on-write

方式共享内存区，该内存区有 3 个页面，由共享的 vm 对象管理。

图 15-4 copy-on-write 继承

当任何一个任务修改页面时，由于保护权限已降低到只读，系统会产生一个失效。失效处理函数分配一个新的物理页面，并通过复制原页面的内容初始化它。然后，处理函数建立新映射，每个任务都引用自己的页面副本。与此同时，它们还共享那些尚未更改的页面。

Mach 应用一个称为影子对象的概念实现这种映射，图 15-5 给出了任务 A 修改了页面 1 而任务 B 修改了页面 3 时的情景，每个任务都获得一个影子对象来管理任务被修改的页面。影子对象有一个指针回指它们镜像的对象，也就是原始对象。

图 15-5　使用影子对象管理被修改的页面

为了处理页面失效，内核自顶向下搜索影子链。任务 A 在影子对象中找到页面 1，而在原始对象上找到页 2 和 3，任务 B 在影子对象中找到页面 3，而在原始对象是找到页面 1 和 2。

随着任务不断创建子任务，它生成长长的影子对象链。这不仅浪费了资源，而且由于内核必须在这些链上遍历而降低了页面失效处理的速度。因此 Mach 中使用一算法用来探测这种情况，并在可能时消解影子链。如果链中某对象的所有对象均在其上的对象中出现，该对象就可以去除掉了。但这种简约方法并不总是适用的，如果对象上的某些页面已被换出，对象就不能删除了。

copy-on-write 共享也用于传递大消息。任务可以发送 ont-of-line 内存消息(见 6.7.2 节)。应用程序利用这一设施实现大量数据的发送而无需复制物理内存。内核在新进程中创建新的 vm_map_entry，将那些页面映射其地址空间上。此时它与发送者以 copy-on-write 方式共享这些页面，这与父子任务间的 copy-on-write 共享相似。

但它们之间有一个重要的差别，当数据在传递过程中时，内核将这些数据临时映射到自己的虚空间。系统在内核的映射表中创建一个 vm_map_entry 与发送者共享页面，在数据映射到接收者之后，内核删除那个临时映射。这就防止了在接收任务访问消息之前发送任务可能对消息进行的修改。这还防止了一些其他异常事件，比如发送任务终止等。

15.3.2　读写共享

task_create 在复制标记为 VM_INHERIT_SHARED 的内存区时使用读写共享。此时只有内存区的一个副本，共享它的所有进程可以自动看其上的修改。这修改包括数据，以及保护权限和其他属性。后者可能将内存区分成多个子内存区，共享原内存区的进程仍以读写方式共享每个子内存区。

这是一种完全不同的共享方式，内核要用一种不同的机制实现它。Mach 使用共享映射图的方式描述共享内存区。共享映射图本身是一个 vm_map 结构，有一组标志描述共享内存，还有一个记录共享该内存区任务的引用计数。其上维护有一个 vm_map_entry 链表，开始时只有一个表项。

图 15-6 所示为共享映射图的实现，每个 vm_map_entry 或者指向一个 vm_object，或者指向一个共享映射图。在这个例子中，共享内存区包括 3 个页面，最初均为可读可写。之后，一个任务调用 vm_protect 置页面 3 为只读，内存被分为两部分，如图 15-7 所示。利用共享映射图机制，在共享内存区上实现这些操作就很容易了。

15.4　内存对象和 Pager

一个内存对象就是某个数据源，比如一个文件。一个 pager 是管理一个或多个内存对象的任务，通过与内核交互完成对象与物理内存间的数据移动。内核，pager，以及用户任务间通过预先定义好的接口进行交互。

15.4.1　内存对象初始化

要访问内存对象中的数据，用户任务首先要获得相应端口的发送权。只有端口的属主才

能授权给其他任务，任务只能从对象的 pager 处获得发送权。这些操作超出了 VM 子系统的范围，而且可能还包含用户任务与 pager 间(可能还有其他实体)的进一步交互。例如，v 节点

图 15-7　分裂-共享内存区

pager②管理文件系统对象，允许用户以文件名打开一个文件。当一个任务打开文件时，pager 将相应文件端口的发送权返回给任务。

一旦任务获得了端口(获得端口是指获得了端口的发送权)，它就调用

```
vm_map(pager_task, base_addr, size, mask,
    flag, memory_object_port, offset,
    copy, cur_prot, max_prot, inheritance);
```

将内存对象映射到自己的地址空间。

这与 SVR4 中的 mmap 很相似，它将内存对象[offset，offset+size)上的字节映射到任务的[base_addr，base_addr+size)空间上。内核根据 flag 决定是否将其映射到不同的基址上，函数返回内存对象最终被映射的地址。

对象首次被映射时，内核为该对象另外创建两个端口——一个是控制端口，用于 pager 向内核发送缓存管理请求；另一个是名字端口，用于标识该对象，其他任务可以用 vm_regions 调用获取对象的信息。内核是这两个端口的属主，拥有它们的发送权和接收权。接下来，内核调用

```
memory_object_init(memory_object_port,
    control_port,
    name_port,
    page_size);
```

请求 pager 初始化内存对象。这样，parer 就获得了对请求端口和名字端口的发送权。图 15-8 是初始化后的情景。

图 15-8　同 pager 通信

15.4.2　内核与 pager 间的接口

这些通道建立好之后，内核和 pager 就利用它们互相发送请求。每个请求都是通过远程过程调用完成的，即将带有参数的请求消息发送到某个端口中。内核通过内存对象端口向 pager 发消息，而 pager 则使用 pager 应答端口向内核发消息。系统为这些消息的交换提供一组过程式的高级函数接口。某些调用是内核用来同 pager 通信的：

```
memory_object_data_request(memory_object_port,
    control_port, offset,
    length, desired_access);
memory_object_data_write(memory_object_port,
    control_port, offset,
    data, data_count);
memory_object_data_unlock(memory_object_port,
    control_port, offset,
    length, desired_access);
```

内核提供一套接口供 pager 用于缓存管理，它包括如下函数：

```
memory_object_data_provided(control_port, offset, data,
    data_count, lock_value);
memory_object_lock_request(control_port, offset, size,
    should_clean, should_flush,
    lock_velue, reply_port);
```

```
memory_object_set_attributes(control_port, object_ready,
    may_cache_object, copy_strategy);
memory_object_data-unavailable(control_port,
    offset, size);
```

这些函数最后都变为一个异步消息。如果内核向 pager 发送请求，pager 就利用内核接口函数发送一个消息应答。下面将给出一些内核与 pager 间的交互过程。

15.4.3 内核与 pager 交互

内核与 pager 间的交互主要有四大类，

·页面失效 在处理页面失效时，内核遍历地址映射图，寻找发生失效的 vm 对象，其上有一个可提供所需数据的内存对象的引用。然后内核调用 memory_object_data_request()，指定所需的页面范围以及访问类型。pager 调用 memory_object_data_provided()将数据在 pager 地址空间中的指针返回给内核。pager 决定如何提供所需的数据——例如，可以从文件中读取，也可以从某种基于网络的存储设备中读取。

·换出 当内核准备回写一个已修改的页面时，它调用 memory_object_data_write()将页面发送给 pager。pager 在成功地保存了页面后，调用 vm_deallocate 释放缓存资源。

·保护 内核调用 memory_object_data_unlock()请求提高页面的访问权限(比如对当前只读页面的写访问)。相反，pager 也可以调用 memory_object_lock_request()降低页面的访问权限(例如，设置页面为只读)。

·缓存管理 pager 用带有 should_flush 标志的 memory_object_lock_request()调用请求内核清除其数据副本，回写所有的修改。如果 pager 使用带有 should_clear 标志的 memory_object_lock_request()调用，内核就要回写指定页面范围内的任何改动，但它可以继续使用所缓存的数据，在这两种情况下，内核都像是换出操作一样调用 memory_object_data_write()回应。

15.5 外部 pager 和内部 pager

内核 pager 接口与传统实现有很大不同。Mach 的 pager 是独立的内核任务或用户级任务，内核与任务之间通过消息传递机制进行交互，尽管这似乎有点笨拙而且复杂，但它还是有着非常显著的优点。用户任务可以实现各种 pager，提供对不同种类的二级存储设备的支持，SVR4 也可以有多个段驱动器，但内核对每一个都要有显式地了解和控制，无法加入用户定义的段驱动程序。而在 Mach 中，内核不对用户级任务有什么特殊的了解，它仅是简单地通过 vm 对象访问 pager。

使用消息传递带来了位置透明性及可扩充性等优点。pager 任务不必在同一台机器上。它可运行于网络上的任何一个远程节点，而在本地机上，只需用一个代理端口表示相应的内存对象即可。这样 Maeh 的内存管理就可以扩充为这样一种配置，多个处理机通过消息传递来共享内存。下一小节将给出这样一例子。

用用户任务实现的 pager 称为外部 pager。Maeh 同时还提供一个缺省 pager，它是内部的(即，用内核任务实现)③。该 pager 管理所有没有永久性后备存储的临时内存对象。这些对象可分为两种类型：

·影子对象，包括内存区中已修改的页面，它们以 copy-on-write 方式共享。

·全 0 内存区，比如栈，堆，未初始化数据等等。通过调用

```
vm_allocate(target_task, address, size, flag);
```

创建全 0 内存区。

内核在系统初始化期间调用 memory_object_create()创建缺省 pager。它管理的对象最初没有内存。当这些页面被首次访问时，内核调用 memory_object_data_request()，然后

pager 返回 memory_object_data_unavailable()，说明页面须全部置 0。

15.5.1 一个网络共享内存服务器

网络共享内存服务器经常用来展示外部 pager 接口的多才多艺。服务器是一个用户任务，客户任务可以通过网络共享其地址空间上的一组页面。在客户发生页面失效时，它可以向其提供数据并同步对共享数据的访问，保证一个客户所做的修改对其他所有客户都是可见的。客户察觉不到同一帆器上不同任务共享内存与不同机器上共享内存的差异。

在单机上内存共享是很自然的。内存中仅有数据的一个副本，客户对其所做的任何修改对其他所有任务都是立即可见的。不同机器上的任务间共享数据是很复杂的，由于同一页面在每台机器上都有一个副本，完成它们的同步非常困难。

让我们考虑一下在不同机器上的两个任务共享同一页面的情况。共享内存服务器可以运行于其中任何一台机器或另外一台机器上。图 15-9 所示是客户一开始如何将内存对象映射到自己的地址空间。每个客户与服务器的交互过程如下：

图 15-9 两个客户连接到共享内存服务器的网上

1.客户从服务器获得内存对象的端口。

2.客户调用 vm_map 将对象映射到自己的地址空间。

3.内核调用 memory_object_init()向服务器发送一个新对象初始化消息，同时传递给服务器向内核监听的控制端口的发送权。

4.服务器调用 memory_object_set_attribute()通知内核对象就绪。

5.内核完成 vm_map 调用，恢复客户运行。

第 5 步不必等到第 4 步发生，但直至对象就绪前客户将不能访问数据(访问将被阻塞)。图 15-10 所示是任务 T1 写页面时交互过程：

图 15-10 任务 T1 试图写页面

1.任务 T1 发生写失效，调用内核中的失效处理函数，

2.内核调用 memory_object_data_request()向服务器请求一个可写的页面副本。

3.服务器调用 memory_object_data_provided()返回页面。

4.内核更新任务 T1 的地址转换表并恢复其运行。

任务 T1 现在可以修改页面了，但服务器尚不知道它所做的修改。现在看任务 T2 读这个页面时将会怎样(图 5-11)：

图 15-11 任务 T2 试图读页面

1.任务 T2 发生读失效，调用内核 B 的失效处理函数。

2.内核 B 调用 memory_object_data_request()向服务器请求一个只读的页面副本。

3.服务器知道节点 A 上有一个可写的页面副本。它随即通过内核 A 的控制端口调用 memory_object_lock_request()，其中设置了 should_clean 标志，同时 lock_value 的值为 VM_PROT_WRITE④。

4.内核 A 将页面上的保护权限降为只读，并调用 memory_objeet_data_write()向服务器回写已修改的页面。

5.现在服务罪有了页面的最新版本。它调用 memory_object_data_provided()发送给内核 B。

6.内核 B 更新任务 B 的地址转换表，并恢复任务 T2 运行。

显而易见，这是一个非常慢的过程。如果两个任务频繁地修改页面，它们就不断地重复这些步骤，造成大量的 IPC 消息，不断地从一个节点向另一个节点复制页面。这些代价的大部分是跨网络同步数据问题所固有的。注意，每次写失效将造成两次跨网络复制页面——第一次是从另一个客户到服务器，然后是从服务器到失效客户。也许我们可以想到，如果服务器以某种方式通知一个客户直接发送页面给另一个客户，这样就可以减少到一次复制操作

了。然而这样会破坏模块化的设计，并在多于两个客户时，造成相当复杂的交互过程。如果服务器与其中的一个客户运行于同一机器，就会减少一次复制操作，而且还会减少网络上的其他消息流量。

除了这些缺点，这个例子很好地说明了，消息传递和内存管理的紧密结合将使我们可以构造强大的，功能丰富的应用程序。另一个例子[Subr 91]是一个管理可丢弃页面的外部pager，用于自行完成垃圾收集工作的应用。pager 接收客户任务关于哪些页面可丢弃的信息并重新刷新这些页面，籍此来影响页面的替换，这会为系统释放出许多可用的内存。

15.6 页面替换

Mach 的页面替换算法不同于 SVR4 和 4.3BSD 的算法，它们都采用双表针时钟算法(见15.5.3 节)。Mach 的算法称为多选 FIFO[Dray 91]。如图 15-12 所示，它使用了 3 个 FLFO(先入先出)链表——活动，失活和空闲。

图 15-12 页面替换

页面以如下方式在链表间转移：

1.页面的首次访问产生一个页面失效。失效处理函数将空闲表表头的页面删除，用相应数据对其进行初始化(全部置 0 或从内存对象中读取数据)。然后将页面加到活动链表的末尾。最终，随着该页面前面的页面不断失活，该页面会转移到活动链表的表头。

2.当空闲内存低于一个阈值(vm_page_free_min)时，系统唤醒 pagedaemon。它从空闲链表的头部转移一些页面到大活链表的尾部，并清除这些页面硬件地址转换映射中的引用位。

3.pagedaemon 还检查失活链表头部的若干页面。那些被设置了引用位的页面被重新放回给活动链表。

4.如果 pagedaemon 发现某个页面的引用位仍为 0，说明页面在失活链表期间未被访问过，系统将其移到空闲链表尾部。如果页面已修改，系统要先将其回写到它的内存对象上。

5.如果访问了空闲链表上页面，仍可以回收该页面。在这种情况下，它返回到活动队列的末尾，否则，页面会逐渐转移到空闲链表的头部，最终由系统重新使用。

与 BSD 的方法类似，在不支持引用位的系统上，Mach 通过清除页面映射模拟引用信息。此外，还有几个参数影响每次 pagedaemon 被唤醒时它的工作量，它们列在表 15-1 中。

表 15-1 Mach pagedaemon 的参数

参数	使用	缺省值(页)
vm_page_free_min	当空闲链表的页面数低于这个值时，系统唤醒 pagcdaemon	25+physmem/100
vm_page_free_target	在主闲链表达到这个值之前，系统将失活链表中的页面转移到空闲链表	30+physmem/80
vm_page_free_reserved	为 pagedaemon 保留的空闲页面数	15
vm_page_inactive_target	在失活链表达到这个值之前，系统将活动链表中的页而转移到失活链表	physmem*2/3

尽管 Mach 的数据结构与算法在某种程度上与 SVR4 和 4.3BSD 的双表针时针算法不同,但效果相似。失活链表对应前后表针之间的页面。在上面的介绍中，步骤 2 与前表针的功能相仿，清除被扫描页而的引用位。步骤 3、4 完成后表针的功能，检查失活队列头部的页面。

最大的不同是 pagedaemon 如何选择移动到失活队列的活动页面。在时针算法中，不是根据页面的使用模式，而是根据页面在物理内存中的位置顺序挑选页面。而在 Mach 中，活动链表也是一个 FIFO，pagedaemon 可以选择最早被调用的页面，这与时针算法相比是一个改进。

Mach 不提供可替代的替换策略。像数据库这样的应用程序没有非常好的访问局域性，不合适使用类 LRU 的替换策略。[McNa 90]律议对外部 pager 接口进行简单扩充，允许用户级任

务选择自己的替换策略。

15.7 分　析

Mach 的 VM 体系结构设计精湛，有些非常先进的功能。它的大部分功能与 SVR4 相似，比如 copy-on-write 共享，内存映射文件访问，以及对大型的稀疏地址空间的支持等。与 SVR4 相似，它也采用面向对象的方法，使用若干对象来描述模块化的编程接口。它将机器无关代码和机器相关代码分离，并将机器相关代码封装在 pmap 层，定义若干接口函数访问该层。在移植到新硬件结构时，只需重写 pmap 层。

此外，Mach VM 还有一些 SVR4 中没有的功能，它将共享与继承分开，提供了更为灵活的共享内存设施。它还将内存管理与进程间通信集成起来。IPC 使用 VM 传递大消息(大到任务的整个地址空间)，利用 copy-on-write 机制高效地传递大消息。VM 使用 IPC 进行交互，从而使它的对象具有位置无关性，而且可以将 VM 的设施扩展到分布式环境中。特别是用户级任务可以管理内存对象的后备存储，内核通过 IPC 与这些任务通信。这种结合营造了一个高度灵活的环境，可以有多个用户级(外部)pager 同时共存，提供不同类型的分页操作。网络共享内存管理器就是一个非常好的使用这一接口增添新功能的例子。

但 Mach 也存在许多缺点，许多缺点与 SVR4 的相似。VM 系统比 BSD 的设计更大、更慢、而且更复杂。它使用了更多和更大的数据结构。由于该设计要将机器相关指令最小化，它不能为某种特定的 MMU 结构进行必要的优化。

此外，消息传递机制给系统增加了相当大的负担。在某些情况下，可以通过优化内核到内核的消息传递减轻这一代价。但不管怎样，消息传递要比简单的函数调用复杂得多。除了网络共享内存管理器外，外部 pager 并没得到广泛使用。外部 pager 的灵活和用途广泛是否能干衡它的高代价，这是人们一直的疑问。基于 Mach 的主要商业系统，Digital UNIX，不支持外部 pager，也不开放 Mach 的 VM 接口。它的 VM 子系统在许多方面都不同于 Mach。

15.8 4.4BSD 的内存管理

在 13.4 节中我们介绍了 4.3BSD 的内存管理模型。对于那些 4.3BSD 当时要支持的机器而言，它的设计是有效的。在设计 4.3BSD 的那个时候，一般的计算机都是些大型的集中式分时系统，用户通过以串行线连接的终端同时使用系统。它一般有较大的磁盘空间(几百个 mb)，但处理器较慢(1-2 mips)，而且内存容量小(4mb 就相应大了)且价格高。在 UNIX 支持网络时，远程文件系统尚未流行，大多数都使用本地磁盘作为文件系统和交换区。4.3BSD 虚存模型通过额外的存储或 I/O 负担优化对内存的使用。

20 世纪 90 年代初，当 4.4BSD 正在设计时，这种情况发生了很大的变化。用户一般都专用一台有大量内存(32mb 很常见)和高速处理器(有几十个 mips)的桌面工作站。另一方面，像 NFS 之类的网络文件系统流行开来，用户可以将自己的文件保存在一个集中式的文件服务器上，而工作站只需很少的本地磁盘或者根本没有磁盘。4.3BSD 的内存管理设计已极不适用于这样的环境，于是 4.4BSD 使用了一个新模型替代它[McKu 95]。

4.4BSD 的内部 VM 框架基于 Mach 的 VM 结构。而其外部接口更近于 SVR4。4.4BSD 没有开放外部 pager 及内存对象等概念，相反它提供了 mmap 系统调用，它的语法与 SVR4 的相似(见 14.2 节)：

```
paddr = mmap(addr, len, prot, flags, fd, off);
```

它建立进程地址空间[paddr, paddr+len)与文件字节区[off, off+len)之间的映射，fd⑤表示该文件。和 SVR4 一样，addr 是用户建议的映射地址，prot 指定保护权限(可以是 PROT_READ，PROT_WRITE 和 PROT_EXECUTE 的组合)。标志 MAP_SHARED，MAP_PRIVATE 和 MAP_FIXED 与 SVR4 的意义相同。

4.4BSD 的 mmap 还有一些附加功能。flags 参数必须包含 MAP_FILE(映射文件或设备)或 MAP_ANON(映射匿名内存)。此外，还有两个标志——MAP_INHERIT，指定在 exec 系统调用后，保留相应的映射；MAP_HASSEMAPHORE，指定相应的内存区中可能有信号量。

进程可以通过两种方式共享内存。它们将同一个文件映射到各自的地址空间里，在这种情况下，文件是这片内存区的初始内容和后备存储。另一种方式是，进程映射一片匿名内存区，并用一个文件描述符与其关联。之后，进程将该描述符传递给其他要共享这片内存区的进程。这样做省去了映射文件的负担，文件描述符仅起命名的作用。

4.4BSD 通过共享内存区中的信号量可以实现进程间的高速同步操作。在传统的 UNIX 中，进程使用信号量同步其对共享内存的访问(见 6.3 节)。处理信号量需要系统调用，而系统调用带来了额外的高代价，抵销了共享内存获得的性能受益。4.4BSD 将信号量存放在共享区，减少了这些代价。

实验研究表明，在大多数使用同步的应用程序中，当一个进程获取一个共享资源时，大多数情况下共享资源都处于解锁状态。如果协同的进程将信号量存放在共事内存区中，通过系统提供的原子性的 test-and-set 或其等价指令,进程可以不调用系统调用而同样获取共享资源。仅在资源被锁住时，进程才需要调用系统调用等待资源解锁。然后内核再次检查信号量，并在其仍为加锁状态时阻塞进程。与此相似，释放资源的进程也可以在用户级释放信号。之后，它检查是否有其他进程等待资源，如果有就调用系统调用唤配它们。

4.4BSD 提供如下信号量管理接口。包含信号量的共享内存区必须用 MAP_HASSEMAPHORE 标志创建。一个进程要获取信号量时，调用：

value = mset(sem, wait);

其中，sem 是指向信号量的指针，wait 是一个布尔值。为真时，调用者可以阻塞在信号量上。返回时，如果进程获取了信号量，value 为 0。释放信号量，调用：

mclear(sem);

系统使用原子性的 test-and-set 指令在用户级实现 mset 和 mclear 函数。要在信号量上阻塞和去阻塞，4.4BSD 有如下调用：

msleep(sem);

检查信号量，如果仍处于加锁状态，阻塞调用者。调用

mwakeup(sem);

唤醒至少一个阻塞在信号量上的进程，如果没有阻塞进程，什么也不做。

System V IPC 设施(见 6.3 节)中的信号量接口与此皆然不同。System V 可以通过一个系统调用处理一组信号量。内核保证一个调用中的所有操作都是原子性。这种接口也可以在 4.4BSD 中实现，只需为每组信号量设置一个监护信号量。对于任何该信号量集上的操作，进程首先要获得监护信号量，然后察看它是否能完成所需的操作。如果不能，就释放监护信号量，并在不能获取的某个信号量上阻塞。当其被唤醒时，它必须重复整个过程。

15.9 快表(TLB)一致性

如果处理器在每次地址转换都要访问内存中的页表，系统就会产生大量的内存访问。因此大多数 MMU 部有一个快表(TLB)，缓存当前使用的地址转换。TLB 由硬件实现，其中表项较少(一般情况下从 64～256)，每一项映射一个物理页面。MMU 在每次地址转换操作中检查这个缓存。如果在缓存中找到了相应的地址转换，就可以免去查找内存中地址转换映射图这种耗时的访问。

TLB 是一个关联缓存，即 MMU 同时在缓存的所有表项中查找，以找到与给定的虚地址相匹配的项。表项中包含了相应的物理页面号，以及页面的保护权限，表项的具体格式由不同的机器决定。13.3 节中给出了 MIPS R3000，IBM RS6000 以及 Intel 80386 体系结构中 TLB

的格式及访问语义。

TLB 支持两个基本操作——加载和报废或清除。在每次发生缓存不命中时加载 TLB 表项。在大多数体系结构中，这个操作是由硬件执行的，硬件了解二级地址转换映射图(比如许多系统中的页表，或者是 IBM RS6000 这类系统中的反向页面)的具体位置和格式。MMU 硬件定位页表项，在完成地址转换前加载 TLB 表项。内核既不参与也无法意识到这些操作。

某些体系结构，如 MIPS R3000[Kane 88]，采用软件加载。在这种系统中，MMU 仅知道 TLB，并在每次 TLB 不命中时产生一个失效。内核管理地址转换映射图，硬件既不知道它们的存在也不定义它们的结构。一般情况下，内核使用传统的页表，但并不一定要这样。当 MMU 产生一个 TLB 不命中失效时，内核定位地址转换信息，显式地加载一个含有正确信息的 TLB 表项。

所有的 MMU 都以某方式允许软件清除 TLB 表项。内核会国各种各样的原因而修改页表项，比如需要修改页表项的保护权限信息，或者在页面换出时需要清除该页表项。这样就会造成相应的 TLB 表项不一致，这些表项必须从缓存申报废或清除。内核可以使用 3 种方式清除一个 TLB 表项：

· 清除一个对应某虚地址的 TLB 表项。如果 TLB 中没有对应该虚地址的表项，什么也不做。VAX-11 上的 TBIS(Translation Buffer InvalidationSingle)指令就是这种指令。

· 清除整个 TLB 缓存。例如，在每次页目录基址寄存器(PDBR)被写入时，系统都清除整个 TLB。对 PDBR 的写入既可以是显式的 move 指令，也可以是在发生上下文切换时隐式地进行。VAX-11 上的 TBIA(Translation Buffer Invalidation All)指令也有同样的效果。

· 加载一个新 TLB 表项，覆盖相应虚地址的原 TLB 表项。MIPS R3000 使用这种方法，允许软件加载 TLB 表项。

15.9.1 单处理机上的 TLB 一致性

在单处理机系统中，TLB 管理是非常容易的。在如下事件中需要清除一个或多个 TLB 表项：

· 保护权限变化 用户可以调用 mprotect 升高或降低某地址范围的保护权限。内核或者为了模拟引用位，或为了实现 copy-on-write，也可能改变页面的保护权限。

· 换出 当从物理内存中删除页面时，内核必须清除所有映射到该页面的页表项和 TLB 表项。

· 上下文切换 当内核切换到一个新进程时，旧进程的所有 TLB 都变为无效。由于所有的进程都共享内核，映射内核地址的表项仍然有效。某些体系结构支持带标记的 TLB，每一个 TLB 表项都有一个标记标识其属于哪个进程。对于这样的系统，新进程将使用一个不同的标记，上下文切换时无需清除 TLB 表项。

·exec 当进程调用 exec 执行另一个程序时，所有映射其旧地址空间的映射都变为无效，现在相同的虚地址将引用新映像页面。

清空 TLB 很耗时，内核要采用各种措施尽量将这种代价减到最小。一种很重要的思路就是这种不一致是否是良性的。比如，假设内核降低了页面的保护权限，将其从只读改为可读可写。这时 TLB 表项与页表项不一致，但不必清除 TLB 表项。在最坏的情况下，用户对页面进行写操作，发现 TLB 表项仍表明页面是只读的。此时，页面失效函数(或硬件)可以加载一个新的 TLB 表项。直到必要时才加载 TLB，利用这种方法，我们有时会完全避免加载 TLB(例如，TLB 表项在上下文切换或其他事件被请除)。

在涉及多个页面时，内核必须选择是清除整个 TLB 缓存，还是逐一地清除许多表项。前者由于整个缓存只用一条指令清除，从短期来看比较快。但它可能带来更多的 TLB 失效，从长远看，它的代价更高。优化的方案取决于要清空的表项数目。

15.9.2 多处理机问题

在共享内存多理机系统中，维护 TLB 的一致性是一个更加复杂的问题，尽管所有处理机都共享内存，但每一个都有自己的 TLB。当一个处理器更改一个页表项，而另一个处理器仍使用该表项时，问题就产生了，后者可能在其 TLB 中保存了那个页表项的副本，它会继续使用那个过时的映射。因此，将 TLB 表项的变化传播到所有使用该页面的处理器上是非常必要的。

只有少数几个系统在硬件上支持同步 TLB。例如，IBM System/370 有一条 ipte 指令，它自动修改页表项，并将在系统所有的 TLB 中清除该表项。一般情况下，硬件并不支持 TLB 问的同步，大部分系统甚至不允许一个处理机清除另一个处理机的 TLB 表项。

在许多情况下，一个页面的变化会影响若干个不同的处理机：

·页面是内核页面。

·页面由多个进程共享，每个进程运行于不同的处理机。

·在多线程系统中，同一进程的不同线程可以并发地在不同处理机上运行。如果一个线程修改了映射，所有线程都应看到相应的变化。

还有一种相关的情况，在某个处理机上运行的进程更改了运行于另一个处理机上的进程的地址空间。因此.即使仅修改一个 TLB 表项，除了需要更改最初更改 TLB 的处理机的 TLB 之外，系统还需要更改其他处理机的 TLB。

由于缺少硬件支持，内核必须利用基于跨处理机中断的通知机制，从软件上解决这个问题。为了以下论述清晰，我定义术语“启动者”和“应答者”。启动者是修改映射的处理机，之后其要清除几个远程 TLB。应答者是持有相应映射 TLB 的远程处理机。启动者向应答者发送一个中断，应答者接收中断后清除相应的 TLB。

由于我们需要同时完成两件事——修改页表项和清空 TLB 表项，处理起来要复杂些。假定应答者在启动者修改 PTE 之前清除相应的 TLB 表项，运行在应答者上的进程可能在其间访问页面，造成硬件重新加载无效的 TLB 表项。将顺序倒过来也会有问题，如果启动者首先更改了 PTE，应答者可能为了更新引用位和修改位而将过时的 TLB 表项回写页表。

为了保证这些操作的一致性，应答者必须在其清空页表项之前等待(忙循环)启动者更新 PTE。下面我们讨论 Mach 的 TLB 同步算法。

15.10 Mach 的 TLB 击落算法

Mach 的 TLB 击落(shootdown)算法[Blac 89]涉及启动者与应答者之间一系列复杂的交互。击落是指清除另一个处理机上的 TLB。为了实现该算法，Mach 在每个处理机上使用了一组数据结构：

·一个活动标志，说明处理机是否正在以正常方式使用页表。如果该标志被清除，处理机正在参加击落过程，不会访问任何可能修改的 pmap 表项(pmap 是任务的硬件地址转换映射图，通常包括页表)。

·清除请求队列。每个请求对应一个必须从 TLB 中清除的映射。

·一组当前活动 pmap。每个处理通常用两个活动 pmap——内核 pmap 和当前任务的 pmap。

每个 pmap 都用一把旋锁保护起来,将 pmap 上的操作串行化。每个 pmap 有一个处理机链，记录正在使用该 pmap 的处理机。

当一个处理机修改了地址转换并需清除其他处理机上的 TLB 表项时，内核调用击落算法。图 15-13 所示为一个应答者的情景。启动者首先关闭所有中断，清除自己的活动标志。接着，它锁住 pmap 并在 pmap 链上的处理机的请求队列中记录 TLB 刷新请求。然后它向这些处理机发送处理机间中断，等待它们的应答。

当应答者接收到中断后，它也关闭所有中断。然后通过清除活动标志对中断应答，并且等待启动者对 pmap 解锁，与此同时，启动者一直在等待所有有关处理机进入失活状态。当所

有的中断均应答后，启动者刷新自己的 TLB，修改 pmap，最后对其解锁。应答者现在退出等待循环，处理它们的请求队列，刷新所有过时的 TLB 表项。最后，启动者和应答者都重置它们的活动标志，打开中断，恢复正常操作。

图 15-13　Mach 的 TLB 击落算法

15.10.1　同步和死锁避免

击落算法使用了几种同步机制，操作间的顺序是很重要的[Rose 89]。关闭所有中断很重要，否则一个设备中断可能使许多处理机在很长一段时间内处于空闲状态。pmap 上的旋锁防止两个处理机同时对同一个 pmap 启动击落过程。必须在锁住页面之前关闭中断，否则处理机可能在其加锁成功之后，因接受到处理机间中断(另一个活动 pmap)而造成死锁。

为了避免某些死锁的可能性，启动者在对 pmap 加锁前清除自己的活动标志。假定两个处理机，P1 和 P2，修改同一个 pmap。P1 关闭中断，对 pmap 加锁，并向 P2 发中断。同时，P2 关闭中断，阻塞在同一把锁上。现在死锁发生了，P1 正在等待 P2 应答中断，而 P2 正在等待门对 pmap 解锁。

清除活动标志可以在中断到来之前应答这些中断。在上面的例子中，由于 P2 在对 pmap 加锁之前清除了它的活动标志，P1 不会阻塞。当 P1 对 pmap 解锁时，P2 将恢复执行并处理由 P1 发送的请求。

击落算法对所有的加锁操作有些微妙的影响。它要求对是先关闭中断还是先获取锁达成一个一致的策略。假定处理机 P1 允许中断同时持有资源，P2 不允许中断并正在获取资源，而 P3 启动了一个击落，应答者是 P1 和 P2。P3 向 P1 和 P2 发送处理机间中断，等待它们的应答。P1 响应中断，并且在 pmap 释放前一直阻塞。P2 阻塞在锁上，且关闭了所有中断，因此它看不见或不能响应这个中断。结果出现了 3 方死锁，为了防止这种死锁，系统必须为某把锁强制一个固定的中断状态；在获取锁时，或者不允许中断，或者允许中断。

15.10.2　讨论

Mach 的 TLB 击落算法解决了一个复杂的问题，而且除了处理机间中断外，不依赖任何硬件的特殊功能。但它消耗很大，而且可扩展性不好。所有的应答者都要在启动者修改 pmap 时处于忙等状态。在有数十或数百个 CPU 的大规模多处理机系统中，击落会使许多处理机同时进入空闲状态。

这种复杂性是必需的，这主要有两个原因，第一，许多 MMU 自动将 TLB 表项回写页表，修改引用位和修改位。这种更新覆盖整个 pmap 表项。第二，硬件页面大小通常不同于软件页面大小。因此，当内核修改某个页面的映射时，它可能要修改若干 pmap 表项。对所有处理机而言这些更新都须是原子性的,唯一的解决方法就是在修改期间让所有正在使用 pmap 的处理机空闲。

人们给出了许多其他的 TLB 击落算法的建议和实现。下一节将介绍一个减少 TLB 刷新频度的 ad hoc 方案。其他方法都依赖某些硬件特性简化问题。例如，[Rose 89]针对 IBMResearch Parallel Processor Prototype(RP3)[Pins 85]提出了一个有效的算法，它利用了 RP3 的一些特性，RPS 不自动回写 TLB 表项，而且有很大的硬件页面尺寸(16KB)，一个软件页面不必跨越多个硬件页面。其他一些研究[Tell 88]还建议对 MMU 体系结构进行修改，辅助完成 TLB 的击落过程。

15.11　SVR4 和 SVR4.2 UNIX 中的 TLB 一致性

由一系列计算机公司组成的 Intel 多处理机财团开发了 SVR4/MP，SVR4 的 Intel 系列多处理机版本。随后，UNIX 系统实验室发行了 SVR4.2/MP，SVR4.2 的多处理机版，同时支持轻量级进程。本节，我们介绍这两个版本的 TLB 一致性算法。

15.11.1　SVR4/MP

Mach 的击落算法尽可能高效地解决通用 TLB 同步问题，不依赖硬件的特殊功能(除了处理机间中断)，也不依赖那些击落事件的天然特性。SVR4/MP 的方法则是分析造成击落的事件，找出解决它们的最好的方法。有四种需要进行 TLB 同步的事件类型：

1.进程调用 sbrk，brk 或者释放一片内存区缩减自己的地址空间。

2.释放页面或模拟引用位时 pagedaemon 清除页面。

3.内核将系统虚地址重新映射到另一个物理页面。

4.进程写一个共享的 copy-on-write 页面。

SVR4/MP 中，硬件在每次上下文切换时自动刷新 TLB[Peac 92]。囚此，如果操作系统不支持多线程进程(SVR4/MP 的这个发行尚不支持)，第 1 种类型不是问题。SVR4/MP 对第 2，3 种类型进行优化处理。

为了减少第 2 种情况中的 TLB 刷新，pagedaemon 将许多清除操作进行批处理，在一步操作中更新所有 TLB 表项。它用全局 TLB 刷新的代价抵偿一系列页面清除操作的代价。

SVR4 中产生 TLB 同步操作最主要的原因是支持 read,write 系统调用的 seg_map 驱动器。内核通过 seg_map 实现 read 和 write 系统调用，将文件映射到内核地址空间，再将它们复制到用户空间。它管理 seg_map 段，动态地将文件映射到内核空间，或者对它们进行占映射。结果，该段虚地址的物理映射频繁变化。由于所有的进程都共享内核，必须防止其他处理机通过过时的 TLB 映射访问这些地址。

为了跟踪过时 TLB，SVR4/MP 内核维护一个全局计数，同时每个处理机也维护一个本地计数。每次处理机刷新它自己的 TLB，就将全局计数加 1，并将新的全局计数值复制到本地计数中。当 seg_map 释放页面的映射时，内核用全局计数标记该地址。当 seg_map 为某个新物理页面重新分配这个地址时，内核比较它保存的计数和处理机的本地计数。如果本地计数小于所保存的计数值，TLB 可能有过时的表项，于是内核执行一个全局 TLB 刷新。否则，所有处理机在该地址清除后都有一个已刷新的 TLB，没有该页面的过时 TLB。

SVR4/MP 对这种情况所进行的优化处理，是基于这样的假定，一旦某个 seg_map 映射被清除后，内核在其重分配该地址前是不会访问这个地址的。为了尽量减少刷新，seg_map 释放的页面按先入先出方式复用。这增加了一个地址从释放到再利用之间的时间，使得处理机同时更新一个 TLB 表项的机会减少了。

15.11.2 SVR4.2/MP

SVR4.2/MP 是 SVR4.2 的多处理机，多线程版本。它的 TLB 击落策略与实现[Bala92]与 15.11.1 节中介绍的 SVR4/MP 有某些共同的特征，但进行了若干重要的改进。所有同 TLB 的交互都局限在与机器相关的 HAT 层(见 14.4 节)。SVR4,2/MP 的参考移植是针对 Intel 386/486 体系的，但 TLB 一致性算法和接口都设计得易于移植。

与 SVR4/MP 相同，内核完全控制自己的地址空间，可以保证自己不会访问无效内核映射(比如，那些由 seg-map 段释放的映射)。因此，内核可以针对映射内核地址的 TLB 表项采用一种 lazy 击落策略。然而，SVR4.2/MP 支持轻量级进程(lwps，3.2.2 节中有述)，同一进程的多个 lwp 可能同时运行在不同的处理机上。由于内核无法控制用户进程的内存访问模式，必须对无效用户 TLB 表项采用立即击落策略。

对于有很多处理机的系统，SVR4/MP 的全局击落算法扩展性不佳，在击落期间整个系统都处于空闲状态，SVR4.2/MP 在每个 hat 结构中维护了一个处理机链表。由于内核可能运行于所有的处理机上，其地址空间 hat 结构上的链表可能有所有的在线处理机。每个用户进程 hat 的链表上有所有运行该进程的处理机(运行进程的一个 lwp)。通过如下接口访问这个链表：

- hat_online()和 hat_offline()在内核 hat 结构中增加或删除 CPU。
- hat_asload(as)将处理机加入 a5 结构对应的 hat 结构上的链表，并且向 MMU 中加载地

址空间。

·hat_asunload(as，flags)卸载进程的 MMU 映射，从 as 对应的链表中删陈该处理机。

参数 flags 支持一个标志，指示在卸载映射之后是否必须更新本地 TLB⑥。

内核使用一个仅由 HAT 层可见的对象，称为 cookie。可以将其实现为时间戳或一个计数(在参考移植中是时间戳)，而且必须满足新 cookie 的值总大于以前的 cookie 这个条件。例程 hat_getshootcookie()返回一个新 cookie，其值是对 TLB 年龄的一种量度。内核将 cookie 作为参数传递给 hat_shootdown()例程，它负责完成内核 TLB 的击落。如果其他的 CPU 均有一个较早的 cookie，它的 TLB 可能过时了，需要更新。

下一小节阐述内核如何实现 lazy 和立即击落算法。

15.11.3 Lazy 击落算法

与 SVR4/MP 相同，内核也针对 seg_map 段使用 lazy 击落算法，同时针对用于动态分配内核内存的 seg-kmem 驱动程序使用 lazy 策略。当这些驱动程序清除一个页面时，内核延缓击落过程，盲到页面再次由内核使用时才进行击落。例如 seg_map 驱动程序在页面的引用计数变为 0 时清除页面，此时，不再有引用该页面的进程了。因此，将这些过时的无效的页面地址转换信息保留在 TLB 中是相对安全的。内核将页面返回给空闲链表，然后调用 hat_getshootcookie()为页面设置一个新的 cookie⑦。

当内核准备复用这个页面时，它将页面上的 cookie 传递给 hat_shootdown()，hat_shootdown()将其与所有其他 CPU 的 cookie 进行比较。所有持有较早的 cookie 的 CPU 都有一个该页面的过时 TLB，需要对其进行击落。

hat_shootdown()例程运行于启动者环境中，它首先获取一个全局旋锁，保证一次只进行一个击落。然后通过处理机间中断向所有有旧 cookie 的处理机发送中断，一旦每个响应处理机已开始处理请求，启动者就释放旋锁并完成对页面的再分配。

应答者在收到中断后，不必等待任何同步事件。它们简单地刷新自己的 TLB，恢复正常处理。由于启动者在击落之前修改了地址转换信息，这之后由应答开发出的页面访问将加载一个有效的 TLB 表项。处理机间中断的中断优先级最高。此间不会处理其他中断，否则如果某个处理函数引发了另一个击落，系统就会死锁。

该算法中涉及的同步相比 Mach 中的击落算法要简单和高效得多。这是由于 SVR4.2/MP 内核可以保证不访问无效页面，而在 Mach 里没有做这样的假设。

15.11.4 立即击落

当内核清除一个用户 PTE 时，它必须立即击落所有运行该进程的处理机。这是因为内核无法控制用户的内存访问模式，也就不能保证用户不会访问无效页面。由于应答者必须等到启动者修改了 PTE 之后方可从中断中返回，算法也就更复杂一些。

SVR4.2/MP 的用户 TLB 同步算法与 Mach 的很相似。内核必须使用一个额外的同步计数器作为由启动者和应答者之间共享的信号量。图 15-14 给出了操作的顺序。

图 15-14 SVR4.2/MP 中的用户 TLB 同步

当内核必须清除一个用户 PTE 时，它首先对 hat 结构加锁，获取全局旋锁(防止与其他击落冲突)。接下来，它通过处理机间中断向共事进程地址空间的处理机发送中断(利用 hat 上的链表)。当应答者收到中断后，它进入循环，等待同步计数被加 1。

当所有的应答者均开始循环后，启动者修改页表项，然后对同步计数器加 1。应答者退出循环，刷新它们的 TLB，从中断中返回。最后，启动者刷新本地 TLB，对全局旋锁和 hat 结构解锁。

该算法在启动者须在同一地址空间中修改多个 PTE 时，可以针对 Intel 体系做进一步的优化，比如在去映射一个段时等等。386 有一个二级页表{见 13.3.2 节)，第 1 级页表包含第二级页表的 PTE。在多个 PTE 的情况中，启动者只须在对同步计数加 1 前简单地清除相应的

第 1 级页表。

这样，应答者就不必等待启动者更改所有的 PTE，而直接完成自己那部分的击落，然后恢复正常处理。如果应答者上的 lwp 访问无效页面时，由于其 1 级 PTE 是无效的，系统产生一个页面失效。由于 hat 结构已被启动者锁住，失效处理函数将被阻塞住。这样，应答者在启动者完成工作和加载映射之前，就不会访问无效地址转换信息。尽管这种优化方法很完美，但它是过于依赖 Intel 体系，只适用于某些情况。

另一个需要立即击落的例子是换出操作。在 SVR4.2/MP 以前，pagedaemon 采用一种全局替换策略。它扫描全局页面池的许多页面，清除引用位(收集引用信息)或修改位(在清除页面之后)。这两个操作都需要进行全局击落。SVR4.2/MP 替代厂这个算法，它使用一种基于本地工作集计龄方法的算法。在完成计龄时换下所有要计龄的进程(除了在启动者上运行的 lwp 之外的所有 lwp 部从相应处理机上切换下来)。在切换回 lwp 时，386 的上下文切换自动刷新 TLB。

15.11.5 讨论

SVR4/MP 和 SVR4.2/MP 都利用某些硬件特有的特征优化 TLB 击落算法。此外，它们还针对不同的击落情况做不同的处理，利用这些触发击落事件中的固有的同步性优化算法。

这种方法比 Mach 那种适用于任何情况，任何机器的大一统方法要有较好的性能。然而由于 SVR4 的方法依赖于硬件和软件的许多特性，它们移植起来比较困难。例如，MIPS R3000 支持带标记的 TLB，上下文切换时不自动刷新 TLB，这带来一系列完全不同的问题，15.12 节中针对这一结构给出了解决方案。

简而言之，我们再一次面对的是，是采用通用的统一方案，还是采用若干 ad hoc 方法解决各种不同情况的权衡问题。

15.12 其他 TLB 一致性算法

MIPS R3000 上多理机版 SVR3[Thom 88]也提出了一个软件解决 TLB 一致性问题的方案。MIPS 结构[Kane 88]使用带标记的 TLB(见 13.3.4 节)，每个 TLB 表项有 6 位标记，TLBpid，它用于标识该地址转换属于哪个地址空间。这给系统带来了重大的影响。由于新进程有不同的 TLBpid，无需在上下文切换时刷新 TLB。因此，一个进程可能在其运行过的处理机上遗留 TLB 表项。如果进程再次在这些处理机上运行，除非这些表项被刷新或被替换掉，否则进程可以再次使用它们。

这种系统必须正确地处理地址空间缩减。假设一个进程开始运行于处理机 A，然后又运行于处理机 B。在运行于 B 时，进程缩减了自己数据区，并且刷新了 B 上的无效 TLB 表项。如果进程稍后又运行在 A 上，它就有可能通过其遗留在 A 上的旧表项访问无效页面。

为了解决这个问题，内核为每个缩减了地址空间的进程赋予一个新的 TLBpid。这样就会自动清除处理机上所有该进程的 TLB 表项⑧。当内核重新将一个旧 TLBpid 分配给另一个进程时，内核要执行一次全局 TLB 刷新。内核按先入先出的顺序重新分配 TLBpid，这可以大大减少全局 TLB 刷新次数，旧 TLB 表项随着系统运行会被自动清除。

在 MIPS 的实现中，系统还处理进程写 copy-on-write 页面的情况。内核生成页面的一个新副本，并将其分配给执行写操作的进程。然后刷新本地 TLB。如果进程先前在其他处理机运行过，那些处理机可能有该页过时地址转换信息。内核为每个进程维护一个它运行过的处理机链表。如果进程在写 copy-on-write 页面之后，又运行于这些处理机上，内核首先要刷新那个处理机的 TLB。

本节中给出这些优化方法减弱了全局 TLB 同步的需要，提高了系统性能。特别是由于所有进程都共享内核映射，seg_map 经常使用，对 seg_map 的优化非常有用。然而，这些方案都是 ad hoc 的，依赖于引发同步的硬件和操作系统的特定性质，没有一个统一而通用的系统

(除了 Mach)是硬件无关且适用于任何情况的。

15.13 虚地址缓存

如同 TLB 是地址转换的缓存一样，计算机还为物理内存设置了高速缓存。大部分机器有分离的数据、指令缓存或一个公用的缓存。这些缓存的大小通常为 64～512KB，而且要比内存访问要快得多。缓存一般采用写回方式，就是说数据写操作仅修改缓存。仅在缓存需要为新数据分配或因为其他原因而替换缓存数据行时，才向主存刷新数据。

传统的硬件体系结构使用物理地址缓存(图 15-15)。MMU 首先转换虚地址，然后再访问物理内存。所有对物理内存的访问都要通过缓存。如果在缓存中找到了数据，MMU 就无需访问内存了。

图 15-15 典型的物理地址缓存体系结构

这种缓存的优点是简单。硬件保证厂缓有的一致性，操作系统既不知道缓存的存在，也不负责其一致性维护。缺点是只能在地址转换之后才进行缓存查找，减弱了缓存的优点。此外，如果 TLB 中没有有效的地址转换，MMU 必须从物理内存中或缓存访问页表项。这需要额外的缓存和内存访问。

许多现代体系结构都使用虚地址缓存，在某些结构中一并取消了 TLB。图 15-16 所示为一种典型的结构。MMU 首先在缓存中查找相应的虚地址。如果发现了，就不必进一步查找了。如果数据不在缓存中，MMU 再进行地址转换，从物理内存中获取数据，

该结构中还可以同时有虚地址缓存和 TLB。在这种结构中，比如 MIPS R4000[MIPS 90]和惠普 PA-RISC[Lee 89]，MMU 同时查找缓存和 TLB，这种结构通过一定的结构复杂性代价换取了更高的性能。

虚地址缓存由若干缓存数据行组成,每一行都映射若干连续的内存字节。例如,Sun-3[Sun 86]的缓存大小为 64kb，每行 16 字节。缓存用虚地址进行索引，有时还可以将进程 ID 或上下文 ID 与虚地址组合起来作为索引。由于许多虚地址(从同一或多个地址空间)映射到同一个缓存数据行，每个行上必须包含一个标识其映射的进程和地址空间的标记。

图 15-16 虚地址缓存

使用虚地址作为索引有一个重要的影响——每个缓存都有一个对齐因子。两不同的虚地址可能映射到同一个缓存数据行，对齐因子通常等于一个或多个缓存大小。我们用“对齐地址”代表映射到同一缓存数据行的地址。

尽管物理地址缓存对操作系统是完全透明的，硬件却不能保证虚地址缓存的一致性。一个给定的物理地址可能映射到若干个虚地址上，因而也可能映射到多个缓存数据行上，这引起了内部的一致性问题。缓存的写回特性造成内存的数据相对于缓存过时。一般有 3 种类型的一致性问题——映射变化或多义地址，地址别名或同义地址和直接内存访问(DMA)操作。

15.13.1 映射变化

在虚地址被映射到不同的物理地址时需要修改映射(图 15-17)。如下几种情况会修改映射：

图 15-17 映射变化导致 cache 项失效

·上下文切换 上下文切换用一个新进程的地址空间替换原地址空间。在大多数结构中，缓存数据行上的标记标识该行属于哪个进程。这样，上下文切换就不会清除整个缓存。然而，在许多系统中，u 区在内核地址空间中。上下文切换时，内核将 u 区的地址重新映射到新进程的 u 区上。由于内核是共享的，其缓存行有一个特殊的标识，对所有的进程始终都是有效的。因此，上下文切换要清除缓存中原 u 区的所有数据行。

·换出 当 pagedaemon 从内存中删除页面时，需清除页面的所有缓存数据行。

·保护权限变化 页面的保护权限发生变化时，缓存数据行也受到影响。在响应显式的

mprotect()调用，处理 copy-on-write 页面或者模拟引用位时，内核都要修改保护权限。当保护权限变宽松时(例如，只读页面变为可读可写)，这种变化是良性的，无 需更新缓存。当进程访问数据时，将产生页面失效，再由失效处理函数加载正确的表项。当保护权限变严格时(例如，一个可读可写页面变为只读)，该变化就必须传播到所有缓存该数据的缓存数据行。

· copy-on-write 当进程写一个以 copy-on-write 方式共享的页面时，内核创建一个新的页面副本，标记其为可写，修改相应映射引用该副本。此时，要清除引用原页面的所有缓存数据行。

15.13.2 地址别名

地址别名，或同义地址，是指多个虚地址映射同一物理地址(图 15-18)。如果进程用其中一个地址修改了内存单元，该变化不会自动地传播到其他缓存数据行中。如果另一个进程用其他地址读取这一数据，它将访问到过时数据。此外，如果两个进程用不同的地址写同一个内存单元，它们被刷新到内存的顺序是不确定的。同义地址可以因如下原因而引起：

图 15-18 使用别名导致不同的 cache 项引用同一数据

· 共享内存 当多个进程共享内存区时，每个进程都将其映射到自己的地址空间。如果这些地址不是对齐的，它们映射到不同的缓存数据行上，产生同义地址。由于进程可以将共享内存区映射到地址空间的任意一处，内核不能保证这些地址始终是对齐的。

· mmap 进程可以用 mmap 将文件或内存对象映射到地址空间的任意一处。如果多个进程将同一对象映射到不对齐的地址上，就产生同义地址。进程也可以将同一内存区映射到其地址空间的不同地址上，这一样会产生相同的问题。

· DVMA 在支持直接虚存访问(DVMA)的系统中，设备可以在已有虚地址的页面上建立新的虚地址映射。这要求在建立新映射和完成 DVMA 操作后各刷新一次缓存。

15.13.3 DMA 操作

许多设备都可以不经过 CPU 直接与内存进行数据传输。这种功能称为直接内存访问或 DMA(与 15.13.2 节所述的 DVMA 不一样)。DMA 传输通常不经过缓存，而直接在内存上进行。尽管这样做有速度上的优点，但会产生缓存一致性问题。假定缓存中尚有一些还未刷新到主存中的已修改数据，DMA 读将读不到这些数据，结果它读到了过时的数据。与此相似，DMA 写也不会覆盖缓存，从而使缓存中的数据过时。稍后，过时的缓存数据可能被(不正确地)刷新到内存中，破坏了新数据。

15.13.4 维护缓存一致性

在有虚地址缓存的系统中，内核必须清楚地了解每一种可能造成缓存不一致的事件，并在其发生时采取某些必要的措施。通常，缓存都向内核开放两个接口——刷新和清除——两者都从缓存中删除或清除一个缓存数据行。刷新操作还同时向上存回写已有的修改；清除操作则不进行这样的操作。内核依情况采取不同的措施。让我们来看看具体的例子。

当修改页面的映射后，内核必须刷新页面的所有缓存数据行。缓存刷新代价很高，必须尽可能避免。因此内核只刷新那些由有效变为无效的映射。例如，在上下文切换时，当旧进程 u 区的映射被清除后，内核刷新 u 区所对应的缓存数据行。当映射新 u 区时，由于这是一个由无效到有效的变化，无需刷新。

pagedaemon 清除要释放的页面。此时，还要刷新相应的缓存数据行，否则，某个进程可能继续访问在缓存中的那部分页面。而当页面装入新数据时，无需刷新。

在 fork 期间，内核可能将子进程 u 区临时映射到一片称为 forkutl 的内核地址空间上。使用 forkutl 的例程在返回时没有显式地释放该映射。然而，对于虚地址缓存系统来说，由于内存区已被隐式地去映射，该例程必须刷新相关的缓存数据行。

最容易的处理地址别名的方法就是从硬件层上防止它的发生[Chao 90]。当进程在某页面上发生失效时，内核检查是否已有其他映射映射该页面。如果是这样，内核清除那个映射，

在缓存中刷新相应的页面，然后为产生失效的进程创建新映射。这样，共享内存在同一时刻仅有一个映射，需要不断地对它重新映射和刷新。

然而，由于刷新缓存十分耗时，这种方案代价很高。此外，任务在共享内存时还会不断发生失效，处理失效的代价也很高。我们需要考虑另一种较为高效的方案。一种立竿见影的改进方案就是只允许只读访问对别名地址。当某进程写这样一个页面时，内核清除页面的所有映射。

仅当两个地址不对齐时，别名地址才产生不一致现象。尽管 UNIX 允许应用程序将共享内存区或其他内存对象映射到用户指定的任意地址上，大部分应用并不依赖这一特性。像 shmat 和 mmap 这类系统调用都可以指定由内核选择一个合适的映射地址。内核尽可能选择对齐的地址，从而消除了缓存一致性问题。例如，在 Sun-3 上，如果两个地址相差恰为 128kb 的倍数，它们映射到同一个缓存数据行上。在这种系统中，内核选择相同或相差 128kb 倍数的虚地址映射共享内存区。

许多 UNIX 实现甚至更保守，它们彻底取消了这些设施，或限制它们的使用。例如，许多惠普的 HP-UX 操作系统[Cleg 86]版本不支持 mmap，仅允许共享映射到同一地址的内存区。这些系统不是通过内存映射而是利用全局共享虚地址段实现正文共享。SunOS[Chen87]对那些有可能有多个非对齐虚地址的页面不进行缓存，藉此来解决这一问题。

DMA 操作也必须针对虚地址缓存做不同的处理。在启动 DMA 读之前，内核必须从缓存中刷新该页面的所有已修改数据。这可以保证内存的数据与缓存一致。与此相似，住 DMA 写时，内核首先要从缓存中清除被覆盖的缓存数据行。否则，那些过时的缓存行稍后可能被回写到内存中，破坏了 DMA 刚刚写入的数据。

15.13.5 分析

虚地址缓存可以大大改善内存访问时间。但它带来许多必须由软件解决的一致性问题，此外，它从根本上改变了内存结构，需要对操作系统设计中的若干问题重新考虑。尽管设计虚地址缓存是为了提高 MMU 的性能，但它与操作系统基于的某些假设相冲突，相反，这会影响整个系统的性能。

现代 UNIX 系统支持多种内存共享和内存映射的方式，如 System V 的共享内存，内存映射文件访问，以及内存继承和进程间通信的 copy-on-write 技术等等。在传统的体系结构中，这些技术减少了内存复制，减少了同一数据在内存中的副本数，降低了内存消耗。这些都带来极大的性能改善。

而在虚地址缓存结构中，这种内存共享造成许多同义地址。操作系统需要精心设计以保证缓存的一致性，往往采用缓存刷新，标记某些页面只读，去除或限制某些设施的使用等措施，这样一来就消弱了内存共享所获得的性能改进。[Chen 87]表明，在一些典型的测试用例中，用于缓存刷新的仅占总时间的 0.13%，而某些测试用例的这个值涨到 3%。

在许多情况下，有些算法必须重新设计，这样它们才能高效地运行在有虚地址缓存的系统上。为解决这些问题，[Inou 92]中针对 Mach 和 Chorus 系统的操作进行了修改。[Whee 92]介绍了许多去除不必要的缓存一致性操作的方法，从而极大地提高性能。但其中的某些建议是针对惠普的 PA-RISC 结构的特性提出的。在 PA-RISC 结构中，TLB 查找与缓存的地址匹配并行完成，缓存用一个物理地址标记。这可以使系统发现许多软件中的不一致问题，及时采取更有效的措施。

15.14 练　　习

1.Mach 中的内存继承与 SVR4 的有哪些不同?

2.在图 15-2 所示的例子中，如果任务 A 写页面会怎么样?

3.为什么 Mach 的外部 pager 接口会有较差的性能?

4.VM 对象与内存对象有哪些不同?

5.如果客户机崩溃了，网络共享内存服务器将如何工作?如果服务器崩溃了又会怎样？

6.为什么 Mach 不需要 SVR4 中的那种页面保护数组?

7.假设某厂商希望在基于 Mach 模型的系统中提供 System V IPC，他(她)将如何实现共享内存语义?需要解决哪些问题?

8.Mach 的 vm_map 调用，4.4BSD 的 mmap 以及 SVR4 的 mmap 之间有哪些相似和相异之处?

9.15.8 节中提到在 4.4BSD 中用一个监护信号量实现类似于 SystemV 的信号量集，监护信号量是由谁分配和管理的?内核还是用户库?简要陈述一下实现。

10.为什么 Mach 的页替换策略称为多选 FIFO?

11.由操作系统加载 TLB 与硬件加载相比有哪些好处?

12.假设 TLB 表项中有一个硬件支持的引用位，内核将如何使用这一位?

13.由于 UNIX 内核不参加换页，什么操作会造成内核页面的 TLB 表项被修改?

14.为什么 TLB 击落代价很高?为什么 Mach 中的击落要比 SVR4/MP 或 SVR4.2/MP 的代价更高?

15.你认为在写一个 copy-on-write 页面时,SVR4/MP 和 SVR4.2/MP 将如何处理 TLB 清除?

16. 引入轻量级进程后，又带来了哪些 TLB 一致性问题?

17.为什么在许多场合下，lazy 击落比立即击落要好?什么时候立即击落是必须的?

18.有了虚地址缓存的 MMU 是否还需要 TLB?有了之后会有哪些优缺点?

19.地址别名与映射变化之间有何区别?

20.exec 系统调用期间，内核如何保证 TLB 表项的一致性和虚地址缓存的一致性?

① 如果子集恰好在被映射页面的中间时，是 3 个不同的区。

② Mach 的早期版本使用 i 节点 pager，使用 vm_allocate_with_page，调用映射文件[Teva 87]。v 节点 pager，是开发出来支持 vnode/vfs 接口的[Klei 86]。

③ 甚至有可能用一个用户任务来完成缺省 pager 的功能。[Colu 91]介绍了这种实现方法。

④ 相反，如果任务 T2 试图写页面，服务器就要使用，should_flush 标志，请求内核 A 置它自己的页面副本无效。

⑤ 我们依从于这样的指定范围的标准：方括号标识包括边界，而圆括号不包括边界。

⑥ 由于 TLB 在新的 u 区映射后就被刷新了，Intel 的上下文切换实现中没有再调用 hat_asunload()来刷新 TLB。

⑦ 对待 seg_kmem 页面要稍有不同。这些面画是通过一个位图来管理的，而位图又分成若干区(缺省情况下，一个区的大小是 128 位)。每个区都有-个 cookie 相关联，当区中的地址被释放时设置这个 cookie。

⑧ 这一操作也同时刷新了进程的所有有效 TLB 表项。