

Slab 内存分配策略与移植

李 毅

(电子科技大学 计算机科学与工程学院, 四川 成都 610054)

摘 要: slab 内存管理算法具有分配和释放内存速度迅速、内外部碎片非常小等优点。介绍了 Linux 下该算法实现时采用的主要数据结构及相互间的组织关系, 阐明了其管理内存的实现机制。讨论了该算法的可移植性问题, 其中主要讲述了一个简便并且高效的分页系统的设计, 以及信号量移植的相关问题。

关键词: cache; slab; 内存管理; 分页; 信号量

中图分类号: TP311

文献标识码: A

文章编号: 1673-629X(2007)10-0168-03

Strategy and Transplantation of Slab Memory Allotment

LI Yi

(School of Computer Sci. and Eng., Univ. of Electronic Sci. Techn. of China, Chengdu 610054, China)

Abstract: Slab memory management algorithm has such advantages as low time consumption in allocating and freeing memory, little internal and external memory fragments. The implementation of this algorithm in Linux is presented at first, including its substantial data structures and those relationships. Transplantation of this strategy is discussed in the following, containing an effective paging design and issues in semaphores.

Key words: cache; slab; memory management; paging; semaphore

0 引言

slab 分配算法首先由 Sun 的工程师 Jeff Bonwick 提出并在 SunOS 中应用。由于该算法分配和释放内存迅速, 基本上不出现严重的外部碎片并且内部碎片非常小, Linux 2.2.0 之后的内核都采用了此种分配策略, 以满足对小块内存高效高速存取的需要。

1 Linux 下 slab 内存分配算法的实现

slab 分配器主要有三层数据结构: cache, slab, object, 三者的包含关系如图 1 所示。

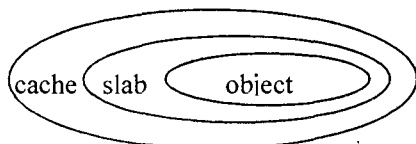


图 1 包含关系

cache 通过链表将属于它的 slab 组织起来。一个 slab 可能由 1 个、2 个、4 个或一直到最多 32 个连续页面构成, 其大小根据对象大小而定, 在初始化时通过计

算得出最合适的大小。Object 的内容最终存放在 slab 上^[1]。

1.1 重要数据结构及组织关系

cache 对应的结构 kmem_cache_t 及 slab 对应的结构 slab_t 在此算法中的作用非常重要。前者管理着该 cache 所对应的 slab, 后者管理着 slab 上所分配的 objects。

1.1.1 kmem_cache_t

该结构中重要成员有以下几个^[2]:

slabs_full, slabs_partial, slabs_free 分别指向满、部分满和空闲的 slab 队列。

objsize /* 该 cache 对应的 slab 中存放的对象的
大小 */

num /* 每个 slab 中对象的数目 */

gfporder /* 表明每个 slab 中页面数目是 2 的多少
次方 */

slabp_cache /* 采用 off slab 模式时指向 slab 控制
块对应的 cache 的指针 */

next /* 指向下一个 cache 结构的指针 */

1.1.2 slab_t

slab_t 结构中重要成员有以下几个^[2]:

list /* slab_t 通过此成员挂接到 cache 的某个队

收稿日期: 2006-12-03

作者简介: 李 毅(1982-), 男, 四川乐山人, 硕士研究生, 研究方向为集中式架构的三层交换系统; 导师: 章 毅, 教授, 研究方向为神经网络。

列中,如 slabs_free 队列 */

s_mem /* 指向第一个对象的指针 */

inuse /* slab 中已被占用的对象数 */

free /* slab 中第一个空闲对象的索引 */

slab 管理结构的存放方式有两种: on_slab 和 off_slab^[3]。对于大小不超过内存一页 1/8 的对象, slab 采用 on_slab 模式存放,即每块 slab 的控制信息 slab_t 就存放在该 slab 的开头位置。接下来是若干个(每个 slab 所存放的对象个数) kmem_bufctl_t (类型为无符号整形),每个 kmem_bufctl_t 与相同索引值的 object 一一对应。当要往 slab 中加入一个对象时,首先通过 slab_t 中成员 free 的值计算出空闲对象的地址,并更新 free 的值,将其填入下一个空闲对象的索引值。当发现 free 的值为 END 时,则说明本 slab 已经不存在容纳新对象的空间,此时更新 slab_t 中的 list, 将其挂到对应 cache 的 slabs_full 链表^[4]。 on_slab 的存放结构如图 2 所示。

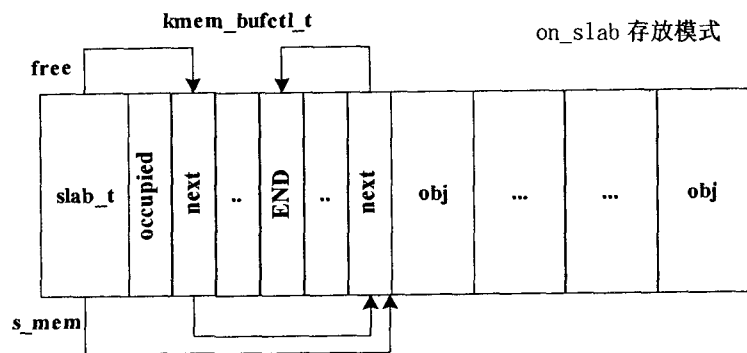


图 2 on_slab 存放模式

在采用 off_slab 模式时, slab 控制信息存放在该 slab 所在 cache 的 slab_cache 成员所指向的内存(成员 slab_cache 是指向 kmem_cache_t 的指针, off_slab 模式时 slab_t 结构就存放在它所指向的 cache 的某个 slab 内)。其结构关系如图 3 所示:

off_slab 存放模式

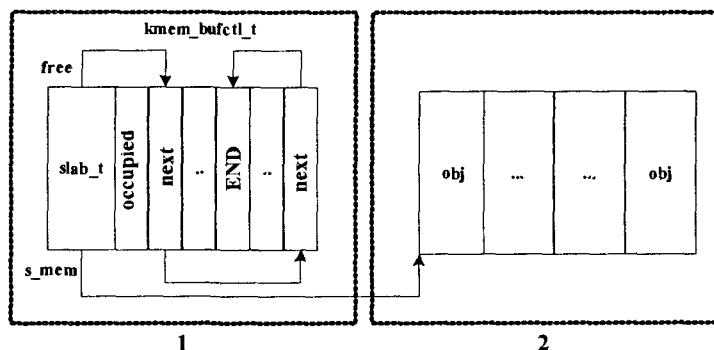


图 3 off_slab 存放模式

其中第一个方框的内容存放在该 slab 对应 cache 的成员 slab_cache 所指向的内存。

1.1.3 cache 与 slab 的组织关系

cache 中有相同属性的 slab(比如同为空闲 slab)通过 list 成员连接在一起,并链接到 cache 对应的链表上^[4]。各个 cache 又通过 next 成员组成链表,它们之间的相互关系如图 4 所示。

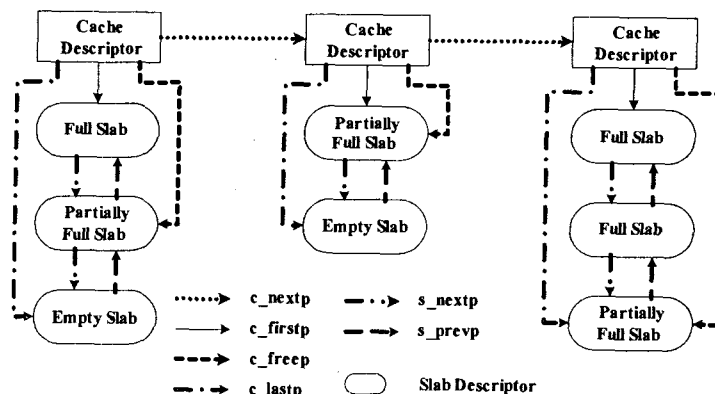


图 4 cache 与 slab 的组织关系

1.2 cache_cache 与 cache_sizes

为了更好地管理系统运行时产生的 cache 实例,引入了全局变量 cache_cache。该变量也是结构 kmem_cache_t 的一个实例,系统中所有其它 cache 实例都存放在 cache_cache 所拥有的那些 slab 里。也就是说每构造一个新的 cache 时,实际上是在 cache_cache 的 slab 中分配一个新 cache 的描述符。

用户在使用 slab 分配策略管理内存时,除了可以自己创建需要的 cache 外,还可以使用系统提供的通用 cache。这些通用 cache 都存放在全局数组 cache_sizes 里,并在初始化时在 cache_cache 对应的 slab 上为这些通用 cache 分配了内存空间。通用 cache 的对象大小从 32Byte、64Byte 一直到 128kByte。

用户在调用 kmalloc 函数请求分配内存时,slab 分配器将首先根据用户请求的内存大小在 cache_sizes 数组中找到合适的通用 cache,接下来在该 cache 对应的 slab 块上分配内存。这样一来,用户不必自己创建 cache 就能使用 slab 分配策略了。不过用户通过这种途径来获取内存会造成一定的内部碎片。比如用户请求分配 80Byte 的内存空间,但在通用 cache 中至少也得使用对象大小为 128Byte 的那个 cache 来进行分配。

1.3 分页对 slab 的支持

向 cache 申请新的内存时,如果该 cache 对

应的 slab 都没有空闲空间了,则会调用函数 `kmem_cache_grow` 来新获取一块空闲 slab。该函数最终将调用分页模块提供的接口获得所需的内存页面。同样在释放内存时也将适时调用分页模块的接口函数,将以前获得的内存页归还。由此可见,slab 并不是一个完全的内存管理算法,它只能算是一个内核内存分配模块^[4]。

2 slab 内存分配策略的移植

2.1 主要涉及内容

如上文所述,分页系统对 slab 算法提供了实现基础,要移植该算法,首先就需要分页系统的支持。在一些实时操作系统如 VxWorks 中,地址采用的是实地址模式,并不提供分页功能。因此,要在这些系统上实现 slab 算法,首先就需要自己建造一个简单的分页系统来支持 slab。另外,不同的操作系统对互斥、信号量等有不同的实现,移植工作自然也包括了这些功能在其它平台的实现。

2.2 分页系统的设计与实现

为了管理各页面,提出了页控制结构 `page`,它的主要字段如下所示:

`list /* next 和 prev 字段分别指向该页所分配的 cache 和 slab */`

`pointer /* 该页内存的起始地址 */`

`nextIndex /* 下一个 free 的索引号 */`

在释放一块内存时,用户提供的仅仅是它的内存地址。分页系统需要根据这个地址计算出它所对应的页面号,从而最终找到相应的页控制结构。

slab 模块在分页系统的协助下找到该页控制结构后,根据该结构的 `list` 字段就能知道这块内存位于哪个 cache 的哪块 slab 上,再根据内存地址就能很快算出它的索引值。接下来 slab 仅需要修改一下 free 链就能完成释放操作。可以看出,slab 的释放操作也是很迅速的。

由于 slab 需求的内存页以 2 为幂,大小从 4k 到 128k,因此可以把用户内存分为 6 个连续的区,每个区由若干个大小相同的内存页组成。其中第一个区存放 4k 大小的内存页,第二个存放的是 8k 大小的内存页,依此类推最后一个区存放的是 128k 大小的内存页。在内存起始阶段依次存放各内存控制表项。内存分划如图 5 所示。

2.3 信号量的移植

slab 中为了保证对 cache 链的互斥访问使用了信号量 `cache_chain_sem`,在修改缓存链之前通过 `down`

操作获取信号量,修改完毕后通过 `up` 操作进行释放,这样就保证了多进程对缓存链的互斥访问。在实时系统如 VxWorks 中,可以通过锁定任务的函数来完成互斥的操作。

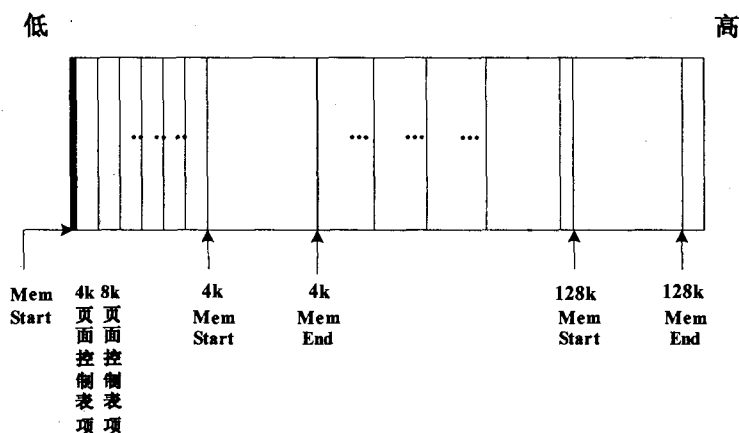


图 5 内存分划示意图

此外,cache 结构中还有自旋锁,并有一系列相关操作如: `spin_lock_irq`, `spin_unlock_irq` 等。实际上, `spin_lock` 在 `CONFIG_SMP` 的前提下才会生成的,也就是说,在单处理器系统这种简单情况下, `spin_lock_irq()` 就只需要锁中断就可以完成任务了。

2.4 与 Buddy 算法的简要比较

与 Linux 下采用的 Buddy 页分配算法相比较,此分页系统的页面分配和回收的速度是非常快的。因为它没有涉及到页面的拆分与合并操作,在分配内存时也不需要去查找合适的页地址。对于 slab 分配策略中的页分配与回收请求都能在一个常数时间内完成。

但性能提升的同时也在一定程度上降低了该算法的灵活性。比如当 128kB 大小的页面分配完后,如果此时 slab 还需要这么大的页块,分配请求是无法满足的。就算此时系统中有两个连续的 64kB 内存块,由于此分页系统的局限也无法满足这样的请求。不过一个特定系统在运行时对内存的需求是有一定规律的,在此分页系统中可以通过适当的调整各内存组块的数量来尽量减少或避免此类情况的发生。

3 总 结

侧重于介绍 slab 的基本实现原理以及移植所需的工作,这里省略了该算法中与此主题不太紧密的内容,如 slab 算法中为了更好地利用 CPU 缓存而采用的着色技术^[5],以及对 SMP 的支持^[4]等。

参考文献:

[1] 赵鲲鹏,苏葆光. Linux 内存管理中的 Slab 分配机制[J]. 现

(下转第 177 页)

性,可以通过对同一行业内部有关业务数据描述达成共识,进而设立行业数据标准作为企业之间相关业务数据的参照系。这样各企业就可以依据这个参照系进行数据交换,这就是行业数据交换的理论依据。

下面以企业 A、B 之间的协同服务交互过程为例来说明行业数据交换的原理:

(1)把行业数据标准 X 存储在行业供应链管理平台成为行业标准数据格式模板;

(2)接收 A、B 的企业私有数据格式 F(A)、F(B),存储在企业私有数据格式模板,并建立与行业标准数据格式模板之间的映射关系,即 $F(A \rightarrow X)$, $F(X \rightarrow A)$, $F(B \rightarrow X)$, $F(X \rightarrow B)$;

(3)依据上述映射关系,由 $F(A \rightarrow X)$, $F(X \rightarrow B)$ 自动推导出 $F(A \rightarrow B)$,由 $F(B \rightarrow X)$, $F(X \rightarrow A)$ 自动推导出 $F(B \rightarrow A)$,就建立了企业 A、B 间的直接数据交换关系 $F(A \rightarrow B)$ 和 $F(B \rightarrow A)$,并存储到数据模板映射中心;

(4)企业 A 向企业 B 进行服务请求时,首先检查本地有没有 $F(A \rightarrow B)$,若没有则到行业供应链管理平台去下载并缓存到本地,然后进行数据交换,即: $DATA(A) \rightarrow DATA(B)$,这样就得到了 $DATA(B)$,再封装成 SOAP 请求发送到企业 B;

(5)企业 B 接收并解析 SOAP 请求,调用 Web 服务实现处理请求,得到处理结果,检查本地有没有 $F(B \rightarrow A)$,若没有则到行业供应链管理平台去下载并缓存到本地,然后进行数据交换,即: $DATA(B) \rightarrow DATA(A)$,这样就得到了 $DATA(A)$,再封装成 SOAP 应答发送到企业 A;

(6)企业 A 接收并解析 SOAP 应答,得到结果数据。整个服务请求、应答过程结束,企业 B 对企业 A 的服务请求同理。

(上接第 170 页)

代计算机(专业版),2006(5):93-97.

[2] 毛德操,胡希明. Linux 内核源代码情景分析(上册)[M]. 杭州:浙江大学出版社,2001.

[3] 杨伟,刘强,顾新. Linux 下的存储管理[J]. 电子科

(上接第 173 页)

知识相结合,避免了传统手工收集表单信息的弊端,为实现信息提取自动化提供平台。由于技术等条件的限制,本系统仍在设计中。

参考文献:

- [1] 张笈秋. 深网的概念、规模及内容[J]. 中国信息导报,2004(10):57-60.
- [2] Sherman C, Price G. The Invisible Web: Uncovering Sources

4 结 论

为屏蔽企业计算环境的分布性、异构性,解决行业供应链上企业间应用集成的难题,提出了一个面向行业供应链的企业应用集成架构参考模型,分析了模型实现的关键技术。该模型综合运用了 Web 服务、J2EE 平台和行业数据交换原理,具有简单、开放、安全、高效、标准、可扩展的特点。

文中所提出的集成架构模型已经初步应用于中国摩托车商务平台-协同供应链管理系统,是一个典型的行业供应链的企业间集成的应用。此外,面向行业供应链的企业应用集成架构模型还适用于多种行业(特别是制造行业)供应链上企业之间的集成与协同商务应用,为行业供应链管理系统的开发和实施提供了一个通用的参考模型。

参考文献:

- [1] 黄国青,章勇. 面向供应链管理的企业应用集成技术选择模型[J]. 计算机工程与应用,2005(23):221-229.
- [2] 陈兵兵. SCM 供应链管理——策略、技术与实务[M]. 北京:电子工业出版社,2004:626-627.
- [3] 陈传波,张道杰,李涛. 基于 Web 服务的企业应用集成模型研究[J]. 计算机工程与科学,2004,26(12):15-29.
- [4] 柴晓路. 技术剖析:传统应用与 Web 服务的接口[EB/OL]. 2002-09-26. http://industry.ccidnet.com/art/732/20020926/26335_4.html.
- [5] 张玉东,刘广钟. 基于 J2EE 平台和 Web 服务的企业应用集成方案[J]. 计算机工程与设计,2004,25(11):2015-2017.
- [6] 殷庆,刘卫宁. 面向行业数据交换中间件 EasySwitch 的系统设计与实现[J]. 计算机科学,2004,31(7):159-162.
- [7] 技术,2005(9):7-10.
- [4] Gorman M. 深入理解 Linux 虚拟内存管理[M]. 白洛,刘森林等译. 北京:航空航天大学出版社,2006.
- [5] Bryant R E, O'Hallaron D. 深入理解计算机系统[M]. 龚奕利,雷迎春译. 北京:中国电力出版社,2005.02.
- [6] Search Engines Can't See[J]. Library Trends,2003(2):282-298.
- [3] Lackie R J. Those Dark Hiding Places: The Invisible Web Revealed[EB/OL]. 2005-02-25. <http://library.rider.edu/scholarly/rlackie/Invisible/Inv-Web-Main.html>.
- [4] 吴志强,严贝妮. 从隐蔽网络到国际互联网信息资源控制计划[J]. 图书情报工作,2004,48(3):82-85.
- [5] 罗海蛟,刘显. 数据挖掘中分类算法的研究及其应用[J]. 微机发展,2003,13(5):48-50.