一个基于 IA-64 体系的内存管理大页面的实现模型

陈鸣春　潘金贵

（南京大学计算机软件新技术国家重点实验室　南京 210093）

摘　要　本文提出了一种基于 IA-64 体系结构的内存页面大页面化的模型，可执行文件 ELF 的 Data Segment 使用大页面。由于转换解析缓冲区（TLB）能映射更大的虚拟内存范围，从而可减小未命中率，因此可以提高使用大页面的高性能计算（HPC）应用程序或使用大量虚拟内存的任何内存访问密集型应用程序系统性能。

关键词　大页面，ELF，Data Segment，IA-64

A Model of Realization of Large Page of Memory Management Based on IA-64

CHEN Ming-Chun　PAN Jin-Gui

(State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093)

Abstract　A model of large page of memory management is put forward in this article, and the Data Segment of the ELF file uses large page of memory. Because when large page is used, TLB(translation lookaside buffer) can map much larger virtual memory area, which leads to the reduction of miss rate. Therefore, computing performance of HPC applications or any other applications which use a large amount of virtual memory and access the memory extremely frequently will be improved.

Keywords　Large page, ELF, Data Segment, IA-64

1　引言

目前，一般操作系统的内存管理都是段页式，内存按 4k 分页。但是通常用来处理有关军事，气象，经济等方面海量计算量应用程序，频繁的切换导致了计算性能的低下。

TLB(translation lookaside buffer)是一个高速缓冲存储器，它将虚拟地址直接映射到物理地址，而不必通过页表。这是一个很昂贵但是却很有效的硬件设备，所以操作系统希望能最大限度地利用它。而且这种优化显得尤其重要，因为现在的物理内存越来越大，达到 GB 数量级。如果使每个 TLB 目录项能够映射一块更大的内存分页时，则会提高程序运行时访问内存的命中率，从而有助于提高性能。

IA-64 体系结构支持多个 page size：4K，8K，64K，256K，1M，4M，16M，256M。Linux kernel 2.6 中提供了两种方法对 large page 的支持，即 HugeTLB：①调用 mmap 函数；②调用 shmget/shmat 函数。

本文提出了一种基于 IA-64 体系结构使用大页面的模型。它使得动态链接的 ELF 可执行文件在运行动态链接器的时候申请一块大页面，然后将 ELF 文件的 Data Segment 运行在大页面的内存中。

2　ELF 文件

2.1　ELF 文件整体结构

Elf 文件有三种类型：①后缀. o 的目标文件，这是可重定位文件；②后缀. so 的动态库文件，这是共享库；③可执行文件。

ELF 文件中代码，链接信息，注释都是以段存放。在节头表中有一个表项与每个小节对应。一个 ELF 文件的总体结构如下：

```
ELF header (ELF 头部)
Program header table(程序头表)
Segment1
Segment2
  Section1
  Section2
  …
  Sectionn
…
Segmentn
Section header table(节头表)
```

2.2　ELF 文件头

ELF 文件开始处的一段叫 ELF 文件头。它记录了程序头表在文件中的偏移，程序头表的表项数目，程序头表每项的字节长度，节头表在文件中的偏移，节头表的表项数目和节头表每个表项的字节长度。而且他还记录了节名表所在节的索引序号。

ELF 文件头的数据结构如下：

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];
/* Magic number and other info */
  Elf64_Half e_type;
/* Object file type */
  Elf64_Half e_machine;
/* Architecture */
  Elf64_Word e_version;
/* Object file version */
  Elf64_Addr e_entry;
/* Entry point virtual address */
  Elf64_Off e_phoff;
/* Program header table file offset */
  Elf64_Off e_shoff;
/* Section header table file offset */
  Elf64_Word e_flags;
/* Processor-specific flags */
  Elf64_Half e_ehsize;
/* ELF header size in bytes */
  Elf64_Half e_phentsize;
/* Program header table entry size */
```

陈鸣春　硕士研究生，研究方向为高性能计算；潘金贵　教授，博士生导师，研究方向为中间件、Agent 技术、多媒体远程教育。

```
  Elf64_Half e_phnum;
/* Program header table entry count */
  Elf64_Half e_shentsize;
  /* Section header table entry size */
  Elf64_Half e_shnum;
/* Section header table entry count */
  Elf64_Half e_shstrndx;
/* Section header string table index */
 } Elf64_Ehdr;
```

2.3 ELF 程序头

在 ELF 文件头部的后面是程序头表，它是一个结构数组，程序头从加载执行的角度来看待 ELF 文件的结果。从这个角度看，ELF 文件被划成很多段，每个段保存着用于不同目的的数据，有的段保存着机器指令，有的段保存着已经初始化的变量，有的段会作为进程映像的一部分被操作系统读入内存，有的段则只存在文件中。

ELF 文件程序头如下：

```
typedef struct
{
    Elf64_Word p_type;
  /* Segment type */
    Elf64_Word p_flags;
  /* Segment flags */
    Elf64_Off p_offset;
  /* Segment file offset */
    Elf64_Addr p_vaddr;
  /* Segment virtual address */
    Elf64_Addr p_paddr;
  /* Segment physical address */
    Elf64_Xword p_filesz;
  /* Segment size in file */
    Elf64_Xword p_memsz;
  /* Segment size in memory */
    Elf64_Xword p_align;
  /* Segment alignment */
} Elf64_Phdr;
```

3 动态链接

链接就是将不同部分的代码和数据收集和组合成为一个单一文件的过程。随着虚拟地址的出现，每一个程序都有一个属于自己的空间。计算机总是运行一个以上的程序，经常是同一个程序的几个拷贝。如果不同体系结构的计算机运行实现同一功能的程序，那么这个程序的很大部分都是一样的的，只是其中的一小部分是和各个不同体系有关的。如果这个不变的部分能够从中分离出来，那么操作系统能够节省很多的空间。即使不同程序在同一个计算机上面运行，这些不同的程序经常有很多公共的代码。比如，几乎所有的 C 程序都使用 fopen 和 printf 函数，所有的数据库的应用程序都使用大量的访问函数去连接数据库，运行在 GUI 诸如 X Window，MS Windows，或 Macintosh 都使用了 GUI 库。

大多数的系统现在都提供了共享库，这样所有的使用共享库的程序都共享一个拷贝，因而提高了程序运行时刻的性能，并且能够节省很多的磁盘空间。特别在小程序中，公共的库函数比程序本身占用更多的空间。

共享库可分为静态链接和动态链接。在相对简单一点的静态共享库中，每个库都在它建立的时候被绑定在一个具体的地址上，在链接的时候，链接器把程序对库函数的引用绑定到这些特定的地址上面。静态库很不方便，因为当库中的任何部分都改变时候，程序都要重新链接。这样系统就引入的动态连接库的概念。库的节和符号都不会限定在实际的地址上面，直到使用这些库的程序开始运行为止。有时候绑定会更加延迟，直到第一次调用共享库中的函数才绑定地址。

动态链接的优点是：动态链接的共享库比静态链接的共享库容易生成，并更加容易升级；动态链接的共享库的语义更加接近不共享的库；并且动态链接能够在运行时加载和卸载共享库。

4 实现模型

4.1 IA-64 linux 虚拟空间

IA-64 支持 64 位虚地址空间，它将 64 位虚地址空间分成 8 个相等的区，每个区 2048P 字节。

7 区（对等映射）
6 区（非 cache）
5 区（页表映射）
4 区（堆栈）
3 区（数据段）
2 区（正文段）
1 区（共享内存段）
0 区（IA32linux）

图 1 IA-64 Linux 地址空间划分

4.2 具体的模型

Linux 内核加载可执行文件的 Data Segment 和 Text Segment 到虚拟空间中，如果是动态链接的，则加载动态链接器到相应的虚拟空间。由于内核都是申请的普通页面，所以我们提出的模型是首先加载我们自己写的动态链接器 myld.so，在 myld.so 中申请一块大页面，把可执行程序的 Data Segment 拷贝到大页面中。然后加载系统的 ld.so，再跳转到系统的 ld.so。

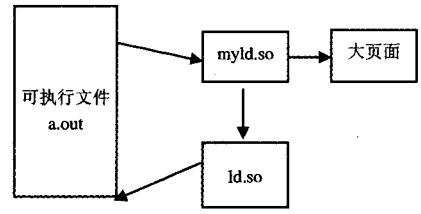

图 2 具体模型如上所表示

5 实现步骤

Step 1 设置栈。在 myld.so 中要将可执行文件中的参数传递给系统 ld.so。

首先分配一块内存空间，可执行文件 main 函数中的参数压入堆栈。

先是 main 函数中的 argc，然后是指向指针的 argv，后面是指向环境变量的指针的指针，再后面是 auxv，这是一组有关动态链接器变量的数组。然后就是具体的指向某一个命令字符串的指针 argv[0]，argv[1]…再后面是指向具体环境变量的指针 envp[0]，envp[1]…最后是这些指针所指向的真正的字符串。

具体的细节如图 3。

Step 2 加载系统动态链接器。动态链接器也是一个可执行文件 ELF 格式，因而它也有程序头表。对所有在程序头表中的 PT_LOAD 属性的段，都要加载到进程空间。可以根据 p_offset 找到所要加载的 segment 的起始地址，大小为 p_filesz。每一个段都有一个虚拟地址 p_vaddr 和在内存中的大小 p_memsz，将这个段加载到这个虚拟地址中，由于在内存中是使用分页机制，因此要用 p_align 对齐 p_memsz。

Step 3 跳转到系统动态链接器。每个 ELF 可执行程序都有一个程序入口点，系统的动态链接器也不例外。在可执

行程序的文件头结构有 e_entry 成员，这就是程序的入口点。然后用汇编无条件跳转语句跳到系统的动态链接器的 entry point。

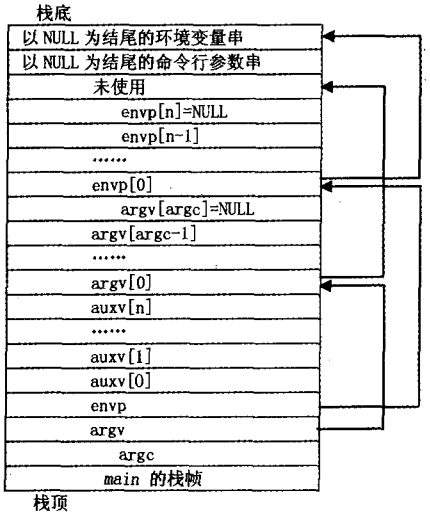

图 3　栈的结构

6　实验和分析

测试环境：

1. Red Hat Enterprise Linux AS release 4 (Nahant Update 2)

2. glibc. 2. 3. 4

3. 大页面大小为 256M，普通页面大小为 16k

实验方法：

每次申请一块表 1 所示大小的内存，分别用大页面和普通页面。对其每一个单元顺序地进行赋值操作，并记录时间。然后用所申请内存的大小除以所用时间得出效率，参见表 1。

表 1　测试用例及结果

申请的内存大小（单位：B）	使用大页面效率（单位：MB/S）	不使用大页面效率（单位：MB/S）	标志
256	1220. 20	1220. 70	A
1024	9882. 81	9765. 62	B
8192	9765. 62	10020. 83	C
16354	13020. 83	13531. 25	D
262144	11792. 45	10373. 44	E
1048576	10515. 25	8319. 47	F
16777216	2170. 21	2000. 09	G
268435456	2154. 65	1850. 0	H
1073741824	2199. 53	1900. 54	I

结论：

其中到标志 D 时，所申请的内存大小为 16k，此时由于普通页面不需要换页面，因此表现出来的是大页面和普通页面的效率差不多，但是所申请的内存大小更大的时候，由于大页面不需要更换页面或者更换的次数比较少，因此表现出来的效率要高。

但是大页面不是在任何场合都能够高效运转，因为如果运行小程序，大页面的内存浪费比较严重，而且如果程序中调用多次 fork()，则由于要反复申请大页面，开销很大。

结束语　本文提出了一个实现 ELF 可执行文件的 Data Segment 的大页面化的方法。这是一个很新的课题，对技术能力要求比较高。下一阶段的工作是对 ELF 可执行文件的执行空间中的栈和堆使用大页面。

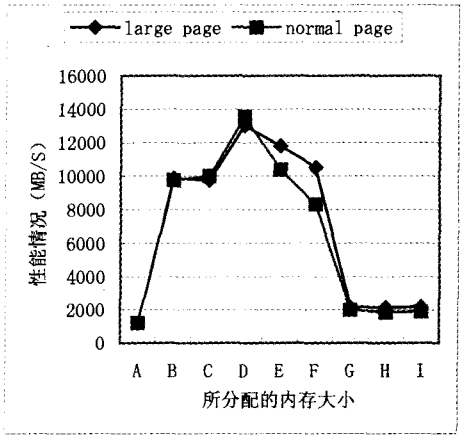

图 5　比较结果

参 考 文 献

1　Bryant R E, O'Hallaron D 著. 龚奕利，雷迎春译. 深入理解计算机系统. 中国电力出版社，2004. 524

2　倪继利著. Linux 内和分析及编程. 电子工业出版社，2005，9：86，552～555

3　http://www. phrack. org/phrack/58/p58-0x05 grugq & scut，2001

4　http://lists. grok. org. uk/pipermail/full-disclosure/attachments/20040101/fea4fb1f/ul_exec. txt

5　http://downloads. securityfocus. com/library/subversiveld. pdf grugq，2001

6　http://www. madchat. org/coding/Cheating_elf. pdf

7　http://www. skyfree. org/linux/references/ELF_Format. pdf

8　http://www. iecc. com/linkers

（上接第 272 页）

参 考 文 献

1　Darbhá S, Agrawal D P. Optimal Scheduling Algorithm for Distributed-Memory Machines. IEEE Trans, Parallel and Distributed Systems，1998，9(1)：87～94

2　Park C I, Choe T Y. An Optimal Scheduling Algorithm Based on Task Duplication. IEEE Trans Computers，2002，51(4)：444～448

3　Kwok Y K, Ahmad I. Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors. IEEE Trans Parallel and Distributed Systems，1996，7(5)：506～521

4　Ahmad I, Kwok Y K. On Exploiting Task Duplication in Parallel Program Scheduling. IEEE Trans Parallel and Distributed Systems，1998，9(19)：872～891

5　Yang T, Gerasoulis A. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. IEEE Trans Parallel and Distributed Systems，1994，5(9)：951～967

6　Gerasohlis A, Yang T. A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors. J Parallel and Distributed Computing，1992，16(4)：276～291

7　Palis M A, Liou Jing-Chiou, Wei D S L. Task Clustering and Scheduling for Distributed Memory Parallel Architecture. IEEE Trans Parallel and Distributed Systems，1996，7(1)：46～55

8　Colin J Y, Chretienne P. Cpm Scheduling with Small Computation Delays and Task Duplication. Operation Research，1991，39(4)：680～684