基于 Minix 的进程间通信系统的设计与实现①

陆冠群 胡 光 涂时亮 (复旦大学 计算机科学与技术学院 上海 200433)

摘 要: 进程间通信作为操作系统中最重要的原语之一,提供了在多个隔离的进程之间相互通信交流的可能性.提出了一种适用于微内核操作系统的进程间通信系统,并在具体的 Minix 操作系统平台之上予以实现,解决了 Minix 系统中由于进程间通信模块的缺失而导致开发人员无法顺利移植其他平台的实用程序的问题。实验数据表明,该进程间通信系统具有高效的特点;同时由于该系统的设计原则,它保持着易扩展的特点。该系统的实现虽然是基于 Minix 平台,但该设计同样适用于其他微内核的系统,对其他操作系统具有借鉴意义。

关键词: 进程间通信;共享内存;信号量;同步;虚拟内存;操作系统;Minix;微内核

Design and Implementation of Inter-Process Communication on Minix

LU Guan-Qun, HU Guang, TU Shi-Liang

(School of Computer Science and Engineering, Fudan University, Shanghai 200433, China)

Abstract: Inter-process communication (IPC), as one of the most important primitives in the operating system, provides the possibility of exchanging data with several different processes. This paper proposes a new design of the inter-process communication system. And this system is implemented on Minix operating system which lacks the support of IPC before. With the implementation, it eases the developers' porting and developing useful programs based on this feature. It is extensible and efficient as seen from the experiment. The design of this inter-process communication system can be studied for other operating system.

Keywords: inter-process communication; shared memory; semaphore; synchronization; virtual memory; operating system; Minix; micro-kernel

进程间通信作为操作系统中最重要的原语之一,提供了在多个隔离的进程之间进行通信交流的可能性。Minix 3 作为一个正在快速发展中的微内核操作系统,十分遗憾地尚缺乏符合 POSIX 标准的进程间通信系统。该功能的缺乏直接导致了某些重要的应用程序(比如 PostgreSQL)无法成功地移植到 Minix 3 平台中,另外使得在该平台上开发多进程合作的程序变得异常复杂。

本文的项目基于与阿姆斯特丹的 Vrije 大学合作,设计开发了基于 Minix 3 操作系统且完全符合 POSIX 标准的进程间通信系统。

本文立足于进程间通信系统的设计和实现,同时展示了如何在微内核的架构上实现具有良好语义规范的进程间通信系统,并且比较了微内核与宏内核在实现同样功能上的差异。主要的研究意义基于下面几点:

(1)研究如何高效实现基于微内核的进程间通信。

(2)研究如何有效利用微内核的安全机制来保障进程间通信的安全。

(3)研究进程间通信所能够解决的诸多现实问题。

通过本文所阐述的工作,基于本文开发的所有代码以及相应移植程序都已经成功进入到 Minix 3 系统的主干代码仓库。

本文的结构组织如下:在第一部分中,将对 Minix 系统做相应的背景介绍;在第二部分中,将详细阐述

① 收稿时间:2009-11-06;收到修改稿时间:2009-12-19

所提出的进程间通信系统的总体架构，并在接下来的篇幅中分别对共享内存模块以及信号量集模块的设计与实现进行描述；在第三部分中，则会通过实例分析对该系统做进一步的验证。最后一部分进行总结并简述进一步可研究的方向。

1　Minix背景介绍

Minix 系统是由 Vrije 大学的 Andrew Tanenbaum 领导开发的一个操作系统。追根溯源，Minix 系统在二十多年前就开始了它的雏形，曾经还有过 Minix 的开发者和 Linux 的开发者之间关于微内核与宏内核优劣的激烈论战。现在 Minix 已经开发至第三代。Minix 3 不同于先前作为传统教学工具的 Minix 1 和 Minix 2，是一个全新的开源操作系统，旨在于高可靠性，灵活性和安全性，可以被使用在有资源限制的工作环境上或者嵌入式环境中[1,2]。

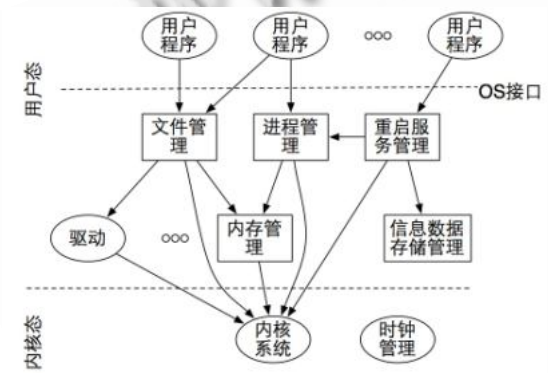

图 1　Minix 3 系统组成

如图 1 所示，Minix 3 按照微内核的思想来设计和实现整个操作系统，它真正的内核非常小，只有四千行左右的可执行代码。而操作系统的其他部份被分成多个互相独立的模块，并且都运行在用户态。比如：每个驱动程序都作为一个独立的用户进程而运行，这样，即使这个驱动程序中有错误发生也不会影响到操作系统的其他部份。甚至连虚拟内存、进程管、文件管理等模块也运行在用户态。

更重要的是，在重启服务管理模块以及信息数据存储管理模块的帮助下，驱动程序以及其他重要的模块在遇到错误的时候可以在用户完全透明的情况下重新启动[3,4]。正是这样的保障机制，Minix 3 极大地提高了操作系统的可靠性。

2　基于Minix的进程间通信系统

2.1 进程间通信系统概述

在操作系统的世界中，进程是所有任务的主体，每一个进程就代表了一个可执行的代码线索以及所能使用的资源集合。各个进程拥有独立的地址空间，无法相互访问。但是很多时候，两个不同的执行线索之间需要跨越这种鸿沟，来进行协调工作，跨越的方法有很多，按照两个不同的执行线索在物理上的差异，可以分为以下两类[5]：

(1)不同计算机的远距离范围。在跨越不同计算机之间的协调合作可以通过网络协议来完成，比如传统的 C/S(服务器/客户端)模型、RPC(远程过程调用)方法等。

(2)同一操作系统范围。在同一个操作系统之上，可以采用线程模型在同一个进程中进行通信，或者采用进程模型并结合进程间通信方式来完成信息的传递。

本文主要侧重于后者，详细阐述了符合 POSIX 标准的进程间通信系统设计。POSIX 标准所标识的进程间通信包括共享内存(shared memory)、信号量(semaphore)和消息队列(message queue)。其中最被广泛使用的就是共享内存与信号量，特别是信号量，作为一种有效的同步机制，为多个进程互相操作提供临界区的保护支持。

2.2 进程间通信系统总体架构设计

我们将进程间通信作为一个基于用户态的独立模块。采用这种方案具有如下的优点：

一致性。这种分离的架构符合当前 Minix3 的整体系统框架，可以与整个系统无缝结合，甚至可以做到随需启用，而不需要整个操作系统的重启。

健壮性。即使该模块发生了致命的错误，也丝毫不会影响到整个系统的正常运行。而且该模块会被立即重新启动，这些动作对其他用户完全透明，完全不需要用户的干预。因此，对于使用进程间通信服务的用户而言，不需要引入额外的错误处理机制，健壮性由此得以体现。

扩展性。共享内存与信号量集模块都建立与进程间通信的框架之上，如图 2 所示，今后可以按需开发更多的子模块来构建更多有用的进程间通信原语。

对于用户进程而言，它只与库函数进行通信，而库函数作为中间的代理者，进行必要的请求包装，随

即将请求发送至进程间通信系统。而基于进程间通信架构之上的各个子模块在初始化阶段分别向进程间通信主模块注册自己基于消息的回调函数；此后，当进程间通信模块获得了某个请求，就会根据之前注册的信息将该请求分发到对应的子模块之中，继而该模块按照具体的回调函数逻辑进行处理。

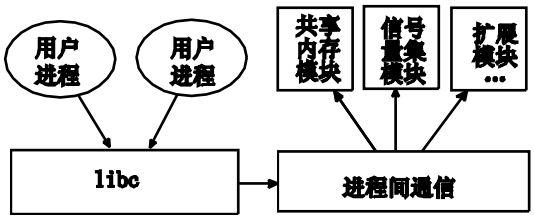

图 2　系统架构

2.3 共享内存模块的设计与实现

共享内存是指系统中的一块内存，该内存可以同时被多个互相独立的进程程序访问而无需额外的复制操作[6]。

共享内存的优点是读取速度快，对于共享内存的读取操作与进程的其他内存数据读取没有任何差异。但也存在缺点：1)共享内存本身不提供任何的同步机制，无法保护多个进程同时读写的情况。因此，当多个进程在独立对该内存段进行操作的时候，需要利用额外的机制来保证操作的原子性。2)共享内存是系统级的数据。除非明确指明销毁该共享内存段，否则当进程退出的时候它仍然存在于系统之中，这就会导致内存泄漏以及一定的信息安全隐患。

由于本文设计的进程间通信是作为一个独立的架构而存在，并不依附于底层的内核，因此当用户进程在试图获取一个共享内存段时，实际的内存分配是发生在这个进程间通信的地址空间中，如图 3 所示，而目的是将地址空间映射到其他用户进程。当共享内存收到用户进程的请求之后，如图 4 所示，在更新内部存储的信息之外，主要通过调用虚拟内存的帮助程序以完成必要的功能。

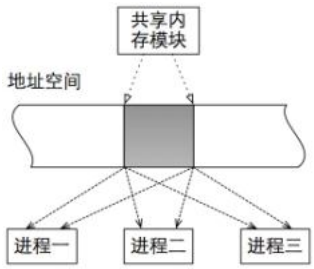

图 3　共享内存分配示意图

因此，实现共享内存的根本就在于对于虚拟内存管理模块的功能扩展。为了实现共享内存的原语，虚拟内存管理模块必须提供如下的一些功能：

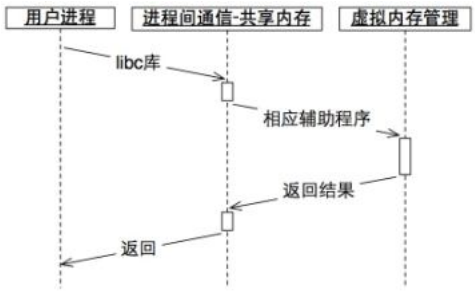

图 4　操作共享内存的时序图

(1) 重新映射的功能。当一个进程可以访问到某一个地址，这就意味着在该进程的页表中存在着一个{虚拟地址，物理地址}的地址对，那么当系统在访问该虚拟地址时，在内存管理单元(MMU)的帮助下，实际的物理地址会被寻址。因此，修改相应的页表就会使得一个虚拟地址去寻址不同的物理地址。既然虚拟内存管理模块拥有所有进程的页表信息，就可以通过将一个进程的一个虚拟地址同时映射到另外一个进程的某个地址上，这样就解决了一个进程连接到一个共享内存段的关键问题。

本文设计实现了如下的函数原型：

```
int vm_remap(endpoint_t src_e, void *src_p,
             endpoint_t dst_e, void *dst_p,
             size_t len)
```

该语义表示将源进程(src_e)的起始地址(src_p)映射到目的进程(dst_e)的起始地址(dst_p)，有效的映射长度为 len。

虽然这种方式有效，但是无形中引入了安全问题，因为既然提供了这样的功能，那么其他的用户进程同样可以采用该接口来使其他进程的内容重新映射到自己的进程空间中，那么就会轻而易举地窥探和获取到其他进程的信息。为了防止这种情况的出现，需要加入适当的权限控制机制。权限控制分两个层次，一方面从源头上来说，要防止这样情况的出现。在现有的 Minix 系统中，对于任何一个服务模块的启动，都必须静态地赋予一定的权限，因此其他的非法服务就无法使用这种“重新映射”的功能。另一方面，从虚拟内存管理服务器而言，它自身也会检查它所接受到的

请求者的来源，由于现在采用的是白名单的策略，在编译阶段就需要指定合法的程序。通过这两种方法，重新映射功能既可以良好地工作，又不至于由于安全隐患而使系统处于危险的境地。

(2) 由虚拟地址获得真正实际物理地址的功能。假设当某个共享内存段被多个应用程序重新映射到自己的地址空间之后，这些应用程序并不知道这块地址的实际物理地址，那么当某个进程需要分离共享内存段的时候，指定的只有虚拟地址，该虚拟地址会最终传递到共享内存模块中，如果没有将虚拟地址转换到物理地址的功能，共享内存模块也就无法正确判断该地址是否是有效的共享内存段的地址，甚至会发生错误的分离操作。

(3) 获得连接到共享内存段的进程个数的功能。因为共享内存模块是一个独立的模块，它并不能主动感知其他进程的变化。只有进程管理模块才会对所有的进程情况了如指掌，而且只有虚拟内存管理模块才拥有关于地址空间引用计数的信息。通过函数的形式将虚拟内存管理模块中该项信息导出，可以实现该功能，从而获得共享内存所需要的信息。

在虚拟内存管理模块上增加以上三项功能，也就基本完成了共享内存模块的实现，剩下的只有内部数据的简单维护与更新罢了。

2.4 信号量集模块的设计与实现

Dijsktra 提出了最早的信号量概念，是为了解决多个串行程序在并行执行的时候互相协调问题[7]。

本文所设计的信号量集模块同样基于进程间通信模块。对于信号量模块的设计最主要需要解决的问题就是如何保存各个进程的状态，以及如何控制他们使之各个用户进程在行为上表现得和 POSIX 关于信号量的标准一致。

基于上面的考虑，设计如下的数据结构以支持信号量：

如图 5 所示，通过一个信号量集队列来记录所有用户进程申请的信号量集，以便于其他用户进程可以获得并使用该信号量集。该信号量集需要包含对应的键值、标识符、关于该信号量的元信息、真正的信号量元素队列。在这里的元素队列中，其中的每一个元素类似于朴素的信号量实现，其中不仅仅含有必须具有的计数器的值，还包含一个等待队列。此外，信号量元素还含有最后一个对该元素进行操作的进程号这样的信息。所有这些设计的初衷都是为了能够方便地实现基于 Minix 平台的符合 POSIX 要求的信号量集接口。

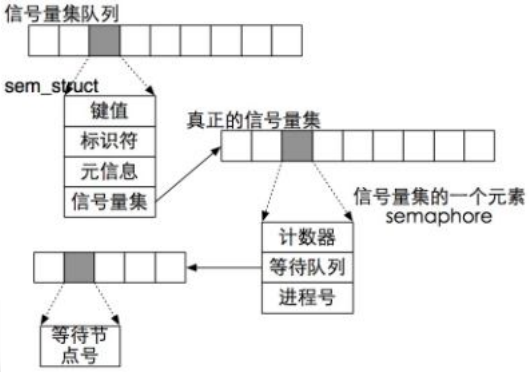

图 5　信号量集内部结构

当用户进程在获取新的整个信号量集时，如图 5 所示，它的操作层次仅仅在 sem_struct 队列；而当需要获得某个单独的信号量时，在模块内部的操作层次就在信号量元素 semaphore 之上，当然有时候它也需要遍历等待队列来判断应该唤醒哪个进程。

如图 6 所示，当某个用户进程试图获得某个信号量的时候，会通过库函数向信号量模块发送相应的请求，由于该请求为阻塞操作，如果经过信号量模块内部的逻辑判断该进程应该获得该信号量，那么就直接发送返回消息，这样用户进程就可以自动继续往下执行。但是如果信号量模块判断该进程不应该获得该信号量，那么不需要做额外的操作让该进程等待，因为该进程已经在阻塞等待了，唯一需要做的事情就是将该进程的结点号记录下，并在适当的时候将它唤醒(即向它发送返回消息)，那么从用户进程的角度而言，该获得信号量的操作是原子的，而且并不知道中间已经被隔离了一段时间，达到了与标准 POSIX 行为一致的行为。

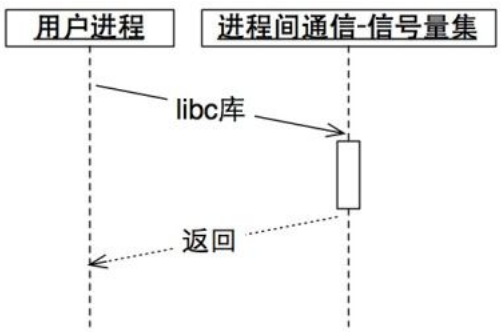

图 6　操作信号量集的时序图

2.5 与 Linux 相应系统的比较

Linux 中的进程间通信模块都在内核态实现。实现的复杂度相对于微内核的用户模块呈指数级的增加。

在实现中，Linux 中的进程间通信系统和其他子系统具有很强的耦合性，特别是共享内存，它必须与虚拟文件系统以及内存管理系统紧密地结合才能完成 POSIX 的功能。因此，当该系统中存在错误的话，必然会对整个操作系统产生致命的错误，已经有这样的例子[8]。

从对比中可以看到，微内核的进程间通信架构具有模块化的特点，能显著提高编程与调试的效率，另外通过将错误隔离在单一的模块中可以提高整个系统的安全性。

3 实例应用与验证说明

3.1 移植 PostgreSQL

除开用于验证进程间通信的单元测试以外，本文另外移植了流行的数据库软件 PostgreSQL，作为一个对象关系型数据库系统，它具有广泛的应用场景[9]。对于本文更重要的价值在于该数据库系统严重依赖于共享内存和信号量集这两种进程间通信方法来完成数据在多个进程之间的传送与同步。因此，PostgreSQL 自然成为验证进程间通信项目在 Minix 操作系统上正确性的一种重要手段。

在克服一些简单的移植问题之后， PostgreSQL 可以完全正常地运行在 Minix 3 操作系统之上，图 7 为运行时的终端截图，具体代码链接见参考[10]。

```
$ /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data > logfile 2>&1 &

$ /usr/local/pgsql/bin/createdb test-postgresql
$ /usr/local/pgsql/bin/psql test-postgresql
psql (8.4.0)
Type "help" for help.

test-postgresql=# create table mytable (id varchar(20), name varchar(30));
CREATE TABLE
test-postgresql=# insert into mytable values('Author', 'Lu Guanqun');
INSERT 0 1
test-postgresql=# select * from mytable;
   id   |    name
--------+------------
 Author | Lu Guanqun
(1 row)

test-postgresql=# \q
$ uname -a
Minix 10.0.2.15 3 1.4 i686
$
```

图 7　PostgreSQL 运行截图

3.2 实验数据

本节通过实验研究了进程间通信在信号量产生竞争情况下的效率表现。所采取的实验如下：两个进程分别独立地试图获取信号量，在获取成功之后对两者所共同拥有的共享内存进行简单地操作，随即退出临界区，将该操作循环重复多次。该实验的设计符合了绝大多数的进程间通信的模式，具有代表意义。

如下表所示，其中第二列与第三列分别为两个进程不产生竞争与产生竞争的情况。我们最后考察两者运行时间的增加比率，用以说明竞争下的效率。整个实验无差异重复进行了 10 次，表中的数据为经过平均的数值。

表 1　进程间通信的效率比较表

循环数（次）	无竞争（秒）	竞争（秒）	增加负载
5000	18.50	18.52	0.1%
10000	35.06	35.51	1.3%
15000	54.73	55.48	1.4%
20000	70.36	72.10	2.5%

可以看到，由于信号量需要进行同步而产生的负载在合理的规模数量上并不显得十分突出，这就说明了进程间通信在信号量产生竞争的情况下仍保持着良好的执行效率。

4 总结与进一步工作

本文研究了基于微内核平台上的进程间通信架构，并在实际的 Minix 平台上加以实现，结束了在该平台上无法使用进程间通信的历史，并且顺利移植了 PostgreSQL 数据库软件。更重要的是，本文所完成的工作以及所取得的成果已经被运用到 Minix 3 的正式系统之中，详细代码链接见参考文献[11]。

但是在本文所涉及的领域中仍然存在研究的空间：虽然 PostgreSQL 的成功移植反映了在 Minix 平台上该进程间通信系统的正确性，但如果需要从形式上证明该实现的正确性，需要做更多的研究工作。当前，已经有一些模型检测方案，但是都是基于宏内核，缺乏对于微内核操作系统的统一研究，因此在这点上值得进一步的研究。

最后，需要感谢涂时亮老师对于这个项目的支持；另外需要特别感谢 Vrije 大学的助理研究员 Ben Gras，感谢他在搭建项目开发环境所给予的帮助并且对我们的设计文档提出了宝贵的意见。

参考文献

1 Herder JN, Bos H, Gras B, Homburg P. MINIX3: A

(下接第 14 页)

(上接第 5 页)

1 Highly-Reliable, Self-Repairing Operating System. Operating Systems Review, 2006,40:80－90.

2 Herder JN, Bos H. Reorganizing UNIX for Reliability. Proc. 11th ofACSAC, 2006:81－94.

3 Mancina A, Herder JN. Enhancing a dependable multi-server operating system with temporal protection via resource reservation. Real-Time Systems, 2009,43: 177－186.

4 Herder JN, Moolenbroek DC. Dealing with driver failures in the storage stack. Proc. Fourth Latin American Symposium on Dependable Computing. 2009: 119－126.

5 Andrew S.Tanenbaum, Albert S.Woodhull. 操作系统设计与实现.第三版.北京:清华大学出版社, 2008.178－179.

6 Daniel P.Bovet, Marco Cesati, 深入理解 Linux 内核. 第三版.南京:东南大学出版社, 2006.22－25.

7 EW Dijkstra. Cooperating Sequential Processes. Technical Report EWD-123,The Netherlands, 1965.

8 [2009-02-04]http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=a68e61e8ff2d46327a37b69056998b47745db6fa.

9 PostgreSQL 官方网站.[2009-08-04]http://www.Postgresql.org.

10 PostgreSQL 移植代码, [2009-08-04]http://gforge.cs.vu.nl/gf/project/minix/scmsvn/action-browse&path=%2Ftrunk%2Fbigports%2Fpostgresql-8.4.0%2F,2009－09－03.

11 Minix 3 代码仓库.[2009-08-04]http://gforge.cs.vu.nl/gf/project/minix/scmsvn/.