摘　要

集群系统是充分利用现有计算机资源的一个很好的解决方案,而要让集群系统获得良好的负载平衡、高通信性能、高可用性等特性，进程迁移功能是不可或缺的。

集群系统进程管理和调度的目标之一是负载分布，负载分布的目标是实现负载平衡。负载平衡是指在整个系统中尽量平均分配工作量，实现方法就是“进程迁移”，即将进程及其所有的相关状态信息重新放置到另一个处理器上。通过将进程从负载较重的节点移动到负载较轻的节点，负载就会得到平衡，从而改善整体性能。

为了实现负载平衡，本文需要做三个方面的工作：收集和监控集群系统中各节点的负载信息；掌握各进程信息，以决定何时需要迁移哪个进程到什么位置；进行进程迁移。

在 Linux 操作系统基础上,本文讨论了为集群系统提供进程迁移功能所需要的“负载监测”、“进程迁出”、“监听迁入”三个模块的设计和实现。其中，通过对现有相关工作的分析和比较，提出了动态进程迁移算法的改进方案，最大程度地将进程状态迁移和进程的运行并行起来，从而提高了迁移速度，网络通信量也较小，而且也没有对源节点的剩余依赖性。最终完成了预定的设计目标，使用测试程序验证了其有效性，并得出了性能比较数据。

关键词：集群；负载平衡；进程迁移；动态进程迁移

Abstract

Cluster system is one of the best methods to utilize existing idle computing resources. In order to achieve the good load-balancing, high communication performance and high availability of cluster systems, the function of process migration is required essentially.

Load distribution is one of objectives of process management and scheduling; the purpose of load distribution is load-balancing in cluster system, load-balancing is a technique which can average the workload of the entire system. The actual implementation of load-balancing is process migration, which can reassign a process and all the relative state information to another processor. Through processes' migration from an overloaded system to a lower-loaded, the workload of system can be balanced and the overall performance of the cluster system is improved.

In order to implement load-balancing, there are three required tasks: gathering and monitoring load information of each processor in cluster systems, monitoring the information of processes and migrating the process.

Based on Linux operating system, this thesis discusses the design and implementation of three modules for the process migration in cluster systems, which are load monitoring, process migrating out and monitoring migrating in. We present a new algorithm for dynamic process migration after the comparison of existing algorithms. The new migration algorithm speeds up the process migration, reduces the communication overhead and avoids residual dependencies. The performance of the proposed algorithm is compared with other algorithms under an application program. The efficiency of the implementation is proved by the experiment results and the predefined objectives are achieved.

Key words: cluster; load-balancing; process migration; dynamic process migration

哈尔滨工程大学
学位论文原创性声明

本人郑重声明：本论文的所有工作，是在导师的指导下，由作者本人独立完成的。有关观点、方法、数据和文献的引用已在文中指出，并与参考文献相对应。除文中已注明引用的内容外，本论文不包含任何其他个人或集体已经公开发表的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

作者（签字）：武星燕

日期：2008年3月12日

第1章 绪论

1.1 课题背景和研究意义

本课题来源于哈尔滨工程大学基础研究基金：分布式计算机系统容错策略的研究(HEUFT05009)与哈尔滨工程大学基础平台建设开发项目：集群计算系统研究。

从20世纪80年代末，进程迁移一直是国际上比较活跃的研究课题。它是集群(Cluster)系统中一项十分重要的技术，也是许多集群系统共同追求的设计目标。集群系统可以用其实现高吞吐量网络计算，提高系统的负载平衡，合理有效利用资源。其作用概括起来主要体现在以下几个方面[1]：

(1)实现集群系统动态负载均衡

由于集群系统中各个节点的负载不断变化且难以预测，经常出现突发性不平衡问题，因此只有使用进程迁移才能根据系统中各节点负载的变化实时地对计算子任务进行动态调度。将进程迁移到负载较轻或空闲的节点上，可以使系统内部的负载合理动态地分布，充分利用可用资源，发挥节点的计算能力，通过减少节点间负载的差异来全面提高性能，实现集群系统内部真正的动态负载均衡。

(2)实现高效率容错和可用性

在集群系统中，当一个节点发生故障时，将该节点上正在运行的进程迁移到其它节点恢复运行，这将极大地提高系统的可靠性，也保证了系统在遇到灾难时的可用性。在某些关键性应用中，这一点尤为重要。

(3)主人优先使用

在集群系统中有效利用系统中的空闲资源(主要是空闲节点)来完成计算任务，可大大提高集群系统的计算能力。但是，系统中的节点一般为不同用户所拥有，为维护各节点的自主性能，网络计算应利用各节点的空闲时间运行。因此当节点的主人要使用其机器时，系统应能及时地利用进程迁移将该机器上运行的外来任务尽快迁出，保证主人优先使用其机器。

(4)提高通讯性能

如果一个进程需要与其他进程频繁地进行通讯，这时将这些进程放置得近一些，具体方法就是将一个进程迁移到其它进程所在的CPU上，则会减少通讯的负担，提高通讯性能。

(5)向数据迁移

一些向文件服务器请求大量数据的进程，需要通过网络从文件服务器传输数据给进程，如果将进程迁移到文件服务器上进行IO操作，将极大地提高效率和减少网络通讯负担。

(6)使用集群中某些机器的特殊能力

如果某个进程能够从集群中的某台特定机器上受益，它就应该在那台机器上运行。进程的迁移可以充分利用特定节点上独特的软硬件资源。如进行数值计算的程序能够通过使用数学协处理器或超级计算机中的多个处理器来缩短程序执行时间。

尽管进程迁移对于集群系统性能提高，负载均衡等方面有很大的作用，但是由于其实现透明迁移的复杂性以及现有的商业操作系统自身不支持迁移等原因，使其并没有得到广泛应用。但是随着计算机中商用部件性能的不断提高和价格的不断下降，使得集群系统在高性能计算中的地位越来越重要。应用于高性能计算的集群系统中有大量的计算资源，如何有效管理和利用这些资源是一个非常重要的问题，进程迁移在这些方面的作用将越来越大。而且随着万维网和移动计算的兴起，进程迁移也被应用于基于移动Agent的透明性迁移研究[2-4]。

1.2 集群系统

集群系统是一种并行或分布式处理系统，它包含许多由通信互联网络连接起来的独立计算节点，每个节点有自己的存储器、I/O设备和操作系统，能够通过集群软/硬件形成一个统一的计算资源,并且提供了单个系统不能支持的高可用性、高可靠性和易伸缩性[5,6]。一般而言，构成集群系统的计算机可以是普通PC，工作站，多处理机以及大型服务器，大多数情况下它们运行的是同一种操作系统。

集群系统的种类有多种分法，从功能上说可以简单分成高性能计算集群和高可用性集群[7]。

(1) 高性能计算集群

简单的说，高性能计算集群(High-Performance Computing)是计算机科学的一个分支，它致力于开发超级计算机，研究并行算法和开发相关软件。高性能计算主要研究如下两类问题：

a) 大规模科学问题，象天气预报、地形分析和生物制药等；

b) 存储和处理海量数据，象数据挖掘、图像处理和基因测序。

顾名思义，高性能集群就是采用集群技术来研究高性能计算。高性能计算集群特别适合于在计算机中各计算节点之间发生大量数据通讯的计算作业，比如一个节点的中间结果影响到其它节点计算结果的情况。

(2) 高可用性集群

计算机系统的可用性(availability)是通过系统的可靠性(reliability)和可维护性(maintainability)来度量的。工程上通常用平均无故障时间(MTTF)来度量系统的可靠性，用平均维护时间(MTTR)来度量系统的可维护性。于是可用性被定义为：

$$MTTF/(MTTF+MTTR)*100\%$$

对于关键业务，停机通常是灾难性的。随着企业越来越依赖于信息技术，由于系统停机而带来的损失也越来越大。

高可用性集群(High Availability)就是采用集群技术来实现计算机系统的高可用性。高可用性集群通常有两种工作方式：

a) 容错系统：通常是主从服务器方式。从服务器检测主服务器的状态，当主服务工作正常时，从服务器并不提供服务。但是一旦主服务器失效，从服务器就开始代替主服务器向客户提供服务。

b) 负载均衡系统：负载均衡是把负载压力根据某种算法合理分配到集群中的每一台计算机上，以减轻主服务器的压力，降低对主服务器的硬件和软件要求。集群中所有的节点都处于活动状态，它们分摊系统的工作负载。一般 web 服务器集群、数据库集群和应用服务器集群都属于这种类型。

高性能计算集群上可开发并行应用程序从而解决复杂的科学问题；高可用性集群可为企业需求提供更实用的系统。实际的一个集群系统通常是上面

两种集群系统的混合。

集群计算系统的核心问题是资源的共享及有效利用。集群系统的作用是显而易见的。一般情况下，除了高负载的服务器，大多数的计算机在绝大多数时间里 CPU 是空闲的，如果有更多的计算机，这种浪费是惊人的。集群系统可以把现有的空闲资源充分利用起来，还能在服务器之间智能地进行负载迁移，保持系统动态平衡。近年来，随着微处理器技术和高性能网络技术的飞速发展，集群系统逐渐能够以低廉的价格提供与传统超级计算机相当的计算能力。因而，越来越多的科学和商业应用能够在集群系统上运行，获得较高的性能价格比。同时，随着 Internet 的日渐发展，大型 Web Server，Database Server 对计算机系统也提出了一些新的要求。这些企业不大可能花费巨资购买传统的、专用的超级计算机，对他们而言集群系统是一种非常适合的选择。

集群系统采用的操作系统主要有 UNIX、Microsoft Windows 和 Linux。目前，最具代表性的集群结构是美国 UC Berkerly 大学的 NOW(Network of Workstation)项目[8]和 NASA 的 Beowulf 项目[9]。

1.3 Linux简介

Linux 最初是由一名芬兰赫尔辛基的大学生 Linus Torvalds 在 1990 年利用微型 UNIX 操作系统 Minix 为开发平台，在自己的 Intel 386 PC 上，开发的 UNIX 类操作系统。Linus 把这一软件奉献给 GNU 计划，并公布了全部源代码。GNU 计划的宗旨是：消除对于计算机程序拷贝、分发、理解和修改的限制。也就是说，任何人都可以从网上下载、分析、修改、添加新功能。1994 年 Linus 发布 Linux 第一个“产品”版 Linux1.0，随后 Linus 转向 GPL 版权，这除了规定有自由软件的各项许可权之外，还允许用户出售自己的程序拷贝，并从中赢利。于是，Linux 得到更多专业人士和商业软件公司的支持，并通过 Internet 得以迅速发展。

Linux 以它的高效性和灵活性著称，作为 UNIX 类操作系统，具有下列特点[10-12]：

(1) Linux 是真正的多任务系统，允许多个用户同时在一个系统上运行多道程序。它还是真正的 32 位操作系统，工作在 Intel 80386 和后来的 Intel 处

理器的保护模式下。

(2)图形用户界面。XFree86是Linux平台上的X Window系统。X Window系统是功能强大的图形界面。

(3)支持TCP/IP协议。在Linux系统中通过Ethernet可连接到当地的局域网或Internet。

(4)虚拟内存和共享库。Linux采用虚拟内存技术扩展可用内存数量，同时使用共享库技术，允许使用标准子过程的程序在运行时共享子过程，从而节约系统空间。

(5)Linux 内核中的代码均为自由代码。Linux 是在自由软件基金会的GNU计划下开发的，而且世界各地的专业人士甚至商业公司也加入了Linux软件开发的行列。可通过Internet免费下载Linux。

(6)Linux 支持商业版 UNIX 的全部功能，而且其一些系统功能是 UNIX 系统所不具备的。

(7)GNU软件的支持，如GNU C和GCC编译器、gawk、groff等。

(8)Linux特别注重可移植性，符合IEEE POSIX. 1标准。

(9)Linux支持多种硬件平台。从低端的Intel386直到高端的超级并行计算机系统，都可以运行Linux系统。

(10)Linux 系统网络功能强大。内核中集成了网络功能和有大量的网络应用程序，在超强网络需求下有很好的健壮性。

Linux 与其它操作系统相比，还具有采用阶层式目录结构、支持多种文件系统和可与其它操作系统如Windows98/2000/xp并存于同一台计算机上等特点。

Linux 最初虽由 Linus 编写，但可以说其起源于 Internet。Linux 通过Internet不断发展，九十年代末期，Linux操作系统不断走向成熟，它的健壮性不断增强，最重要的是Linux在普通PC机上提供了对高性能网络的支持，这样就大大推动了基于Linux的集群系统的发展，以它为平台来构建集群有很多优点。

(1)廉价：与Windows以及其它商品化UNIX操作系统相比，Linux的一个显而易见的优势就是廉价。硬件的花销加上很少的软件费用就可以拥有一个PC工作站或服务器，这方面显然是其它操作系统无法比拟的。而且Linux

对于硬件的要求比 Windows 要低的多。

(2) 自由开放：自由开放是 Linux 的一个最迷人的特点。开放源码为提高性能提供了更加广阔的空间。开发者可以看到这个系统是怎样跑起来的，然后在操作系统一级进一步提高性能便成为可能。而在 Windows 或者 AIX 这样的操作系统中，得到它们的源码已是很不容易，要想轻松地从操作系统着手来优化上层的大型应用更是难上加难。

(3) 高效：目前，由于比较缺乏对 Linux 的性能和功能评价的系统科学研究，在同等硬件配置和应用环境下，Linux 与其它操作系统相比孰优孰劣还不太明朗。但是，已经有不少数据说明，作为工作站或小型服务器，Linux 已经可以与它对手一叫高低了，尤其是它的网络性能以及可靠性都备受称赞，而这些正是一个高效集群不可缺少的。

所谓 Linux 集群，就是利用商品化的工业标准互联网络，将各种普通 Linux 服务器连接起来，通过特定的方法，向用户提供更高的系统计算性能、存储性能和 I/O 性能，并具备单一系统映像(SSI)特征的分布式/并行计算机系统。与 SMP(对称多处理系统)、MPP(大规模并行处理)及 Beowulf 集群相比，Linux 集群在性能价格比、可靠性、可扩展性、可管理性和应用支持性方面有着更为明显的优势。著名的搜索引擎 Google 就是建立在 Linux 集群的基础上的。

1.4 国内外研究现状

进程迁移一直是国内外非常活跃的研究课题。目前，国内外对进程迁移已经做了大量的工作，并在此基础上实现了相应的集群系统。如 MOSIX[13-14], Mach[15]，Charlott[16]，Amoeba[17]，V System[18]和 Sprite[19]等系统大都通过修改操作系统内核，通过内核原语访问进程状态，对进程迁移提供内核级支持。内核级支持的方法效率高，透明性好，但实现复杂，可移植性差。而 Condor[20-21]，CoCheck[22]和 ChaRM[23-24]等系统是依赖于原有底层操作系统，附加一定的控制软件来达到集群协同工作的效果，它们通常采用让应用程序在编译时链接相关的检查点库文件从而实现进程状态的恢复。这种方法无须修改操作系统，实现相对简单，但效率没有内核级支持方法高。

1.5 论文主要内容及组织结构

主要内容为：

以集群系统、进程迁移以及动态负载平衡为主要研究内容。研究了进程迁移相关理论和技术，并借助进程迁移实现集群系统的动态负载平衡。

通过对已有的经典进程迁移算法的分析，借助各算法的优点，提出一种改进的进程迁移算法，最大程度地将进程状态迁移和进程的运行并行起来，从而提高了迁移速度，网络通信量也较小，而且也没有对源节点的剩余依赖性。

通过对进程迁移机制的分析，设计和实现一个进程迁移系统，将进程从负载较重的节点移动到负载较轻的节点，从而改善集群系统的整体性能。

后续章节内容安排如下：

第 2 章是介绍进程迁移的相关理论以及相关工作，并对已有的经典进程迁移算法进行了分析。

第 3 章是进程迁移机制的分析与设计。本文给出了本系统负载监测模块的流程，以及“待迁移进程”和“目的主机”的选择机制，解决进程迁移由谁启动的问题，同时讨论和解决了迁移进程打开文件和通信的处理问题。最后，给出改进的进程迁移算法以及进程迁移机制和进程迁移系统的框架。

第 4 章是进程迁移系统的实现。主要介绍本系统中各模块的具体实现，并对关键函数进行了解析。

第 5 章是测试与结果分析。对改进的进程迁移算法与已有的迁移算法进行了性能比较和结果分析，同时将本文的进程迁移系统与已有系统做性能比较，证明本系统具有较高的性能优越性。

最后是总结与展望。总结全文并指出有待解决的问题。

第2章 进程迁移相关理论

2.1 什么是进程迁移

进程是操作系统中一个非常重要的概念。进程可认为是应用程序的运行实例，是应用程序的一次动态执行[25]。

进程是一个随执行过程不断变化的实体。和程序要包含指令和数据一样，进程也包含程序计数器和所有 CPU 寄存器的值，同时它的堆栈中存储着如子程序参数、返回地址以及变量之类的临时数据。当前的执行程序，或者说进程，包含着当前处理器中的活动状态。

进程迁移是将一个活跃进程从当前节点移动到指定的目的节点上，使其在断点处继续执行且行为与没有迁移前一样。要想在目的节点恢复进程的执行，就要获取足够的进程信息。迁移的进程信息为[26-28]：

(1)进程控制信息：包括了 OS 管理该进程的所有信息，如进程标识符信息（PID,PPID），进程当前工作目录。一旦进程控制信息被源节点冻结，进程将被挂起。

(2)进程执行状态：包括在进行上下文切换时有关的核心存储和恢复信息。如寄存器值、堆栈指针以及程序计数器。执行状态与机器密切相关。

(3)进程地址空间：包括属于该进程的所有虚存空间，这是进程状态信息中最大的部分，也是影响进程迁移算法性能的关键部分。

(4)消息：包括进程收发的缓存消息和关于通信连接的控制消息。迁移进程的消息处理主要包括迁移过程中的通信处理和迁移到目的节点后的通信处理。

(5)打开文件：包括文件内部标识，文件访问位置及缓冲的文件块。

2.2 进程迁移相关工作

2.2.1 非抢占式进程迁移和抢占式进程迁移

进程迁移可用于实现资源负载平衡，达到资源有效共享。负载平衡的实现机制有两种：非抢占式进程迁移和抢占式进程迁移。

非抢占式进程迁移也叫初始放置(Initial Placement)或远程执行(Remote Execution)，是当进程在某个计算节点产生时，根据负载平衡的要求将进程转移到其它节点远程执行的方法[29,30]。通常非抢占式进程迁移只需传送进程代码或进程环境，如打开的文件，不用传送大量的进程状态信息，因此其迁移速度要快于抢占式进程迁移。但是，非抢占性进程迁移也有其自身缺点。它缺少灵活性和对于进程迁移的透明性，进程只能在被创建时迁移，而且各节点随着进程的执行，其负载情况也在不断发生改变，如果这时能获得执行进程的相关信息用于衡量节点的负载情况,可能令系统产生更好的负载平衡策略。

抢占式进程迁移是在进程执行过程中，根据系统的负载情况，将进程从负载较重(Over-loaded)的节点转移到另一负载较轻(Under-loaded)的节点继续运行，从而有效得实现负载的动态平衡。迁移中传送信息的数量依赖于采用的进程迁移算法。通常传送的信息要远大于非抢占性进程迁移，因为进程迁移状态信息中包括进程的地址空间，它是影响迁移性能的关键部分。

抢占式进程迁移的优点是在进程开始运行后迁移，可用于评价整个系统的负载变化情况，并基于此制定更高效的平衡策略。虽然抢占式进程迁移比初始放置代价要高，但它的综合性能更为有效。Horchol－Balter 和 Downey 的研究认为，抢占式进程迁移能够减少平均延迟 35－50%[31]。前面提到的术语进程迁移及后面提及的均指抢占式进程迁移。

2.2.2 用户级进程迁移和内核级进程迁移

进程迁移可以应用到不同级别，而且级别的不同会影响其性能、容错和可重用性不同。已经实现的进程迁移系统大都在内核级或用户级，也有些在应用级实现。

用户级迁移无须改变底层操作系统，相对实现简单，软件开发和维护也比较容易。但由于内核空间和用户空间存在着壁垒，打破这个边界获得内核

提供的服务需要巨大的开销。请求内核服务是较慢且受限制的，因此无法获取内核的所有状态，这意味着迁移进程的类型会受到限制。尽管用户级进程迁移有较高的进程迁移开销，但由于其实现级别更容易获取应用程序的行为，这可用于制定更好的负载平衡策略。有很多用户级实现进程迁移的系统，如Condor[20-21]。

内核级进程迁移需要对底层操作系统进行修改和扩展，这导致其实现复杂，所以应用要比用户级实现困难。但另一方面，它可以直接和快速地访问关于迁移进程的内核信息，所以有高效平滑的性能。内核级进程迁移的另一优点为对于客户应用程序的透明性，应用程序无须考虑迁移而设计或被修改。

MOSIX 和 Sprite 是比较典型的内核级实现进程迁移的系统。

MOSIX 利用给内核增加补丁的方法全面兼容 Intel 架构的 32 位处理器，可以将负载准确迁移到集群成员，系统会自动或手动的将负载优化的分担给各节点[13-14]。实现负载均衡也是 MOSIX 设计的目的。准确的进程迁移使集群类似于一个巨大的 SMP 系统，多个节点就象多个处理器。同时，MOSIX 设计者致力实现快速网络协议，优化网络设备驱动，从进程和用户的角度维持同样的操作系统语义。即使进程迁移，信号语义仍然相同，IPC 通道如管道和 TCP/IP socket 仍可使用，甚至在“home node”（进程迁移前运行的节点）上执行 ps 程序，被迁移进程仍出现在结果中，这些都带来了很好的透明性。

2001 年底，MOSIX 项目决定不再遵循 GPL 许可，新的版本不再是免费的。所以原 MOSIX 项目的核心经理 Moshe 开始了一个新的项目 openMosix，保证 MOSIX 免费版本继续为公众服务。openMosix 代码的很多问题被修复，性能得到提高[34]。而且在增加很多新的特性的基础上，更广泛的、高性能的特性也在研发中，比如自动配置、新的用户工具、节点探测。但 openMosix 集群不能在同一时间在多个处理器上运行单个进程，不支持线程，也不支持共享内存，这样使有些软件在 openMosix 集群上应用，其性能得不到提高，如 mozilla、mySQL、Apache 等。

Sprite 开发于 UC Berkley 大学，是一个有分布式文件系统的网络操作系统[19]。其进程迁移在进程迁移系统中引入了 Sprite 文件服务器，这也是一个里程碑式的进步[33-36]。Sprite 采用 Flushing 算法，成为第一个克服对源节点有剩余依赖性问题的系统。当一个进程迁移时，它将缓存的文件块和进程地址

空间都拷贝到文件服务器上，除此之外的进程状态直接传送到目的主机。对于目的主机上恢复运行的进程没有的内存页或缓存的文件块，都向文件服务器请求。Sprite 运行在 10Mbps 以太网连接的 SPSRC station 工作站上。平均迁移时间是：对一个有 6 个打开文件的 100KB 的进程，其迁移时间为 330ms[36]。

Sprite 设计的动机是“主人优先原则”。当进程用 exec()被创建时，它有可能被迁移到一个空闲节点，但如果节点主人回来，它将被逐出再次迁移。

进程迁移也可以被实现在应用级，但由于其需要了解应用程序语义并可能需对应用程序进行修改或重编译，透明性较差，限制了其应用范围。应用级实现的系统主要有 Freedman[37]等。

2.2.3 同构进程迁移和异构进程迁移

同构进程迁移意味着进程是在分布式系统中同构的各节点间进行迁移。各节点具有兼容的体系结构和操作系统，但不必要具有相同的资源和能力。绝大部分支持进程迁移的系统都是同构进程迁移，如前面提到的 MOSIX 和 Sprite 系统。

异构进程迁移用以解决系统中节点间体系结构和操作系统不同情况下进程的迁移[38]。很明显，在异构环境中实现进程迁移要更加复杂，而且转换翻译的开销要影响其性能。体系结构和操作系统特性要充分考虑，此外，一个进程的状态应以一种机器无关的形式被传送和恢复。异构进程迁移尤其适用于移动环境下，因为其移动单元和基站可能由不同类型的机器组成，但在计算期间又可能需要将一个进程从移动单元迁移到基站，大多数情况下，这是同构进程迁移所不能解决的问题，但采用异构进程迁移就使其成为了可能。异构环境实现进程迁移的系统主要有 Emerald 和 TUI。

Emerald 是一种基于分布式对象模型 Eden 的面向对象的编程语言，Eden 模型用于支持异构环境下一个活跃对象的移动，每个对象对外都有统一的视图，并针对每种目标体系结构预先编译代码[39-40]。TUI 系统通过迁移 C 或 Pascal 语言编写的应用程序来改进 Emerald，因为 C 或者 Pascal 是比 Emerald 更流行应用的编程语言[41]。TUI 系统中迁移进程相对简单。首先程序针对每种目标体系结构预先编译，程序最初在源节点上执行，在迁移时，进程作检查点并将执行状态转换成中间形式写入文件，然后文件被发送到目的节点，在目

的节点针对其体系结构从中间形式重构进程的数据和状态，之后恢复进程的执行。当然，在具体实现时还有很多细节需要考虑，如程序计数器对于不同体系结构、程序的不同编译版本都可能是不同的，这将使程序不是在所有地方都可以停止作检查点操作等等。这些细节问题的考虑使异构进程迁移实现起来要复杂于同构进程迁移，并可能带来更大的迁移开销。

本文主要研究同构进程迁移相关问题。

2.3 已有的进程迁移算法

在介绍主要的进程迁移算法前，先解释几个术语。源节点指进程迁移前运行的主机；目的节点指迁移后进程在其上恢复执行的主机。前面已经介绍过要恢复进程的执行，就要获取足够的进程信息，这些必要的信息称作进程状态，包括执行状态、通讯状态、寄存器值以及其它一些操作系统相关数据。而一次完全的进程迁移需要进程所有的信息，进程信息由进程状态和进程的地址空间组成。剩余依赖性指当进程迁移后，有部分信息存留在源节点上，因此被迁移的进程仍依赖于以前的节点。一旦源节点失效或网络连接失败，那么被迁移的进程将会因为丢失信息而运行失败。

2.3.1 Total-Copy 算法

Total-Copy 算法是最常被使用的算法，它非常简单且第一个被提出。Amoeba[15]和 Charlotte[16,42]都是实现 Total-Copy 算法的系统。

Total-Copy 算法的主要步骤为：

(1)在源节点悬挂起被迁移进程

(2)传送整个进程信息

(3)在目的节点恢复被迁移进程的运行

(4)从源节点除去被迁移进程

该算法的优点是概念上简单，相对容易实现。一次性传送所有进程信息到目的节点，消除了剩余依赖性，而且成功传送后的内存空间可以立即释放。然而，由于传送所有信息造成延时较长，而这段时间不能接受消息增加了通信失败的机会，这对实时系统是不可接受的。

2.3.2 Demand Page 算法

Demand Page 算法只能被应用于支持远程调页的情况下[43-44]。与Total-Copy 算法相比，它有不同的请求调页策略。一种是基于复制的引用(Copy-on-Reference)，它只传送最少必要进程状态，而另一种 Eager Dirty 在最初迁移时间内还传送进程的所有被修改的脏页，两种策略都对源节点请求进程剩余信息。

Demand Page 算法的大概步骤为：

(1) 在源节点悬挂起被迁移进程

(2) 迁移最少必要的进程状态

(3) 在目的节点恢复被迁移进程的运行

(4) 在进程执行期间向源节点请求需要信息

一旦被迁移进程恢复执行，它可能引用的地址空间页仍存留在源节点上，因为没有迁移整个地址空间，为了解决缺页，进程向源节点发送请求，然后立即返回需求页。在请求调页期间，进程也被挂起。这样的延时极可能大于本地解决缺页的延时并且增加进程运行时间的开销。准确的延时依赖于源节点和目的节点间网络连接的特性。

Demand Page 算法挂起被迁移进程后只迁移最少必要信息，极大地减少了传输数据量，此外，该算法也揭示了进程在执行期间通常只使用一小部分地址空间，在迁移后不被使用的地址空间，将不需要传送。

Demand Page 算法主要的缺点是源节点在被迁移进程运行结束前必须维护进程地址空间的信息，这种长期的剩余依赖性是个问题。如果一个进程被迁移多次，那么为了解决缺页将不得不对进程存留过的主机作链式搜索以获取该页信息。然而，一旦源节点失效，进程将不能解决缺页从而导致运行失败。这种剩余依赖性降低了目的节点的容错性。

Accent[43]系统是第一个使用基于复制的引用策略实现 Demand Page 算法的系统，RHODOS[45]和 Mach[15]也实现了该算法。

2.3.3 Flushing 算法

Flushing 算法是第一个在进程迁移中引入第三个实体的算法。它在 Sprite 系统中被提出，除了使用源节点和目的节点外，还使用了一个文件服务器。

Sprite 的进程迁移机制的目标是达到 Demand Paging 算法的高效同时避免剩余依赖性。

Flushing 算法需要操作系统支持虚拟内存技术。在 Sprite 中，使用普通文件实现虚拟内存的备份存储。这些备份文件存储于网络文件服务器并可从网络中任意访问。

Flushing 算法的主要步骤为：

(1)在源节点停止被迁移进程的执行

(2)刷新所有脏页到网络文件服务器

(3)迁移进程状态(进程控制和执行状态、通信连接信息和被缓存的消息、文件描述符）到目的节点

(4)在目的节点恢复进程的执行

(5)进程向网络文件服务器请求解决所有缺页

为使文件服务器高效解决缺页，需要对其进行高度优化。Sprite 采用内存和文件相同的访问机制提供一个基于文件系统的通讯，这可达到快速的文件服务器访问，并加速请求页的传送速度以更快解决缺页。

2.3.4 Pre-Copy 算法

Pre-Copy 算法首先在 V System 中提出用以克服 Total-Copy 算法一次性传送进程信息造成的长时间开销[18]。与以前描述算法不同的是，Pre-Copy 算法直到大部分进程地址空间传送到目的节点后才挂起源节点的进程，也就是说源节点的被迁移进程运行和地址空间的传送并行[46,47]。但在进程的执行期间，进程又会修改其内存页，因此，该算法继续传输被修改的内存页到目的节点，直到修改页的数量足够低，达到一定的阈值，然后在源节点上挂起被迁移进程，传输进程的状态和剩余的修改页。

Pre-Copy 算法的主要步骤为：

(1)传送进程整个地址空间，同时进程继续在源节点执行

(2)检查修改页的数量是否低于阈值，如果没有，则继续传送被修改页

(3)挂起被迁移进程并传送进程状态

(4)最后，传送所有剩余修改页

(5)在目的节点恢复进程的执行

Pre-Copy 算法同样减少了进程的挂起时间，因为它只在修改页数量低于一定阈值时才挂起进程，传输进程状态和剩余修改页到目的节点，同时在传输完毕后，也消除了剩余依赖性，提高了系统可靠性。但这种算法依赖进程内存访问方式，由于在预备时间内，内存页被修改，所以有些页需要被传输多次。如果进程修改太多的内存页，则预先设定的阈值将永远不会达到从而导致算法失败。

2.3.4 进程迁移算法分析

通过对已有算法的简单描述，可以发现这些进程迁移算法都需要实现以下几步：

(1) 决定要迁移的进程

(2) 在源节点挂起被迁移的进程

(3) 传输进程状态到目的节点

(4) 在目的节点重构进程状态

(5) 在目的节点恢复进程的执行

(6) 从源节点移动进程的残留信息（不是必需的）

Total-Copy、Demand Page 和 Pre-Copy 算法只有源节点和目的节点参与了迁移过程，而 Flushing 算法除了源节点和目的节点外，还有文件服务器作为第三方节点，参与迁移过程。在只有源节点和目的节点参与的算法中，可以用三个条件区分迁移算法，这三个条件是：从源节点传输至目的节点的进程状态量；挂起源节点上被迁移进程的时间；在目的节点上恢复迁移进程执行的时间。这三个条件是进程迁移算法需要考虑的最基本问题，也是区分进程迁移算法的有效条件。

表 2.1 进程迁移条件

	传送状态信息	挂起进程	恢复进程执行
1	全部	决定进程迁移时	传输完全部进程状态时
2	部分	决定进程迁移时	传输完最少必需状态时
3	全部	进程状态传输完成时	传输完全部进程状态时
4	全部	决定进程迁移时	传输完最少必需状态时

对于以上条件的取值都有两个。在迁移时，可以传送“全部”进程状态信息，也可以传送“部分”信息；对于待迁移进程的挂起，可取“决定进程迁移时”或“进程状态传输完成时”；而恢复进程的执行则可取“传输完全部进程状态时”或“传输完最少所需状态时”。根据这三个条件值的不同组合，可以形成不同的算法，如表 2-1 所示。

三个条件值的组合可有八种算法，但除了表 2-1 中的 4 种算法外，其余 4 种算法都是不合理的。Total-Copy 算法对应于表 2-1 中的第 1 种组合，Demand Page 算法对应于第 2 种组合，Pre-Copy 算法对应于第 3 种组合，而本文提出的算法参照第 4 种组合。

Total-Copy、Demand Page、Pre-Copy 和 Flushing 是目前经典的几种算法，已经被应用于不同的场合。如果进程在目的节点上并不需要使用很多它的地址空间，则后三种算法较好。另一方面，如果在目的节点上最终将访问大部分地址空间，则地址空间的碎块式转移效率低，不如在迁移时简单地将所有地址空间都转移，可以使用 Total-Copy 和 Pre-Copy 算法之一。在很多情况下中，不可能预先知道是否需要大量的非常驻地址空间，然而，如果进程是由线程构成的，并且如果迁移的基本单位是线程而不是进程，则基于远程页式管理的 Demand Page 算法是最佳的。

虽然经典算法已被广泛应用，但它们还有一些缺点。Total-Copy 算法由于传送进程全部状态信息，其迁移时延依赖于进程状态量的大小，对于实时进程和有大量 I/O 的进程，此算法进程迁移时延是不允许的。Demand Page 算法具有最低的进程迁移初始代价，但其最大的缺点是对源节点有剩余依赖性，并存在一定的进程迁移时延。Pre-Copy 算法减少了进程被冻结的时间，但由于某些页面会被传送多次，这样就增加了系统的通信传输量，有可能进程迁移时延会比 Total-Copy 算法的还长，但此算法对源节点没有剩余依赖性。Flushing 算法没有对源节点的剩余依赖性，进程迁移时延也较短，但它的瓶颈在于如果文件服务器一旦失效，那么正在迁移的进程都会受到影响，而且迁移进程的某些页面会在系统中被传送两次（先传送至文件服务器，再由文件服务器传送至目的节点），增加了系统通信量。

通过上述比较，针对进程迁移时延太长，存在对于别的节点的剩余依赖性，以及偶尔发生网络拥塞等方面问题，本文提出改进。

2.4 本章小结

本章择要介绍了什么是进程迁移、进程迁移状态信息及进程迁移相关工作，并描述了现有的四种经典迁移算法。通过对这些算法的分析，总结出迁移算法的基本步骤和区分迁移算法的有效条件，并提出算法可进一步改进之处。

第3章 进程迁移机制的分析和设计

前面已经介绍了进程迁移的相关理论，本章主要讨论进程迁移机制的设计目标及需要考虑的相关问题，并提出改进的进程迁移算法，以及进程迁移机制和进程迁移系统的框架。

3.1 进程迁移机制的设计目标

进程迁移机制的设计目标包括许多且并非完全统一，因此每个系统针对不同的要求，需要在它们之间进行权衡协调，以满足不同的系统环境。这些设计目标主要为[48-49]：

(1)透明性：进程迁移的透明性指一个进程执行的行为及结果不应受执行位置的影响，应该是位置无关的，它包括对用户的透明性、对系统的透明性、对进程本身的透明性。在进程迁移中，透明性是绝大多数系统共同追求的首要目标。

(2)避免剩余依赖性：与位置有关的操作往往依赖于源节点，这种关系称为对源节点的剩余依赖性，剩余依赖性不仅影响了源节点和目的节点的性能，而且已经被迁移进程的运行仍要受到源节点故障的阻碍，不能满足可靠性的要求。然而，当前大部分系统设计中仍然存在剩余依赖性。这是由于在进程迁移的机制中要完全消除剩余依赖性是不现实的。为了维持进程迁移的透明性，有时剩余依赖性也是必不可少的。

(3)可伸缩性：对于工作站环境下进程迁移的设计，可伸缩性是一个内在必然的要求。因为在大多数系统中，实现迁移主要是为了有效地利用系统资源，提高效率，故而“伸”是必然的；而同时，由于主人优先原则，在主人回来时，要及时将外来进程撤出，“缩”也称为必然。随着工作站集群计算的不断发展，迁移的可伸缩性要求将会越来越受到重视。

其它设计目标还包括异构性、容错性、并行性、效率，以及结构清晰、容易使用、不需修改原应用等。在进程迁移中，这些因素都应全面考虑，并根据不同的系统环境和应用要求进行协调统一，选择不同的侧重点。

本文进程迁移机制的设计目标主要考虑实现复杂性、透明性、避免剩余依赖性以及独立性。

(1)实现复杂性。为提高进程迁移的效率和追求用户透明性，本文选择内核级支持进程迁移，但这又增加了实现的复杂性。为将复杂性降到最低，对Linux 核心修改尽可能最少。

(2)透明性。为实现对用户高度透明，需要解决进程远程执行中的相关问题，如对打开文件的操作和消息通信等。

(3)避免剩余依赖性。通过全部传送进程信息到目的节点，消除对源节点的剩余依赖性。剩余依赖性只存在于信息传送阶段。

(4)独立性。进程迁移机制应适用不同的负载信息收集发布策略和调度策略，使其不依赖于负载信息管理的设计与实现。

3.2 进程迁移的启动

由谁来启动迁移将取决于迁移机制的目标。本文设计进程迁移功能主要为实现集群系统的动态负载均衡，因此由操作系统中监视负载情况的负载监测模块决定何时进行迁移，该模块应告知需要迁移并确定迁移哪个及迁移到哪里，同时该模块需要与其他节点上对等的模块进行通信，使其他节点上的负载情况可以得到监视。

负载监测模块设计时应考虑的问题：

(1)负载信息的表示

负载信息可以从多个方面进行衡量[50]：

a) 使用资源利用率作为负载信息。负载平衡的目的是缩短任务响应时间及提高集群系统的资源利用率，所以可以从资源入手选择负载向量。通常使用的资源利用率负载指标有 CPU、内存及 I/O 等。

b)使用队列长度。队列长度包括 CPU 队列长度、I/O 队列长度等。通常 CPU 队列长度被认为是当前最优的负载平衡指标，其参数容易获取，用于收集和处理该信息的开销小，且容易建立数学模型进行分析。

c)使用综合策略。综合策略一般指同时考虑多项负载指标，以多种简单的负载向量为基准，通过加权计算最后得出一个负载的综合值。

(2)负载信息收集范围

负载信息收集可以是全局信息收集，也可是局部信息收集。全局信息收集指各个节点每次通过广播方式向其余节点广播自身的负载信息。局部信息收集指各个节点每次只与部分节点交换负载信息。

(3)负载信息的收集

对负载信息的收集可以是周期性的，也可以是由事件驱动的。周期性收集是以一定时间间隔收集负载信息，而事件驱动方式通常指进程的创建、结束或迁移时收集负载信息。

(4)负载信息的发布

负载信息的发布同样可以是周期性的，也可以是由事件驱动的。周期性发布是节点本身周期性地向集群系统范围内的其他节点散步自身的负载信息。事件驱动方式通常指状态变化驱动，系统负载情况随着时间而变化，节点仅在负载状况发生变化时，才向其他节点散布最新的状态信息。

(5)收集的负载信息的管理

收集的负载信息管理可以采用集中式管理或分布式管理。集中式管理指系统中只有中央控制节点保存整个系统的负载信息。当某个节点需要启动进程迁移时，需要向中央节点发生申请，负载均衡的决策由中央节点做出。分布式管理指系统中各节点地位平等，每个节点都保存整个系统的信息，当需要发生进程迁移时，可以根据本地保存的信息选择目的节点。

本文中采取综合策略来表示负载信息。主要考虑采用二个负载指标：① CPU 运行队列的长度 A1，权值为 B1；② CPU 的物理速度 A2，权值为 B2；通过将二个指标的加权乘机求和来表示负载信息，则负载信息为：

$$F(hostload) = A1B1 + A2B2 \tag{3-1}$$

对于负载信息收集和发布，本文采取周期性收集负载信息，状态变化驱动方式发布负载信息的全局负载信息收集模式，同时对收集的负载信息选择分布式管理。即集群系统中各节点都保持一个负载信息链表(loadofhostlist)，其中链表的元素是系统中各节点的负载信息。以每秒为时间间隔对本机做负载信息收集和计算，更新本机负载信息，同时判断是否达到状态阈值，是否需要向其他节点广播自身负载信息。

负载监测模块是以周期性信息的收集作为驱动的。模块中有一个

info_daemon 定期收集本机负载信息并通过判断发送给其他节点，还需监听约定的端口，以接收其他节点的负载信息，同时每次负载信息收集和重新计算后要判断是否超载，是否需要触发进程迁移模块。通过检查负载信息链表，可知整个系统的负载情况，只要系统中很繁忙主机不超过 70%[61]，就可以准备进行进程迁移。如果在系统中多数主机都很繁忙的情况下进行进程迁移，是非常得不偿失的。

负载监测模块的流程为图 3.1：

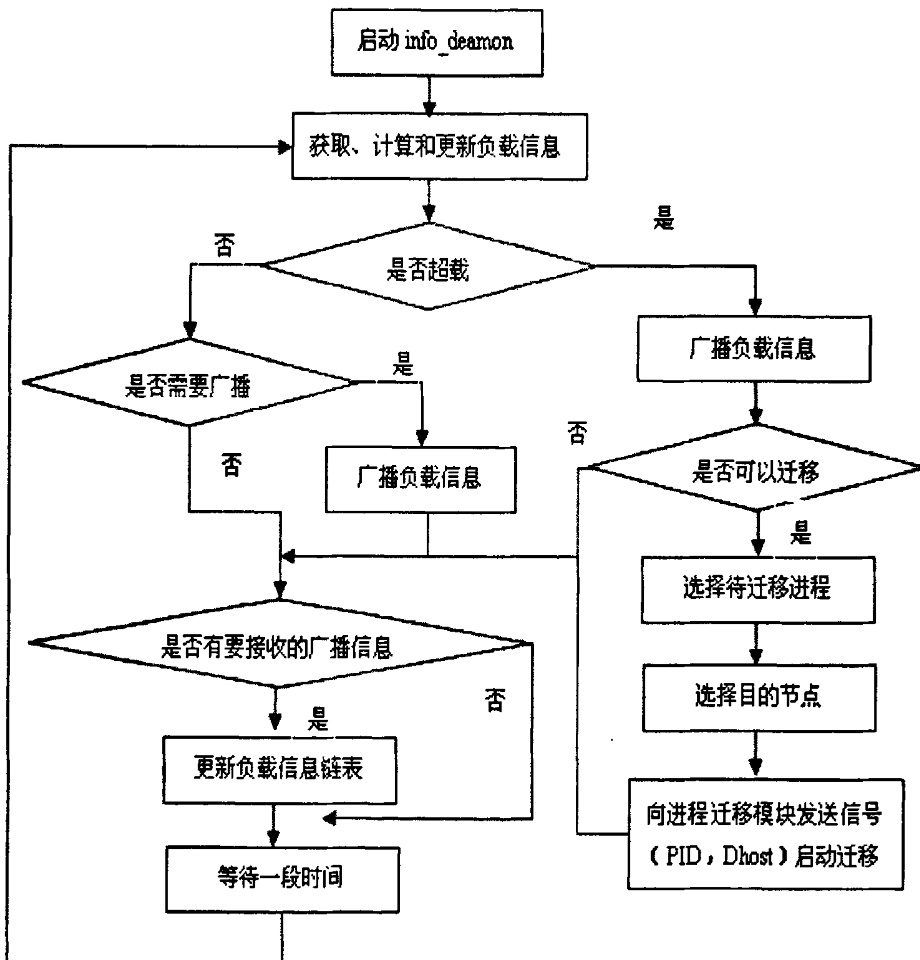

图 3.1 负载监测模块流程

由于是动态进程迁移，对于待迁移进程的选择，倾向于迁移已经运行时

间短而剩余的执行时间还很长的进程，因为对它们做动态迁移所需移动的信息量较小[62]。进程当前的生存时间可以用于负载平衡的目的，在判断进程是否值得被迁移之前要判断进程应该有多“老”。对于那些生存期很短的进程而言，迁移的弊端超过了利益[53]。此外，访问设备和文件越少的进程，越便于迁移。同时，因为本机 Linux 运行必须的系统进程和迁移系统服务进程是不能被迁移的，所以要设法区分迁移系统中允许的用户进程和原有的系统进程，可以使用 uid 来区分，只有本系统专用用户帐号的进程方可迁移。

对于目的节点的选择，前面提到每个节点都保持一个负载信息链表，其中保存了整个系统中所有节点的负载信息（自上一次更新以来），于是每个节点在需要进程迁移时，都能结合各个节点的负载情况和进程迁移的代价来独立选择恰当的目的节点。

查找比较获得目的节点后，将待迁移进程的 PID 和目的节点标识传送给进程迁移模块，启动进程迁移。

3.3 迁移进程文件和通信的处理

3.3.1 打开文件的处理

进程迁移到其他节点上运行，有可能由于主机状态的改变而导致一系列复杂的问题。在进程迁移前，如果源节点上已经打开了一些文件，那么对这些文件的处理就比较复杂，而且进程打开文件的状态是产生进程剩余依赖性的一个主要方面。

对于进程打开文件的处理通常有：

(1)将这些文件的状态及文件随迁移的进程一起拷贝到目的节点，然后用正常的机制对这些文件进行操作，这是比较直观的方法，但开销太大，一般系统都不采用。

(2)利用对源节点的剩余依赖性，在目的节点上拦截访问已打开文件的指令，将其返回到源节点进行操作，结果再送回目的节点。这种方法最简单，但是效率低，开销大，当迁移进程的文件操作较为频繁时，将严重降低源节点和目的节点的效率。

(3)利用全网范围的共享文件系统。在转移过程中，只需将打开文件的状

态转移到目的节点上，即可按正常方式进行操作。当前，大多数系统都实现这种方法。

为了提供透明的迁移，应该为迁移前后的进程提供统一的文件系统观点。为此，本文利用全网范围的共享文件系统方式，借助分布式文件系统 NFS 的远程安装功能，使所有节点获得看上去一致的文件系统结构。

在进程迁移模块中添加自定义的系统调用 getfileinfo() 方法一次性获取需要保存的全部文件状态：文件描述符值 fd、文件路径 path、文件名 filename，文件打开模式 mode 和文件打开偏移量 ops。

Linux 系统中，进程描述符 task_struct 结构中 fs 字段指向 fs_struct 结构，内核用此结构来维护进程与文件系统之间的联系。进程描述符中的 files 字段指向 files_struct 结构，内核用此结构表示进程当前打开的文件。在 files_struct 结构中，fd 字段指向以 file 结构为元素的指针数组，文件描述符值是数组的下标。每一个 file 结构对应一个打开的文件，通过它可获取需要保存的文件偏移量和打开模式等状态。对于文件路径，file 结构的字段 f_dentry 指向 dentry 结构，依据其在内核中的链表结构，循序读取出绝对路径名。将这些状态通过消息发送到目的节点。

在目的节点恢复进程的执行时，根据收到消息中的文件路径全名 path 和打开模式 mode 重新打开文件，用保存的偏移量 pos 重新设置文件的读写指针，即可实现打开文件对应的内核数据结构的重新建立。但此时内核中再次打开文件的文件描述符 fd 与迁移前是不一样的，故需要对文件描述符进行重定向，将保存的文件描述符 fd 重定向到新打开文件后内核中建立的 file 结构，紧接着关闭新打开文件的文件描述符。实现时可借助 dup2() 函数，复制一个文件描述符，使新描述符和旧描述符特性一样。文件描述符重定向实现代码为：

```
int open_force (int fd, char *filename, int flags, int mode)
{
        int ret = open(filename, flags, mode);
        if(ret<0 || ret = =fd)   return ret;
        if (dup2 (ret,fd) <0 )   return -1;
        close (ret);
```

```
        return fd;
    }
```

这里处理的文件都是普通的资源文件，对于管道这类特殊文件，由于NFS并不支持管道，因此本系统中没有解决管道的迁移问题。

3.3.2 进程通信的处理

迁移进程通信的管理主要涉及两个阶段：迁移过程中的通信和进程迁移之后的通信。

(1)迁移过程中的通信处理主要有以下几种方法：

a) 在迁移过程前，迁移进程通知其它进程不再发送任何消息给该进程，即迁移进程不与其它任何进程通信。该方法显然不适合实时系统。

b) 在迁移过程中，源节点收到消息后转发给目的节点。那么迁移进程对源节点存在剩余依赖性。一旦源节点发生故障，迁移的过程将会受到影响。

c) 在迁移过程中，源节点拒绝接收发给迁移进程的消息，并要求发送者稍后将消息重发到目的节点。

(2)处理进程迁移后通信的方法主要有：

a) 消息重定位方法：当进程挂起要迁移时，源节点已经获得目的节点地址，进程迁移后，不通知与之通信的进程，消息仍然发到源节点，由源节点转发到目的节点。

b) 消息丢失保护方法：由源节点通知迁移进程的新地址，消息先发送到新节点暂存，待迁移进程恢复执行时，从本地缓存中取得消息。

d) 消息丢失恢复方法：在迁移过程中发送的消息都将丢失，待迁移完毕后，重新建立连接并重新发送消息。

本文系统中对消息的处理，采取在源节点提供一个 dummy 进程，每当有进程需要迁移时，就伪装成该进程来接收发给该进程的消息。由于dummy进程接替了原进程的工作，所以在迁移过程中，不需要冻结消息的接收。dummy 进程把迁移期间收到的消息转发给目的节点上的进程，同时负责广播通信连接的更新信息，此后所有要给该进程发送消息的进程都知道该进程现在位于目的节点，不会再向源节点的 dummy 进程发送消息。

3.4 改进的进程迁移算法

前面已对经典的进程迁移算法进行了简单描述，本文中结合各算法的优点提出一种改进的进程迁移算法。

主要考虑利用 Demand Page 算法进程挂起时间最短的优点，在发起进程迁移时，先挂起待迁移的进程，传送进程的必要信息到目的节点，然后目的节点重构进程，并恢复执行。但 Demand Page 算法对源节点有较强的剩余依赖性，因为源节点必须维护进程的地址空间用于请求时传输。在考虑到 Flushing 算法是将进程地址空间传送到文件服务器，在请求缺页时向文件服务器申请，消除对源节点的依赖。改进算法可以让源节点直接将地址空间传送到目的节点，不经过文件服务器，在发生缺页时，直接向源节点请求调页，一旦整个地址空间传送到目的节点，将彻底消除对源节点的剩余依赖性。同时，该算法中引入一个 dummy 进程，负责进程迁移通信的处理。

改进算法的流程描述如下：

源节点首先让 dummy 伪装成要迁移的进程，接收其它进程发给该进程的消息，挂起要迁移的进程；然后源节点将进程的控制和运行状态信息整理为一条消息，发送给目的节点。同时，dummy 进程将此前接收到的消息发送给目的节点上的进程，之后广播通信连接的更新消息。源节点继续传送文件信息消息；目的节点根据传送的状态信息重构进程并恢复执行。由于目的节点恢复进程执行时其地址空间为空，会立即产生缺页，所以向源节点请求调页，源节点先响应请求并传送请求页，之后继续传送进程其他地址空间信息。这时源节点的后续迁移工作和目的节点上的进程运行也就并行起来。目的节点上的进程收到源节点 dummy 进程转发过来的所有消息，恢复其消息处理。目的节点在恢复进程的执行后，继续在后台接收源节点发送的其它进程信息，包括进程地址空间的页面。在进程运行产生缺页时，就向源节点申请调页。

改进算法数据交换流程描述如图 3.2：

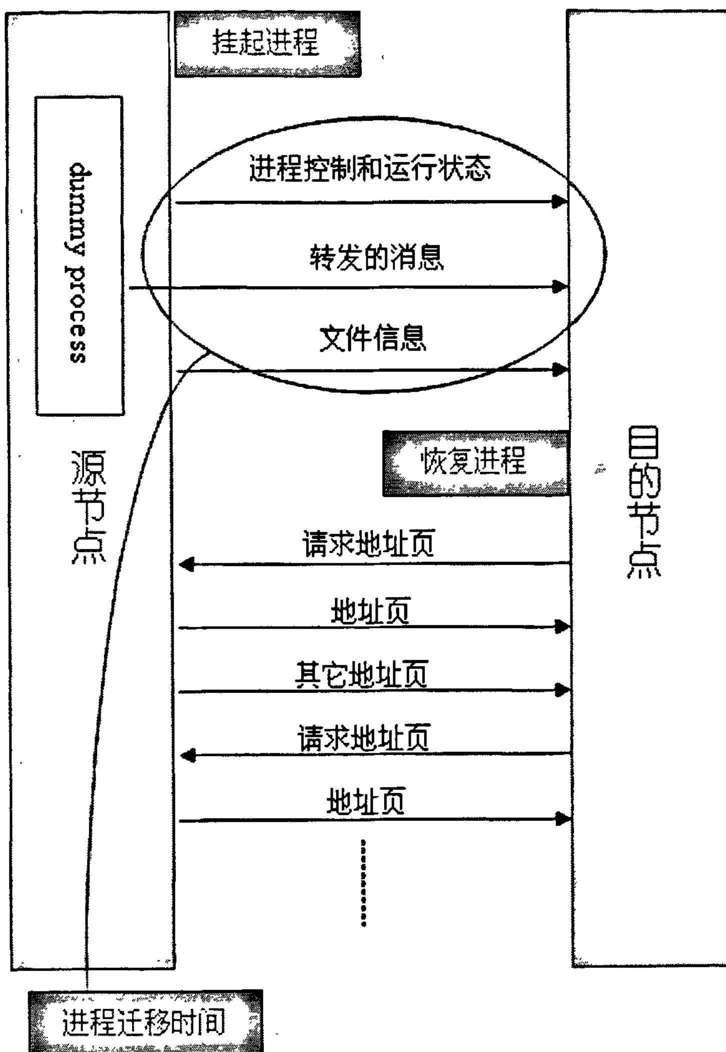

图 3.2 改进的进程迁移算法数据交换流程

改进的算法相对于以往的算法(只与没有第三方机器参与的基本算法进行比较)来说，是有许多优越性的。可以从以下几个方面进行比较：

(1) 从“决定进程迁移”到“在目的节点恢复进程执行”之间的时延 Delay1

(2) 进程迁移的总时延 Delay2

(3) 剩余依赖性

(4) 网络通信量

在对 Delay1 的比较中，改进算法的 Delay1 与 Demand Page 算法的都较小，其他算法的 Delay1 较大。因为改进算法与 Demand Page 算法都只需要传输完最少必须状态，便可以在目的节点上恢复被迁移进程的执行。

“进程迁移后的完成时间”与“如果不进行进程迁移的完成时间”相减得出的时间差是 Delay2。改进算法的 Delay2 稍大于 Pre-Copy 算法的 Delay2，因为在 Pre-Copy 算法的进程执行过程中不会发生“缺页”，但当产生的“脏页”较多，也就是当需要多次传送的页面较多时，Pre-Copy 算法的 Delay2 便会急剧增大。而改进算法的 Delay2 主要来自于，进程在目的节点上执行后，当请求的页面还未按顺序传送至目的节点时，发生的缺页传送会产生时延。虽然在 Demand Page 算法中，不必传送所有的信息到目的节点，但在迁移后的进程执行过程中发生的每次缺页传送，都会产生时延。而在改进算法中，一些页面在进程发生“缺页”之前，便会被传送至目的节点，在所有页面传送到目的节点后，缺页将由本地解决，不会再有传送时延。所以改进算法的 Delay2 比 Total-Copy 算法和 Demand Page 算法的 Delay2 都小。

虽然 Demand Page 算法具有较小的 Delay1 和 Delay2，但它在剩余依赖性方面，却有着不容忽视的缺陷，而 Total-Copy 算法、Pre-Copy 算法和改进算法都没有对源节点的剩余依赖性。

改进算法的网络通信量与 Demand Page 算法的相比，显然要大，它比 Total-Copy 算法的网络通信量也要大。因为在 Total-Copy 算法中，会一次性传送全部进程状态到目的节点，而在改进算法中，不但要传送所有的进程状态信息到目的节点，当进程发生“缺页”时，还要向源节点发送“请求页面”信息。但改进算法的网络通信量要小于 Pre-Copy 算法的网络通信量，因为 Pre-Copy 算法会有“多次页面传输”的附加通信量。

综合以上的比较，可以得出改进算法有较短的进程挂起时间，在源节点上增加一个 dummy 进程伪装进程接收消息，避免消息的冻结，同时在某种程度上将程序的运行和地址空间的传送并行起来，可以减少进程迁移后缺页的发生次数，从而减少进程迁移的总延时。而且没有借助文件服务器，令源节点直接传送地址空间，传送完毕后，消除了对源节点的剩余依赖性。

3.5 进程迁移机制框架

一次进程迁移操作的必需步骤是将被迁移进程的状态从源节点传送到目的节点，因此需要发送和接收进程状态的功能模块；进程迁移的启动由负载监测模块控制，因此需要实现进程迁移的功能模块提供一个调用接口。将进程迁出模块作为发送进程状态的功能模块，而监听迁入模块作为接收状态的功能模块。

进程迁移机制框架为图 3.3：

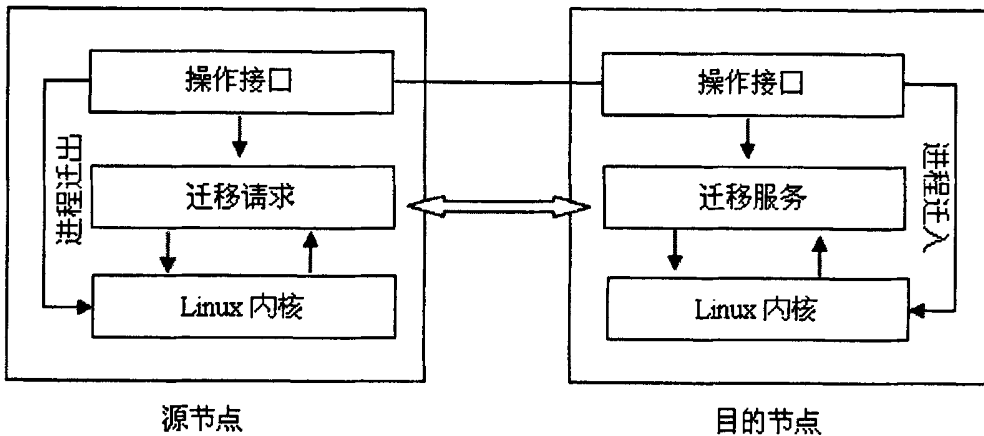

图 3.3 进程迁移机制框架

进程迁出模块启动后进入睡眠状态，被触发信号唤醒后启动 dummy 进程缓存收到的消息，自身向目的节点的监听迁入模块发送连接请求，作为实现迁移的申请客户端，提取进程状态，在连接建立后，发送进程状态消息，以及后续的进程迁移信息。

监听迁入模块是一个多线程服务器，由一个主线程等待在预先定义好的端口上，监听是否有进程迁移的连接请求，如收到请求就交给线程池中的一个线程处理，主线程又返回继续接收新请求。子线程主要根据收到消息调用相应的处理程序，重构进程并令其恢复运行。

3.6 进程迁移系统整体框架

进程迁移系统主要由负载监测模块、进程迁出模块和监听迁入模块组成。

进程迁移系统框架为图 3.4：

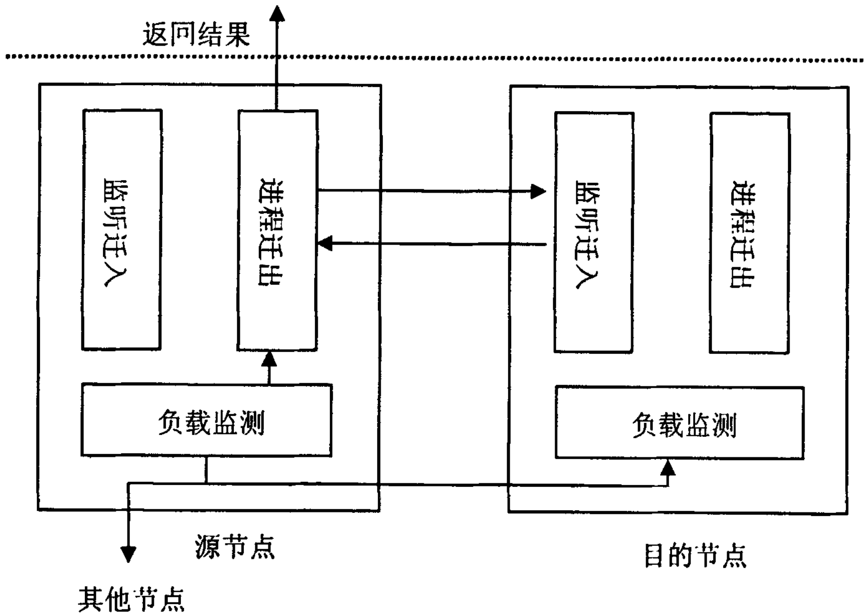

图 3.4 进程迁移系统框架

虚线上方表示迁移系统外用户并不知道进程迁移的存在，也就是进程迁移对于用户来说是透明的。整个系统是出于动态负载均衡考虑，只定期监测本节点是否超载，在超载时迁移动态运行的进程，没有考虑进程初始放置策略。

进程迁移系统的执行过程为：

(1)负载监测模块判断本机负载，如果本机负载过重，就选择待迁移进程和目的节点，并通知进程迁出模块移走进程。

(2)进程迁出模块从负载监测模块获取待迁移进程和目的节点，向目的节点的监听迁入模块发送迁移请求。

(3)进程迁出模块开始向目的节点的监听迁入模块发送迁移进程的控制信息、执行信息、缓存的消息和文件信息。

(4)目的节点的监听迁入模块根据第 3 步中源节点发送来的信息，在目的节点上重构此进程，并让它在目的节点上继续执行。

(5)源节点的进程迁出模块继续向目的节点的监听迁入模块发送迁移进

程的剩余地址空间信息。在此过程中如果收到目的节点发来的页面请求信息，便优先传送此页面。

(6)目的节点的监听迁入模块用源节点发来的消息，建立进程的剩余地址空间。

在上述过程中，每个节点的负载监测模块一方面会定时计算本机负载信息，并报告给其他节点；另一方面，接收其他节点的负载信息，更新自己的负载信息链表。

3.7 本章小结

在本章中，首先讨论了进程迁移启动问题，对启动迁移的负载监测模块流程进行了阐述，回答了“何时（迁移时机）由谁（哪个节点作为源节点）选择谁（哪个进程）向谁（哪个节点作为目的节点）迁移？”这个问题。同时，对迁移进程文件和通信的处理提出解决方案。借助已有进程迁移算法的优点提出了一种改进的进程迁移算法，改进算法与以往的四种经典算法的比较中，都有着较好的表现。

本章还设计了进程迁移机制和进程迁移系统的框架。

第4章 进程迁移系统的实现

本章将在 Linux 操作系统上实现文中设计的进程迁移系统。分别阐述每个功能模块的实现，并对关键函数进行解析，同时说明如何在 Linux 内核添加功能。

4.1 负载监测模块的实现

负载监测模块在系统中主要实现监测负载以启动进程迁移的功能。前面 3.2 节中已对负载监测模块的流程作了描述，这里主要说明实现过程中重要的数据结构和关键函数。

loadinfomation 结构——表示节点的负载信息，其中也包括用于广播时需要的信息，如本节点地址等。

loadofhostlist 表——负载信息链表。表中的每个条目包含 loadinfomation 结构，用于维护整个系统各节点的负载情况。

hosts 表——系统中已有主机的列表，这个列表在每台主机启动时从配置文件中取得，并且可以在运行过程中被修改，各主机只与本表中列出的主机交互。

在负载监测模块的实现中，较关键的函数有三个，它们分别是 Hostload_caculate 函数——负载信息的计算，Select_MigProcess 函数——待迁移进程的选择以及 Select_desthost 函数——目标主机的选择，下面分别说明这三个函数的实现。

本文采用 CPU 运行队列的长度(runqueue)和 CPU 物理速度(cpuspeed)的加权乘积来作为负载信息。Hostload_calculate()实现如下：

```
void Hostload_calculate(void)
{
    static struct sysinfo sysloadinfo;
    static float runqueue＝0.0;
    oldrunqueue＝runqueue;
```

```
oldavail＝loadinfomation.avail; /*保存旧的负载信息供
                                    比较用*/
sysinfo(&sysloadinfo)             /*Linux 内核的编程接口，
                                    可取系统统计信息*/
runqueue＝(float)sysloadinfo.procs;
/*返回 sysinfo 结构中 procs，即 CPU 运行队列当前进程数*/
              ……
 if (completed_nr>0)
 runqueue-=(*completed_nr); /*若从上次更新负载信息以来，
                              有替其他主机新完成的任务
                              (completed_nr 个进程)，则
                              sysinfo 返回的 procs 值应该扣
                              除掉 completed_nr */
if (runqueue<0.0)
   runqueue＝1;
runqueue*=cpuspeed;           /*我们用 CPU 频率的倒数来
                                拟定 cpuspeed*/
loadinfomation.loadvalue=runqueue;
}
```

在该函数统计完负载信息后，还需要判断是否需要广播，如需要则通过下面的代码广播负载信息。

```
int retvalue＝sendto(broadsock,&loadinfomation,0,
(struct sockaddr*)&broadaddr, sizeof(broadaddr));
```

在需要启动进程迁移时，待迁移进程的选择由 Select_MigProcess 函数实现，主要代码为：

```
int Select_MigProcess(void)
{
     ……
  int num;
  int weight[NR_TASKS-1000];       /*由每个用户进程的权重组成的
```

```
                                              数组*/
    struct task_struct *p;
    int tempweight, tempcount;
    for (index=0;index<NR_TASKS-1000;index++)
     weight [index]=0;
    for_each_task(p)
  {       ……
       if (p->times.tms_utime+p->times.tms_stime<temptime)
     {
        temptime=temptask->times.tms_utime+temptask->times.tms_stime;
        weight[p->pid]++;                    /*进程较新，权重增加*/
        if (p->files->count<tempcount)
       {
            tempcount=p->files->count;
            weight [p->pid]++;               /*与进程关联的文件少，
                                               权重增加*/
        }
     }
  }
    for (c=0;tempweight=weight[0];c<(NR_TASKS-1000);c++)
     {
            if (weight[c]>tempweight)
          {    tmpweight=weight[c];         /*找出权重最大的进程*/
               num=c;
          }
      }
          return number;                     /*返回所选中进程的pid*/
}
```

前面提到，每个节点保持有一个 loadofhostlist 链表，其中保存了整个系统中所有节点的负载信息（自上一次更新以来），于是每个主机在需要做进程

迁移时，都能独立选择恰当的目标主机。见函数 Select_desthost：

```
struct hostent * Select_desthost(void)
{     float tempcost;
      struct load_metrics *temphost;
      ......
      for (temphost = loadofhostlist,count=0,
           netdelay = getnetdelay(temphost),
           tempcost = (temphost->loadvalue+netdelay)*
           (temphost->cpuspeed); count<loadofhostlist_nr;
           temphost = temphost->next,count++)
    {
           netdelay = getnetdelay(temphost);
           /*getnetdelay 函数只是简单地向 temphost
             发送一个 ICMP 包，计算往返时间*/
           if (temphost>=(temphost->loadvalue+netdelay)
                        *temphost->cpuspeed)
         {
             tempcost=(temphost->loadvalue+netdelay)
                        *(tmphost->cpuspeed)
           /*取得迁移代价最小的目的主机*/
             selected_host = temphost;
         }
    }
             return (selected_host->host);
}
```

4.2 进程迁出模块和监听迁入模块的实现

迁移算法在 Linux 操作系统上的实现，会对 Linux 的内核进行修改和补充。首先给出重要的数据结构、全局操作函数以及进程迁出模块和监听迁入

模块的流程图，并描述进程迁移协议，最后给出重要函数的实现代码。

4.2.1 数据结构和全局操作函数

lprocess 表——保存了正在本地执行的进程的信息。如果某个进程是在本地创建的，又正在本地执行，则该进程同时在 Linux kernel 本身维护的 process 表和本表中分别占有一个条目。本表中每个条目包括如下域：PID、PPID、子进程信息表(children 结构数据)、PGID、PSID、正在等待其死亡的子进程表、status、exit_status、资源使用情况。其中，如果该进程的父进程正在本地执行，则 PPID 域存放的就不是父进程的 PID，而是一个指向父进程在 lprocess 表中条目的指针。类似地，如果该进程所属的 group(或 session 在本地)，则 PGID(或 PSID)域存放的也就不是其组(或 session)的 gid(或 sid)，而是一个指向本机上的 lgroup(或 lsession)表中对应条目的指针。

lgroup 表——保存了本地进程组的信息。所谓“本地进程组”是指那些其 leader process 是在本地创建的进程组。该表中每个条目还包含了该组所有成员进程的信息。

lsession 表——保存了本地会话(session)的信息。它的结构和含义类似 lgroup。

lfileinfo 表——可以被看作是 Linux kernel 本身为每个进程维护的打开文件表的扩展，lprocess 表中每个条目(每个在本地运行的进程)同时对应该表和 lfilepath 表中的一个条目，其中 lfilepath 表的条目存放了打开文件的完整路径名。

mprocess 表——保存了在本地创建而被迁移出去正在远地运行的进程。每个条目包括的域：PID、PPID、当前主机的节点号、PGID、PSID。

children 结构——表示一个子进程。结构成员：PID、PGID、is_exited、is_remote。特别地，如果 is_exited＝＝1 && is_remote==1，即该子进程在远地执行却已经退出，则还应有指针成员 zombptr，指向 zombchild 结构。

zombchild 结构——表示一个已经死去的子进程。结构成员：PGID、退出状态、资源使用情况。

search_lprocess：在 lprocess 表中查找给定进程，找到则返回该进程条目指针。

modify_lprocess：更改 lprocess 表中指定进程的信息。

change_parent：修改 lprocess 表中指定进程的父进程。

add_child：向 lprocess 表中指定进程增加一个子进程，实际修改其中 children 结构。

4.2.2 进程迁出模块和监听迁入模块的流程图

进程迁出模块收到启动进程迁移的信号，以及负载监测模块选择出的待迁移进程和目的主机的信息，就会启动 dummy 进程伪装为待迁移进程接收消息并缓存，然后挂起待迁移进程，向目的节点传送迁移进程的必要信息，在传送完毕后，需要判断是否收到缺页请求，如收到则优先响应请求，传送被请求地址页到目的节点，否则就继续传送进程的剩余地址空间，直到判断传送完毕向目的节点发送“发送完毕”消息。

进程迁出模块流程如图 4.1：

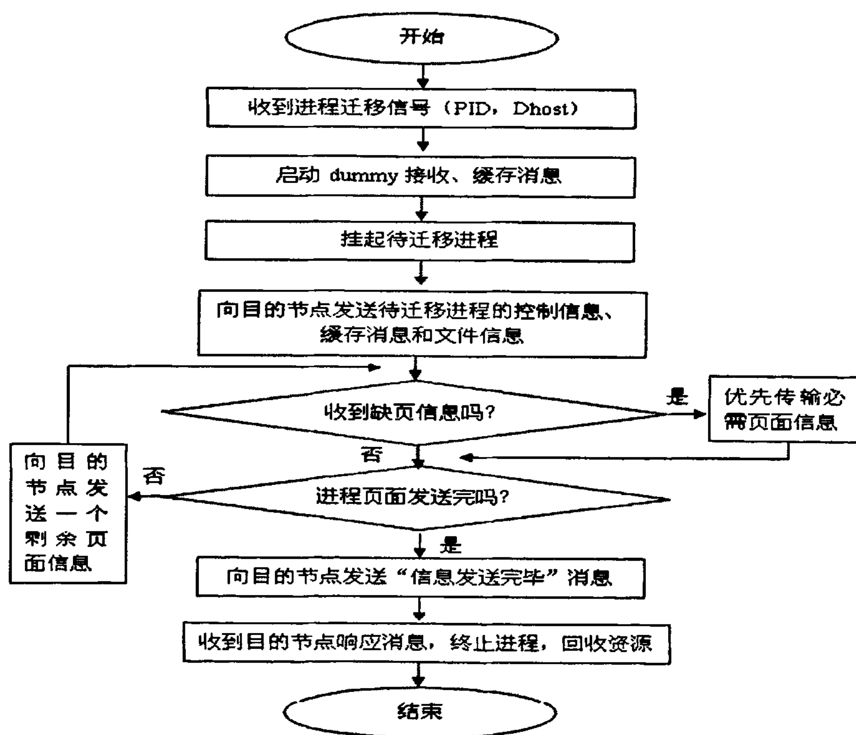

图 4.1 进程迁出模块流程

监听迁入模块同意迁移后，接收源节点传送的迁移进程的必要信息，调用 migrate_do_fork()函数根据收到的信息重构进程，并将其插入就绪队列，等待调度，恢复运行，同时后台继续接收剩余进程信息。在进程运行期间发生缺页时，向源节点发送缺页请求，由源节点发送请求页信息解决缺页。

监听迁入模块流程如图 4.2：

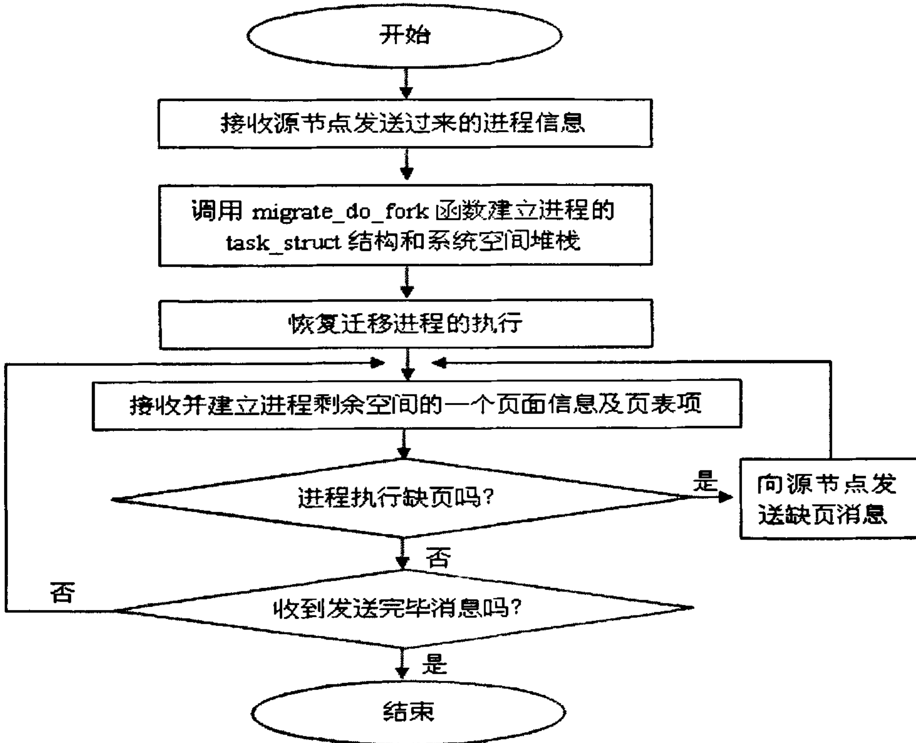

图 4.2 监听迁入模块流程

4.2.3 进程迁移协议

从迁移协议的角度来看，源节点的进程迁出模块中的协议部分，类似于一个客户端 Client，而目标节点的监听迁入模块中的协议部分，就类似于一个服务端 Server。

进程迁移协议是进程迁移课题的另一大类研究。由于个人的时间和精力有限，所以采用的是参考文献[51, 54]的迁移协议，本文只对其进行介绍。下面说明迁移协议的设计。

实际上，可以把 Client 分为上下两层，上层 upper-client 实现设计的专用协议，下层 lower-client。它负责将上层发来的命令装成标准的协议包，并通过 Linux kernel 使用网络功能发送，并且接收通过 Linux kernel 网络功能传上来的协议包，提取出其中的数据交给上层。类似的，Server 也分为上下两层。

可见，lower-client 和 lower-server 将分别负责两端的协议包头的打包/解包, 以及协议包的出错控制，它们的功能是在传输层协议之上一层提供的。

这里倾向于选择 TCP 作为下层协议，因为 TCP 可以帮助解决一些通信的错误, 从而大大减少工作量。然而，当 A 节点与 B 节点之间的连接长期空闲时，A 或 B 某个节点的节点失败却是 TCP 无法检测的，keep alive TCP 虽然可以弥补这一缺点，但是 keep alive TCP 的超时检测时间长达 2 小时，这也是不可接受的。因此，本文选择了 UDP 作为下层协议。

先来分析 Server 这端。lower server 接收到 request message 后，解包，获取请求内容，并且将请求提交给 upper server 执行。

upper server 是多线程服务器，由一个主线程等待在预先定义好的端口上，收到请求后就交给线程池中一个线程处理，主线程又返回继续接收新请求。

由于使用 UDP 作为下层协议，有可能有重复消息到达，为此服务器端必须保存刚才发出的 replies 以处理重复包。当然这些 replies 也需要有专门的线程来把它们从缓存中清除出去。

主线程在系统启动时就开始运行，它创建一些处理线程和一个缓存清除线程，然后就开始在预先指定的端口上等待 requests，当接收到时就检查该 request 是否来自合法的端口(比如来自非系统端口则拒绝——因为请求只应该来自于客户机的内核)，然后主线程把请求挂入请求队列，唤醒那些正在等待请求到达的处理线程，此后主线程又回去继续等待请求到达。

处理线程检查队列上是否有请求存在，如果没有，处理线程睡眠，当被唤醒时再度检查。如果发现队列上有请求存在，就从队列上摘下该请求并处理之，处理时根据消息调用相应的处理程序，得到 reply 后放入缓存区比较，如果缓存中已存在说明该消息已经处理过了，不再发送，否则发送。处理完成后再次检查有无新请求，重复前面的过程。

接着分析 Client 这端。Client 中有两个重要的数据结构。

init_ sentbmp：一个位图，每一位对应一个 host，其值为 1 表示“已经向该 host 成功发送过 INIT 消息” 。

init_ recvbmp：一个位图，每一位对应一个 host，其值为 1 表示“已经从该 host 成功收到过 INIT 消息”。

需要进行进程迁移时，upper client 发出迁移请求，该请求送往 lower client。lower client 首先检查 init_sentbmp 和 init_recvbmp，如果指定的目的节点在两个位图中都为 1(与本机正在通信)，则组装 request 消息，并通过 Linux kernel 网络功能交付网络发送给目的节点。

客户机发出请求后，可能会收到两种网络错误。一是"Connection Refused"，它是指目的节点上服务器程序没有在运行，这是在 ICMP 报告远地机器“Port Unreachable”之后由 UDP socket 报告的；另一是“Network Unreachable”，这是从某个中间路由器接收到同样的 ICMP 消息后报告的，表示网络中有问题使对方机器不可达。前一种错误将被报告给调用者，而后一种错误发生时客户机消息将被自动重发以期望网络错误只是暂时的，而远地机器现在已经可达。

如果没有发生错误但是发生超时，则消息被重传。如果重传多次再失败则返回 TIMEOUT 错误给调用者。

当接收到 reply 时，客户机程序首先将其与发送出去的 request 比较 seq# 和 node-id 以确定该 reply 的确是针对该 request 的回复。如果发生协议错误，或者是 reply 中包含着错误码，则将这些错误返回给调用者。

4.2.4 关键函数解析

本系统监听迁入模块中调用的 migrate_do_fork()函数是在对 Linux 内核原有的 do_fork()函数的改造基础上实现的，因此首先分析 do_fork()函数的实现原理。task_struct 相当于是一个进程的标识符，它包含了该进程的所有信息[64-66]。但由于结构成员较多，所以只介绍跟进程迁移相关的成员。

volatile long state——当前进程的状态。可分为 TASK_RUNNING（运行态)、TASK_ INTERRUPTIBLE(可唤醒睡眠态)、TASK_UNINTERRUPTIBLE（不可唤醒睡眠态)、TASK_ZOMBIE(僵死态)和 TASK_ STOPPED(挂起态)几种状态。运行态进程，将作为迁移的对象。

unsigned long flags——使用此变量来标识进程的一些属性信息。例如，建立进程后是否立刻执行，是否被一信号终止等。

int sigpending——表示进程收到信号后，尚未处理。

mm_segment_t addr_limit——虚存地址空间的上限。对用户进程而言是其用户空间的上限，所以是 0xbfff ffff；对内核线程而言，因为它是可访问整个系统空间的，所以是系统空间的上限 0xffff ffff。

struct user_struct *user——指向用户的指针，每个用户有一个这样的指针，内核线程此指针的值为零。

wait_queue_head_t wait_chldexit——一个进程可以停下来等待其子进程完成使命，此队列头部用来表示子进程等待队列。

struct files_struct *files——当进程有已打开文件时，此指针指向一个 file_struct 结构，否则为零。

struct fs_struct *fs——与文件系统有关的另一个结构。记录的是进程的根目录 root、当前工作目录 pwd、一个用于文件操作权限管理的 umask, 还有一个计数器。

struct signal_struct *sig——如果进程设置了信号处理程序, 指针 sig 就指向一个 signal_struct 数据结构。

struct mm_struct *mm ——mm_struct 数据结构是进程整个用户空间的抽象, 也是总的控制结构。(后面有较详细注解)

struct thread_struct thread——记录着进程在切换时的(系统空间)堆栈指针和取指令地址(也就是“返回地址”)。

对 task_struct 有所了解后，再来看看 do_fork()函数的执行步骤。

do_fork()函数的执行步骤:

```
{   (1)复制 task struct 结构成员:
    {  struct task_struct *p;
       参数检查;
       p=alloc_task_struct();    /*为子进程分配两个连续的物理页面，
                                  低端用作子进程的 task_struct 结构，
                                  高端则用作其系统空间堆栈*/
       *p=*current;              /*整个数据结构赋值。父进程的整个
```

task_struct 就被复制到了子进程的数据结构中。*/

分别检查用户的进程数和内核线程数是否超过上限；
递增具体模块的数据结构中的计数器；
对支持本进程执行映象格式的驱动模块的使用计数器进行递增；
对 task_struct 结构中的某些成员进行清零；
设置进程状态为 TASK_ UNINTERRUPTIBLE； /*因为后面的 get_pid() 操作必须是独占的，当前进程可能会因为一时进不了临界区而只好暂时进入睡眠状态等待 */
将参数 clone_flags 中的标志位略加补充和变换后写入 p->flags；
产生一个新的 pid；
初始化进程父子关系等链接；
初始化子进程等待队列；
初始化待处理信号队列以及有关结构成分；
初始化 task_struct 中各种计时变量；
设置进程开始执行的时间；
}

(2)复制己打开文件的控制结构 files_struct；

(3)复制与文件系统有关的另一个结构 fs_struct；

(4)复制信号；

(5)复制用户空间(copy_mm——此步完成的功能对于迁移过程来说非常重要，后面有详细的阐述)；

(6)复制系统空间堆栈；

(7)设置进程的某些信息(设置优先级、存储页是否可被换出、与父进程的关系等)；

(8)让进程进入它的关系网；

(9)将其挂入可执行进程队列等待调度。

}

迁移算法中除“用户空间信息”之外的所有信息，是必须在进程继续在目标节点上运行之前全部迁移完成的。而对“用户空间信息”的迁移，又是由 copy_mm 函数完成的。因此，难点和重点又落在了对 copy_mm 函数的分析和改写上。下面来看看 mm_struct 结构[56]：

```
struct mm_struct {
struct vm_area_struct *mmap;            /*list of   VMAs*/s
struct vm_area_struct *nmap_avl;             /*tree of VMAs*/
struct vm_area_struct *mmap_cache;   /*last find_vma result*/
pgd_t *pgd;
atomic_t mm_users;                /*How many users with user space?*/
atomic_t mm_count;               /*How many references to
                                   "struct mm_struct"(users count as 1)*/s
int map_count;                    /*number of VMAs*/
struct semaphore mmap_sem;
spinlock_t    page_table_lock;
struct list_head mmlist;          /*List of all active mm's*/
unsigned long start_code, end_code, start_data, end_data;
unsigned long start_brk, brk, start_stack;
unsigned long arg_start, arg_end, env_start, env_endv;
unsigned long rss, total_vm,locked_vm;
unsigned long def_flags;
unsigned long cpu_vm_mask;
unsigned long swap_cnt;      /*number of pages to swap on next pass*/
unsigned long swap_address;
mm_context_t context;
};
```

结构中的头三个指针都是关于虚存区间的。第一个 mmap 用来建立一个虚存区间结构的单链线性队列。第二个 mmap_avl 用来建立一个虚存区间结构的 AVL 树。第三个指针 mmap_cache, 用来指向最近一次用到的那个虚存区

间结构；这是因为程序中用到的地址常常带有局部性，最近一次用到的区间很可能就是下一次要用到的区间，这样就可以提高效率。另一个成分map_count，则说明在队列中(或AVL树中)有几个虚存区间结构，也就是说该进程有几个虚存区间。指针pgd显而易见是指向该进程的页面目录的，当内核调度一个进程进入运行时，就将这个指针转换成物理地址，并写入控制寄存器CR3。另一方面，由于mm_struct结构及其下属的vm_area_struct结构都有可能在不同的上下文中受到访问，而这些访问又必须互斥，所以在结构中设置了用于P,V操作的信号量(semaphore)，即mmap_sem。此外，page_table_lock也是为类似的目的而设置的。

虽然一个进程只使用一个mm_struct结构，反过来一个mm_struct结构却可能为多个进程所共享。所以，还为此设置了计数器mm_user和mm_count。

start_code、end_code、start_data、end_data等分别代表该进程映象中代码段、数据段存储堆以及堆栈段的起点和终点。

最后，分析结构中brk成员的用途。用户程序经过编译、连接形成的映象文件中有一个代码段和一个数据段(包括data段和bss段)，其中代码段在下，数据段在上。数据段中包括了所有静态分配的数据空间，包括全局变量和说明为static的局部变量。这些空间是进程所必须的基本要求，所以内核在建立一个进程的运行映象时就分配好这些空间，包括虚存地址区间和物理页面，并建立好二者间的映射。除此之外，堆栈使用的空间也属于基本要求，所以也是在建立进程时就分配好的(但可以扩充)。所不同的是，堆栈空间安置在虚存空间的顶部，运行时由顶向下延伸；代码段和数据段则在底部，在运行时并不向上伸展。而从数据段的顶部end_data到堆栈段地址的下沿这个中间区域则是一个巨大的空洞，这就是可以在运行时动态分配的空间。最初，这个动态分配空间是从进程的end_data开始的，这个地址为内核和进程所共知。以后，每次动态分配一块“内存”，这个边界就往上推进一段距离，同时内核和进程都要记下当前的边界在哪里。在进程这边由malloc()或类似的库函数管理，而在内核中则将当前的边界记录在进程mm_struct结构中。具体地说，就是由mm_struct结构中的brk来表示动态分配区当前的底部。

下面，便可对copy_mm函数进行分析。

copy_mm函数的执行步骤：

```
{       struct mm_struct *mm,*oldmm;
        初始化变量及参数检查;
        oldmm=current->mm;
        mm＝allocate_ mm ();
memcpy(mm, oldmm, sizeof(*mm)); /*整个数据结构复制。父进程的整个
                                   mm_struct 就被复制到了子进程的
                                   数据结构中*/
retval=dup_mmap(mm);            /*复制 vm_area_struct 数据结构和页
                                   面映射表。vm_area_struct 数据结
                                   构是对虚存区间的抽象。后面有
                                   对 dup_mmap 函数的解析*/
}
```

毫无疑问，接下去应该分析 dup_mmap 函数，此函数的主要功能是复制 vm_area_struct 结构。

dup_mmap 函数的执行步骤:

```
{
  struct vm_area_struct *mpnt, *tmp, **pprev;
  初始化 mm_struct 结构中的部分成员;
  pprev=&mm->mmap;                /*设置单链线性列头*/
for(mpnt=current->mm->mmap; mpnt; mpnt= mpnt->vm_next)
/*轮询父进程中的虚拟区间抽象 vm_area_struct 结构，逐个进行复制*/
{
    tmp=kmem_cache_alloc(vm_area_cachep, SLAB_ KERNEL);
    /*在内存中分配空间*/
    *tmp＝*mpnt;                  /*整个数据结构赋值。父进程的整
                                    个 vm_area_struct 就被复制到
                                    了子进程的数据结构中。*/
    对 mm_struct 和 vm_area_struct 结构的部分成员进行赋值;
    对通过 mmap()映射到某个文件的区间进行处理;
    retval = copy_pape_range(mm, current->mm, tmp);
```

/*逐层处理页面目录项和页面表项。此函数很关键，在后面有详细阐述*/

打开页面映射；

建立线性队列链接；

}

如果虚存区间到了一定数量，便建立AVL树，便于查找；

}

接着分析copy_page_range函数。此函数主要的功能是复制页面映射表，函数主体是三个大循环，最外层的循环是对页面目录项的循环，中间层的循环是对中间目录项的循环，最里层的循环是对页面表项的循环。重点是在最里层的循环上。循环中检查父进程一个页面表中的每个表项，根据表项的内容决定具体操作。而表项的内容则无非是下面这么一些可能：

(1)表项内容为全零。说明该页面的映射尚未建立，或者说是个“空洞”，因此不需做任何事。

(2)表项的最低位为零。说明映射已建立，但是该页面目前不在内存中，已经被调出到交换设备上。此时表项的内容指明“盘上页面”的地点，而现在该盘上页面多了一个“用户”，所以要通过swap_duplicate()递增它的共享计数。然后，就转到cont_copy_pte_range将此表项复制到子进程的页面表中。

(3)映射已建立，但是物理页面不是一个有效的内存页面。有些物理页面在外设接口卡上，相应的地址称为“总线地址”，而并不是内存页面。这样的页面、以及虽是内存页面但由内核保留的页面，是不属于页面换入/换出机制管辖的，实际上也不消耗动态分配的内存页面，所以也转到cont_copy_pte_range将此表项复制到子进程的页面表中。

(4)需要从父进程复制的可写页面。本来，此时应该分配一个空闲的内存页面，再从父进程的页面把内容复制过来，并为之建立映射。显然，这个操作的代价是不小的。而Linux内核采用了一种称为“copy in write”的技术，先通过复制页面表项暂时共享这个页面，到子进程(或父进程)真的要写这个页面时再来分配页面和复制。详细的过程是，首先将父进程的页面表项改成写保护，然后把已经改成写保护的表项设置到子进程的页面表中。这样一来，

相应的页面在两个进程中都变成“只读”了，当不管是父进程或是子进程企图写入该页面时，都会引起一次页面异常。而页面异常处理程序对此的反应则是另行分配一个物理页面，然后将两个页面表中相应的表项改成可写。

(5)父进程的只读页面。这种页面本来就不需要复制。因而可以复制页面表项共享物理页面。

前面分析了在 Linux 操作系统中进程建立的全过程，并且重点分析了与本文的进程迁移相关的部分。接下来，对 Linux 的相关机制进行补充，从而在 Linux 操作系统上实现在目标节点上“fork”“待迁移进程”的过程。

在 Linux 系统中，除 init 进程外，每个进程都是由它的父进程复制出来的，它也只有一个父进程。但由于这个被迁移进程是从集群系统中别的机器上迁移过来的，在此之前它已运行了一段时间，所以在目的节点上重建该进程时，它将有两个“父进程”：一个是源节点上的源进程，一个是 init 进程。在“捏造” 该进程时，它的绝大多数信息都来自于源进程，而它的“关系网”——即它与左临右舍的关系则来自于 init 进程。并且在该进程建立完成后，init 进程是它的最终父进程。

本文所选择的迁移算法的难度在于:要实现只把进程的部分状态信息(进程执行时的必需状态)传送到目的节点上。要建立起进程的“必需状态”信息，就需要改写 do_fork 函数，以及 copy_mm 函数，dup_mmap 函数和 copy_page_range 函数，而得到 migrate_do_fork 函数，migrate_copy_mm 函数，migrate_dup_ mmap 函数和 migrate_ copy_page_ range 函数。

在文中进程迁移设计中，由于现在所要建立的进程已经在别的地方运行了一段时间了，而且现在要接着往下执行，所以它必须把已经存在的物理页面一个不落的拷贝过来。但是为什么又不拷贝页面表项呢?因为现在还没有把物理页面的信息复制到目的节点上，物理页面都还不存在，又怎么能有它的页面映射——页面表项呢?

此时已经复制出被迁移进程自己的 task_struct 结构和系统空间堆栈，实现了 migrate_do_fork()函数。而新的“用户空间”实际上只是个框架，一个“空壳”，里面一个页面也没有。下面就来分析目的节点分配物理页面，建立相应页表项，接收一个页面的过程，也就是实现 receive_page 函数。

首先，应该了解内核中经常要用到这样的操作:给定一个属于某个进程的

虚拟地址，要求找到其所属的区间以及相应的 vma_area_struct 结构。这是由 find_vma 函数来实现的，其代码在 mm/mmap.c 中。当说到一个特定的用户空间虚拟地址时，必须说明是哪一个进程的虚拟空间中的地址，所以函数的参数有两个，一个是地址，一个是指向该进程的 mm_struct 结构的指针。首先看一下这地址是否恰好在上一次(最近一次)访问过的同一个区间中。这也正是在 mm_struct 结构中设置一个 mmap_cache 指针的原因。如果没有命中的话，那就要搜索了。如果已经建立过 AVL 结构(指针 mmap_avl 非零)，就在 AVL 树中搜索,否则就在线性队列中搜索。最后，如果找到的话，就把 mmap_cache 指针设置成指向所找到的 vm_area_ struct 结构。函数的返回值为零(NULL)，表示该地址所属的区间还未建立。此时通常就得要建立起一个新的虚存区间结构，再调用 insert_vm_struct()将其插入到 mm_struct 中的线性队列或 AVL 树中去。

理解了 find_vma 函数后，就不难理解 receive_page 函数了。

```
int receive_page(unsigned long addr)
{
        pgd_t *pgd;     pmd_t *pmd;    pte_t *pte;
        unsigned long page = 0;
        struct vm_area_struct *vma;
        struct task_struct *p=current;
        if(!(vma=find_vma(p->mm, addr)))
                return -EINVAL;
        if(!(page=_get_free_page(GFP_KERNEL)))
              return -EINVAL;
     /*从接收到的数据中拷贝PAGE_SIZE大小的数据到page指向的区域*/
          CopyData((void *)page, PAGE_SIZE, 0));
        /*建立并设置新的页表项*/1
        pgd=pgd_offset(p->mm,addr);
        if (!(pmd = pmd_alloc(pgd, addr))) return -EINVAL;
        if (!(pte = pte_alloc(pmd, addr))) return -EINVAL;
        if (!pte_none(*pte)) return -EINVAL;
```

```
        set_pte(pte,pte_mkdirty(mk_pte(page, vma->vm_page_prot)));
        return 0;
    }
```

但是，监听迁入模块要怎么样才能发现有页面不再本机呢？这需要了解Linux的相关机制。

在程序的执行过程中，因为遇到某种障碍而使CPU无法访问到相应的物理内存单元，即无法完成从虚拟地址到物理地址映射的时候，CPU会产生一次缺页异常，从而进行相应的缺页异常处理。基于这一特性，Linux采用了请求调页(Demand Paging)和写时复制(Copy On Write)的技术。

请求调页是一种动态内存分配技术，它把页框的分配推迟到不能再推迟为止。这种技术的动机是:进程开始运行的时候并不访问地址空间中的全部内容。事实上，有一部分地址也许永远也不会被进程所使用。程序的局部性原理也保证了在程序执行的每个阶段，真正使用的进程页只有一小部分，对于临时用不到的页，其所在的页框可以由其它进程使用。因此，请求分页技术增加了系统中的空闲页框的平均数，使内存得到了很好的利用。从另外一个角度来看，在不改变内存大小的情况下，请求分页能够提高系统的吞吐最。当进程要访问的页不在内存中的时候，就通过缺页异常处理将所需页调入内存中。

Linux采用了“写时复制”技术，主要应用于系统调用fork，父子进程以只读方式共享页面表项，当其中之一要修改页面表项时，内核才通过缺页异常处理程序分配一个新的页面表项，并将页面表项标记为可写。这种处理方式能够较大的提高系统的性能，这和Linux创建进程的操作过程有一定的关系。在一般情况下，子进程被创建以后会马上通过系统调用execve()将一个可执行程序的映象装载进内存中，此时会重新分配子进程的页面表项。那么，如果fork的时候就对页面表项进行复制的话，显然是很不合适的。

在上述的两种情况下出现缺页异常，进程运行于用户态。异常处理程序可以让进程从出现异常的指令处恢复执行，使用户感觉不到异常的发生。对于无法正常恢复的异常，这时，异常处理程序会进行一些善后的工作，并结束该进程。

因为现在目的节点上的页面及页面表项都还不存在，类似于上述的第二

种缺页异常情况，所以将重点关注 Linux 中进行缺页异常处理函数的相关执行步骤。

```
do_page_fault
{
        ......
    asm("movl %%cr2,%0":"=r" (address));    /*用 CR2 中引起缺页异常的虚拟地址给 address 赋值*/
    vma=find_vma(mm, address);              /*找到相应的虚存区间*/
        ......
switch (handle_mm_fault(mm, vma, address, write)) /*页面异常处理程序的关键*/
        ......
}
handle_mm_fault
{
  为要映射到进程虚拟地址空间的页分配三级页表中相应的页表入口指针;
    return handle_pte_fault(mm, vma, address, write_access, pte);
  /*调用真正处理页面错误的函数 handle_pte_fault*/
        ......
}
handle_pte_fault
{
  if(物理页面在内存中)
  {
    if(表项为空)              /*映射尚未建立，此为文中所面临的情况*/
    {
    return do_no_page(mm, vma, address, write_access, pte);
    /*do_no_page 函数的功能是建立映射，其实还包含匿名页分配*/
        ......
```

```
    }
        ……
    }
        ……
}
```

基于上述分析发现，不必关心do_no_page函数的具体细节，因为现在还没有物理页，就更不能去建立它的映射了。应该在现有的do_no_page函数中的开始部分加上这样一段代码：

```
if (current->pid= =migrate->pid)
{
    sys_kill(MigrateIn->pid,SIG_NOPAGE);
    return 2 ;
}
```

这里主要实现向监听迁入模块发送缺页的信息，并告诉它所缺页面的虚拟地址，当然还需要进行两台主机间虚拟地址转换。

4.3 如何在Linux内核添加功能

Linux操作系统已是一个专业化的自由软件，它具有Unix操作系统所具有的功能，对外接口与Unix操作系统很相似，且具有很强的网络功能。但Linux操作系统没有并行处理能力，要使其具有分布式并行处理能力，就必需扩充它的功能。

众所周知，用户作业只能在用户态下执行。而要迁移一个正在执行的用户进程，在用户层是没有办法做到的，必需进入核心态，在操作系统内核增加取进程数据和恢复进程数据代码。这也是进程迁出模块和监听迁入模块需要实现的功能。为了使用增加的功能，可以采取重新编译内核或动态加载模块。

(1)重新编译内核方式

Linux的内核由几十个C语言（或汇编语言）程序文件组成，这些文件主要放在系统的/usr/src/linux的目录下，此目录下又有一些子目录，如kernel、

mm、fs等。可以在此目录下增加一个与kernel并列的migrate子目录，用于存放实现系统功能的代码，作为扩充的C语言程序文件。同时，修改此目录下的Makefile文件，把新子目录migrate的有关信息加到这个文件中。这些工作做完后，需要在此目录下通过make命令编译、连接内核C语言程序源代码就可生成一个新的内核映像文件，它可包含新增的进程迁移系统处理程序。如果采用这种方式，将进程迁移系统直接编译进内核，则在代码调试的过程中，会频繁编译内核，启动内核而消耗很多时间。本文中没有采用这种方式，而是选择采用模块加载的方式。

(2)模块加载方式

Linux可以动态的加载与卸载操作系统部件。Linux模块就是这样一种可在系统启动时或启动后的任何时候动态连入核心的代码块。动态加载模块的好处在于可以让核心保持很小的尺寸同时非常灵活。本文中采用这种动态模块加载方式，令Linux内核模块开机自动挂载支持进程迁移的功能模块。通常这需要修改模块的配置文件modules.conf或modprobe.conf，只需在这个文件中写入模块的加载命令或模块的别名的定义等。

加载模块需要访问核心资源[56]。例如模块需要调用核心内存分配例程kmalloc()来分配内存。核心在其核心符号表中维护着一个核心资源链表，这样当加载模块时能解析出模块中对核心的引用。但模块只能访问符号表中对模块输出的资源，可是本系统功能模块需要访问没有输出到模块的函数和数据。为解决这个问题，实现时采用一种“欺骗”技巧，使用/boot/System.map文件。该文件在内核编译时产生，其中包含了所有内核符号的绝对地址(甚至是没有对模块输出的符号)。例如在System.map文件中tcp_v4_rehash()内核函数的绝对地址是c0173e00，则可以直接通过该地址调用该内核函数。

4.4 本章小结

本章的主要内容是在Linux上实现进程迁移系统。分别介绍了各个功能模块的实现，对关键函数进行了解析，并描述了进程迁移协议，以及讨论了怎样令迁移系统与Linux更好的融合。

第5章 系统测试结果与分析

5.1 测试环境

本章对进程迁移系统进行功能和性能上的测试。

测试环境为：1个主节点、1个备份节点和8个从节点以太网连接构成的海豚一号集群服务器。由于本文设计的进程迁移系统，不存在主控节点，节点间是对等关系，所以实现时通过修改每个节点的hosts文件，屏蔽掉集群系统中的主控节点和备份节点，只允许8个从节点各自交互。节点分别运行加载了功能模块的Linux内核。编写了运行时大小约为100KB的测试程序进行迁移测试。在进程迁移时，任选8个从节点中的一台作源节点，另一台最为目的节点。如图5.1：

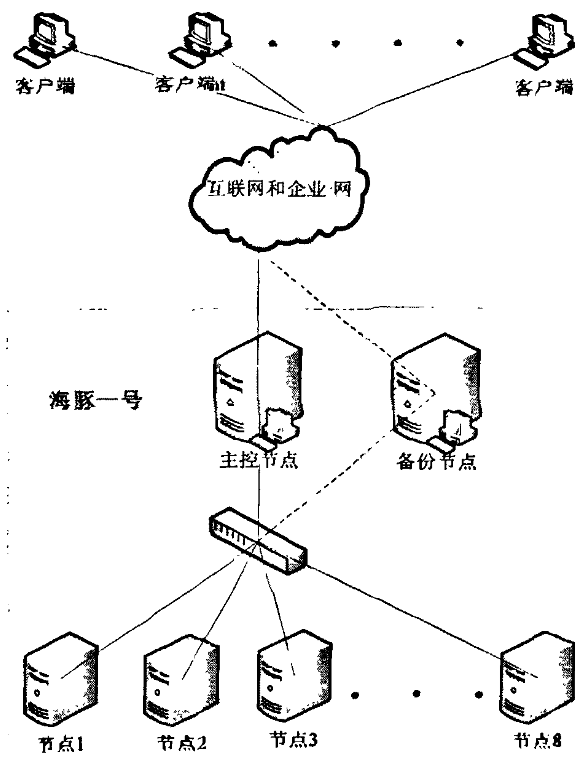

图5.1 海豚一号集群服务器

集群中每个节点配置 Cerelon 2.4GHz 处理器、2G 内存、200G 硬盘和 10/100M 网卡。

5.2 系统测试结果与分析

进程迁移系统的测试包括功能测试和性能测试。功能测试的目的是检验进程迁移系统中各加载模块是否实现预定的功能目标。性能测试的目的有两点：一是用本系统的迁移算法与别的迁移算法进行性能比较；二是将本系统与已有进程迁移系统进行性能比较。

5.2.1 功能测试

系统中包括三个模块：负载监测模块、进程迁出模块和监听迁入模块。负载监测模块定期监测本机是否超载，如果超载，就向进程迁出模块发送信息启动迁移过程；进程迁出模块需要与监听迁入模块建立通信，传送进程信息，同时，目的节点的监听迁入模块根据接收信息重构进程，并恢复运行。整个系统对用户透明，即使用户运行在本机的进程被迁移到目的节点运行，但由于执行完毕后，目的节点通过监听迁入模块发送给源节点的进程迁出模块最后结果，再由源节点的进程迁出模块将结果返回给用户，所以用户并不知道有进程迁移的存在。功能测试就是检验每个模块是否实现了预期的功能，同时还要检验它们集成后是否可以正确地实现预期进程迁移系统的功能。

通过实验，发现各模块工作正常，且可以很好整合通信，实现了预期功能。

5.2.2 性能测试

（1）改进的迁移算法与其他迁移算法的性能比较

为了对迁移算法进行比较，将 Demand Page 算法作为一个选项实现在系统中，没有加入 Total-Copy 和 Flushing 算法的原因是因为这两种算法在恢复进程执行前将传送进程的全部进程状态信息，这在时间上将肯定长于传送部分进程状态信息的改进算法。没有考虑实现 Pre-Copy 算法，是因为它需要如 SMP 结构的并行支持，同时该算法会多次传送地址空间页，这会导致整个进程迁移时间变长，而改进的算法是令源节点传送地址空间和目的节点上的进

程执行并行，会减少进程迁移时间，即使开始时会经常产生缺页，导致进程执行延时，但随着地址空间的传送，将会极大降低缺页次数，减少进程执行的延时。

测试结果如图 5.1：

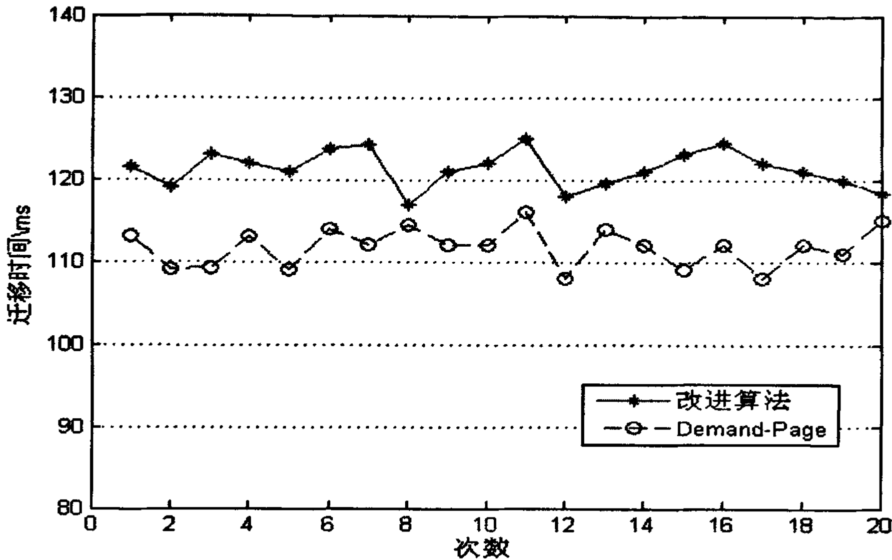

图 5.1 改进算法与 Demand-Page 算法性能比较

从实验结果可以看出，改进算法的迁移时间绝大多数在 120ms 左右。Demand Page 算法的迁移时间大都在 110ms 左右，改进算法略慢于 Demand Page 算法的主要原因是改进算法要比 Demand Page 算法一开始多传送了转发的消息等。表面上看，改进的算法的迁移时间不但没缩短，反而变长了，在分析之后会发现其实不然，Demand Page 算法事先没有传送进程的地址空间到目的节点，因此进程恢复执行后，每次发现缺页，都被挂起然后向源节点请求页面，每次的挂起延时将极大影响进程执行的速度。而改进的算法虽然初期和 Demand Page 算法很像，也会向源节点请求缺页， 由于目的节点后台接收进程其余页面，后期势必会极大减少迁移后进程的缺页请求，这也将加快进程的执行，这一优点也是其它算法所不具有的。

（2）本系统与其他进程迁移系统性能比较

使用 pmake 在本系统上进行实验。采用“编译一棵源码树”的任务来测

试迁移系统的宏观性能。采用的源码树其中有 50000 行代码，要求编译它们然后将它们连接成一个库。

pmake 是 make 工具的一个版本，它允许独立的编译步骤以并行地创建多个进程。真正的并行化是通过允许那些正在执行 gcc 的进程被迁移到其他机器上执行而达到的。

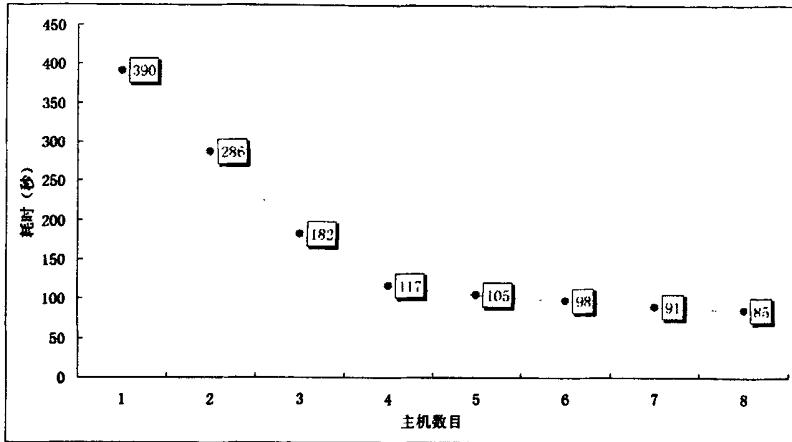

图 5.2 本系统进行 pmake 的速度与节点数关系

从图 5.2 上可以看出，当编译在 1-8 台机器上运行时，其耗时减少，可以看到速度从 1 台主机到 8 台主机时提高了约 4.6 倍。其值低于理想状态下 8 倍的原因是连接和库的创建是不能并行进行的。

虽然在许多集群系统中已经实现了进程迁移，但是很难在相关的论文中找到整体性能报告数据。这里列出所能查到的系统性能数据，并进行比较。

从参考文献[57]中得到图 5.3。注意其中 pmake 一项，在节点数为 8 时，加速倍数约为 4.1。

从文献[58]中得到图 5.4。据参考文献，Enterprise 是一个图形计算环境，用在一个分布式硬件环境中设计、编码、调试、监视和执行程序。

从图 5.4 可知，Enterprise 系统性能较优越。当分布式系统中的节点数达到 8 时，加速倍数约为 4.8 倍，这个值要比本文中的迁移系统得到的 4.6 要大。但是从图 5.3 可发现，pmake 比起一般的应用程序而言，其加速倍数较低。由此可看出，本系统其性能并不比 Enterprise 系统逊色。

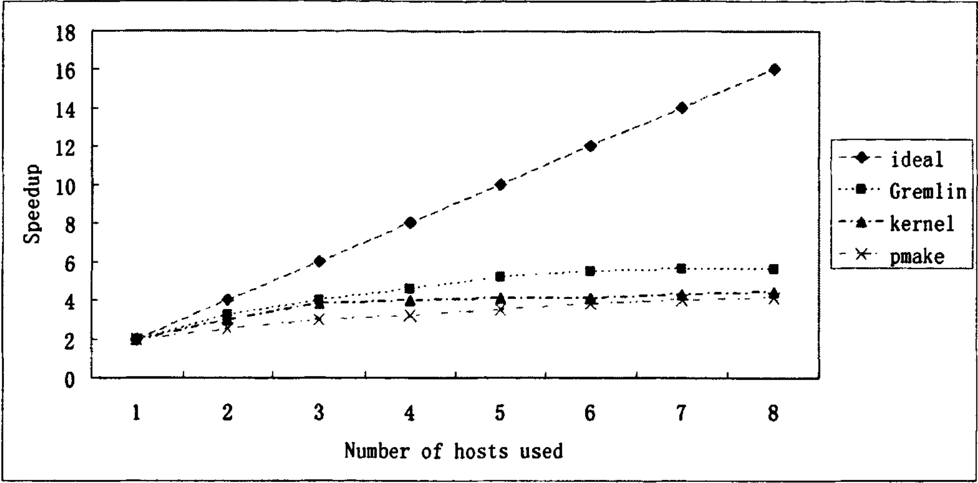

图 5.3 Sprite 系统进行 pmake 的速度与节点数关系图

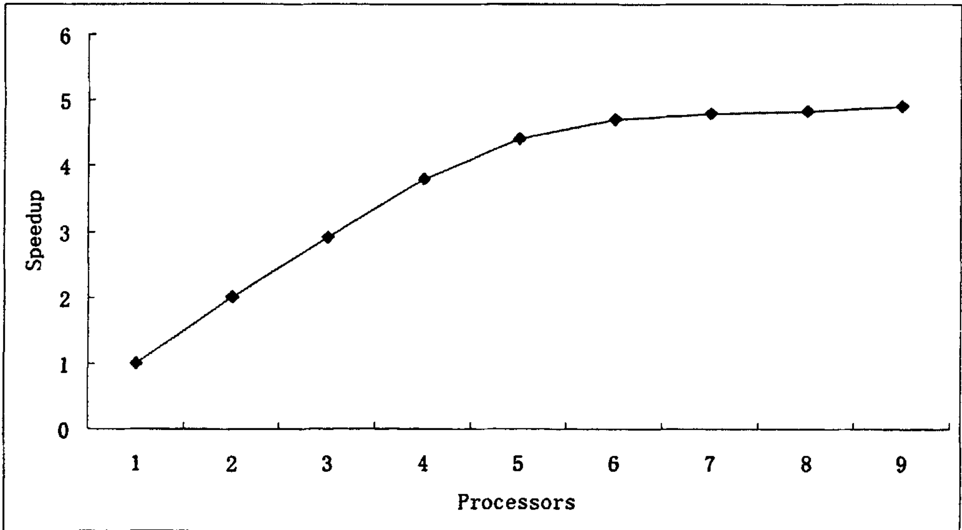

图 5.4 Enterprise 系统中的速度与节点数关系图

5.3 本章小结

在海豚一号集群系统中，对进程迁移系统进行了功能测试和性能测试。

功能测试检验进程迁移系统中各加载模块功能正常。性能测试中将本系统的改进算法与已有迁移算法做了比较测试，改进算法将进程执行与传送状态信息尽可能并行，减少了进程迁移总延时，具有很好的优势；同时使用 pmake 在本系统上编译一棵源码树，测试了系统性能，与已有的系统做性能比较，证明本系统与 Enterprise 系统一样，具有较高的性能优越性。

结　　论

进程迁移一直是国际上比较活跃的研究课题。国内外现有的进程迁移系统还存在着诸多问题。迁移算法落后、迁移时间过长、过重的网络通讯量，而且几乎都是在各商用操作系统中实现的，由于其源代码不公开，所以很难通晓其内部运行的机理。另一方面，对于最大的开放源代码的操作系统 Linux 来说，在其内核中还没有正式支持进程迁移功能。同时集群系统越来越多地使用 Linux 操作系统，因此研究进程迁移系统在 Linux 上的实现，是具有一定的现实意义的。

本文对进程迁移的相关理论进行了分析，基于此对进程迁移系统进行模块的划分：负载监测模块、进程迁出模块和监听迁入模块。在分析各模块功能的同时，给出了各模块的实现流程图。实际上，进程迁出模块和监听迁入模块的实现也就是进程迁移算法的实现。文中对已有的进程迁移算法进行了分析，在借鉴各算法的优点基础上提出了一种改进的进程迁移算法，最大程度地将进程状态迁移和进程的运行并行起来，从而提高了迁移速度，网络通信量也较小，而且也没有对源节点的剩余依赖性，并通过实验验证了算法的有效性。

在实现进程迁移算法时，修改了 Linux kernel，增加了可动态加载的功能模块，并依靠 NFS 实现文件描述符的迁移，以及采用 dummy 进程对进程消息进行缓存转发。最终实现了进程在节点之间的迁移。

由于时间原因，系统中尚存在可以进一步改进的地方。

(1) socket 迁移问题，转发机制速度慢，可以考虑采用对 Linux 中网络部分实现的修改，在保证与原有语义一致的基础上增加对 socket 迁移的支持，当然这会进一步造成系统可移植性降低。

(2) 管道迁移问题。由于采用 NFS 不支持管道，所以本系统没有解决迁移进程使用管道通信的问题。

参考文献

[1] 黄克文.分布式系统的进程迁移.百色学院学报.2006,19(6):50-54 页

[2] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G.C. Bezerra. Integrade: Object-oriented grid middleware leveraging idle computing power of desktop machines. Concurrency and Computation: Practice & Experience. 2004:449-459P

[3] Kapadia, Fortes. PUNCH: An architecture for web-enabled widearea network-computing [J]. Cluster Computing. 1999:153-164P

[4] 王勇,王忠群,韦良芬.移动 Agent 的一种支持安全与容错的迁移机制.计算机技术与发展.2007,17(3):169-171 页

[5] Rajkumar Buyya. High Performance Cluster Computing, Volume 1-Architectures and Systems. Prentice Hall,1999:1-78P

[6] Andrew S. Tanenbaum. Distributed Operationg System. 北京:电子工业出版社.2001:1-107 页

[7] Buyya R.高性能集群计算:结构与系统(第一卷).郑纬民,石威,汪东升等译.电子工业出版社,2001:6-157 页

[8] Tho Anderson, David Culler, David Patterson. A Case for NOW (Networks of Workstations). IEEE Micro.1995:54-64P

[9] T. Sterling, D. Becker, D. Savarese, et al.BEOWULF: A Parallel Workstation for Scientific Computation. Proceedings of the 1995 International Conference on Parallel Processing(ICPP), August 1995:11-14P

[10] 刘锦德,刘崇威.Linux 操作系统纵览.计算机应用.2000,20(3):1-3 页

[11] 陈向阳等.Linux 实用大全.北京:科学出版社. 2000:18-56 页

[12] 雷澍等.Linux 的内核与编程.北京:机械工业出版社. 2000:1-36 页

[13] Milojicic, Giese, Zint.Experiences with load distribution on top of the Mach microkernek .USENIX Distributed and Multiprocessor

Systems(SEDMS IV).1993:19-36P

[14] A. Barak, O. La' adan. The MOSIX multicomputer operating system for high performance cluster computing. Journal of Future Generation Computer Systems.1998,13(4-5):361-372P

[15] D. S. Milojicic et al. Task migration on the top of the Mach microkernel. In 3rd USENIX Mach Symposium.1993:273-289P

[16] Y. Artsy, R. Finkel. Designing a process migration facility: The Charlotte experience. IEEE Computer. 1989:47-56P

[17] F. Douglis, J. K. Osterhout, M. F. Kaashoek, A. S. Tanenbaum. A comparision of two distributed systems: Amoeba and Sprite. Computing Systems.1991,4(3):353-384P

[18] Cheriton. The V distributed system. Communications of the ACM. March 1988,31(3):314-333P

[19] Ousterhout, Cherenson, Douglis, et.al.The Sprite network operating system. IEEE Computer. Feb. 1988:23-36P

[20] LitzKow, Livny. Experience with the Condor distributed batch system. IEEE Workshop on Experimental Distributed Systems. 1990:97-101P

[21] LitzKow, Livny, Mutka. Condor - a hunter of idle workstations. 8th International Conferrence on Distributed Computer Systems. 1988:104-111P

[22] STELLNER G. CoCheck. Checkpointing and Process Migration for MPI[A]. Proc. of the International Parallel Processing Symposium[C]. IEEE Computer Soc. Press. 1996:526-531P

[23] 汪东升，沈美明.一种基于检查点的卷回恢复与进程迁移系统[J].软件报.1999,10(1):68-73页

[24] 裴丹，汪东升.工作站网络系统进程迁移机制[J].软件学报.1999,10(10):1032-1037页

[25] Tanenbaum, A.S. Modern Operating Systems. Prentice Hall, Englewood Cliffs, New Jersey.1992

[26] Fred Douglis, John K. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. Software - Practice and Experience. 1991,21(8):757-785P

[27] 庞毅林,杨国勋,蒋翠玲.分布式系统中进程迁移地分析.武汉理工大学学报.2001,25(3):251-254页

[28] Li K, Naughton J F, Plank J S. Real-time concurrent checkpoint for parallel programs. ACM SIGPLAN Notices. 1990,25(3):79-88P

[29] Pankaj Mehra, Benjamin W Wah. Automated Learning of Workload Measures for Load Balancing on a Distributed System [A]. Int' 1 Conf on Parallel Processing. 1993:263-270P

[30] N G Shivaratri, P Krueger, M Singhal. Load Distributing for Locally Distributed Systems [J]. IEEE Computer. 1992:33-44P

[31] M Harchol-Balter, Downey. Exploiting Process Lifetime Distributions for Dynamic Load Balancing [J]. ACM Trans on Computer Systems. 1997,15(3):352-385P

[32] 肖会会,吴一新,傅育熙. Linux 下集群系统 openMosix 分析.计算机仿真.2005,22(4):142-145 页

[33] Douglis and Ousterhout. Process migration in the Sprite operating system 7th Internationl Conference on Distributed Computing. 1987:18-25P

[34] Douglis and Ousterhout. Process migration in Sprite: a status report. IEEE-CS TCon Operating Systems Newsletter. 1980,3(1):8-10P

[35] Douglis. Experience with process migration in Sprite. USEN IX Distributed and Multiprocessor Systems Workshop. 1989:59-71P

[36] Douglis and Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. Software-Practice and Experience. 1991,21(8):757-786P

[37] D.Freedman. Experience building a process migration subsystem for UNIX. In USENIX Winter Conference. 1991:349-354P

[38] Kasidit Chanchio, Xian-He Sun. SNOW: Software Systems for Process Migration in High-Performance. Heterogeneous Distributed Enviroments, ICPP Workshops. 2002:589-596P

[39] E. Jul. Implementation of distributed objects in Emerald. In Int Workshop on Object Orientation in Operating Systems, Palo Alto, CA, 1991

[40] E. Jul, H. Levy, N. Hutchinson, A. Black. Fine grained mobility in the Emerald system. ACM Transactions on Computer Systems. 1988,6(1):109-133P

[41] Peter Smith and Norman C. Hutchinson. Hetergeneous process migration: The Tui system. Technical Report TR-96-04, UBC Computer Science Department, Vancouver, B.C. 1996

[42] Finkel and Artsy. The process migration mechanism of Charlotte. IEEE-CS TC on Operating Systems Newsletter. 1989,3(1):11-14P

[43] E. Zayas. Attacking the process migration bottleneck. In 11th ACM Symposium on Operating Systems Priciples. 1987:13-24P

[44] E. Zayas. The Use of Copy-on-Reference in a Process Migration System. [PhD thesis]. Carnegie Mellon University. 1987

[45] G. W. Gerrity, A. Goscinski, J. Indulska, W. Toomey, and W. Zhu. RHODOS- a testbed for studying design issues in distributed operating systems. In Towards Network Globalization (SICON 91):2nd International Conference on Networks. 1991:268-274P

[46] Theimer, Lantz and Cheriton. Preemptable remote execution facilities for the V-system. 10th ACM Symposium on Operating Systems Principles. 1985:2-12P

[47] Theimer. Peremptable Remote Execution Facilities for Loosely-couple Distributed Systems [PhD thesis] Stanford University. 1986

[48] Cong Du, Xian-He Sun and Ming Wu. Dynamic Scheduling with Process Migration. Seventh IEEE International Symposium on Cluster

Computing and the Grid.2007

[49] Aric D.Blumer,Henning Mortveit and Cameron D. Patterson. Formal Modeling of Process Migration. International Conference on Programmable Logic and Applications.2007:104-110P

[50] 许封元,房至一,朱维平.进程迁移对负载平衡影响的实验.吉林大学学报.2006,44(6):925-930页

[51] Pawel Czarnul. Dynamic Process Paritioning and Migration for Irregular Applications. Proceedings of the International Conference on Parallel Computing in Electrical Engineering, 2002:22-25P

[52] 储杰,金海,范开钦,杨志玲.一种通用可扩展的抢占式集群进程迁移系统.计算机工程与科学.2005,27(6):102-104页

[53] Leland, W. and Ott, T. Load-Balancing Heuristics and Process Behavior. Proceedings, ACM SigMetrics Performance Conference. 1986:54-69P

[54] 邓海燕.基于 Linux 的分布式系统中的进程迁移及实现.电子科技大学硕士论文.2006:18-29,38-40页

[55] 毛德操,胡希明. Linux 内核源代码情景分析(第一版) [M] 杭州:浙江大学出版社.1999:49-68,269-276页

[56] J. Duell, P. Hargrove, E. Roman. The Design and Implementation of Berkeley Lab' s Linux Checkpoint/Restart.Tech. Rep. pulication LBNL-54941, Berkeley Lab. 2002

[57] Freedman, D. Experience Building a Process Migration Subsystem for UNIX. Proceedings of the USENIX Winter Conference.1991:349-355P

[58] Ian Parsons. An appraisal of the Enterprise Model.[MS thesis] University of Alberta.1993

攻读硕士学位期间发表的论文和取得的科研成果

项目实践：

1. 哈尔滨工程大学基础平台建设开发项目：集群计算系统研究

2. 哈尔滨工程大学基础研究基金：分布式计算机系统容错策略的研究（HEUFT05009）

致　谢

首先，要感谢我的导师汪东升教授。汪老师以自己深厚的理论造诣、丰富的实践经验和对当前学科敏锐的洞察力，为我的研究工作提供了有力的指导和帮助；相信我在以后的工作和学习过程中，汪老师严谨的治学态度，敏锐的思维能力，积极的人生态度会使我终身收益。

感谢门朝光教授对我在学习和生活上的帮助和关心。感谢实验室的其他老师：赵蕴龙、姚念民等。感谢你们对我的帮助和指导。

在论文的撰写过程中得到了周围很多同学的大力支持和无私指导，包括关心帮助和鼓励过我的师兄弟妹、同学、朋友，衷心感谢他们。

最后，感谢我的父母，你们在物质上和精神上都给予了我极大的支持，使我能够安心地进行工作和学习。