代　号 10701　　学　号 0321421213

分类号 TP316　　密　级 公开

西安电子科技大学

硕士学位论文

题（中、英文）目 Octopus OS 系统调度的研究与设计

A Research and Design of Octopus OS Scheduling

作 者 姓 名 闫园　指导教师姓名、职务 王力 教授

学 科 门 类 工学　学科、专业 计算机软件与理论

提交论文日期 二〇〇六年二月

摘要

传统操作系统的体系构架采用层层包裹的形式为用户提供一个统一接口的虚拟机，这种结构最大限度的保证了操作系统底层结构的安全可靠并为用户提供便捷的服务，但是随着操作系统本身和上层应用复杂性的提高，传统的模式将导致系统性能和可信度的发展与提高都受到很大的制约和影响。基于状态的操作系统 Octopus OS 从系统的体系结构出发采用集中管理系统重要状态并自治执行的方式为解决这一问题提供了新的思路。

本文在分析基于状态操作系统和 Octopus OS 体系结构以及传统系统调度算法的基础之上，结合系统特性、针对系统调度需求，为 Octopus OS 提出了系统调度方案：对于传统开环的调度策略导致系统资源利用率低甚至可靠性降低的问题，采用闭环的方式实时反馈系统执行的状态，以判断所选择的调度算法；针对任务自治执行的特性，当资源紧缺时，借鉴微观经济学理论中的竞价模型，通过求得系统的影子价格，以达到系统宏观控制、任务自主竞争的局面从而实现系统资源分配的优化。文中借助 Matlab 仿真，主要从资源的竞价分配角度分析了上述的调度策略。最后，研究和探讨了在 Bochs 虚拟机、以 X86 为体系构架，构建模拟、调试环境的过程，并基于该环境对 Octopus OS 系统进程调度的实现的相关内容进行了讨论。

关键词：自治计算　操作系统调度　反馈控制　竞价模型　Bochs

ABSTRACT

Conventional operating system architecture is the enwrap structure, which system functions are achieved with many of the software components. The disadvantage of this kind of architecture is the dependability of the system would be counteracted with the system complexity. State-based operating system architecture and the management of operating system with control theoretics were designed to give a new resolvent with this problem.

With the Octopus OS, the problem of resource scheme is the key to the system performance and self-evolvement. In this paper ,we study the state-based operating system and structure of Octopus OS, and put on the requirement of scheduling for Octopus OS, then pointing the requirement, we present the method to solve them. due to their unforecastable, traditional scheduling replace by the new strategy which is based on feedback. To achieve reasonable distribution of system resource and maximize utility of system resource used, this algorithm uses a pricing model from economics while resources are scarce. Next, The algorithm is simulated and analyzed in the experiments. In the end, we discuss how to build the environment for simulating Octopus OS on Bochs and critical technology for process scheduling on X86 architecture, and then describe the important data struct and implement.

Keywords: autonomous computing OS scheduling feedback control Pricing model Bochs

创新性声明

本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果；也不包含为获得西安电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中做了明确的说明并表示了谢意。

申请学位论文与资料若有不实之处，本人承担一切相关责任。

本人签名：闫国　　日期 2006.2.25

关于论文使用授权的说明

本人完全了解西安电子科技大学有关保留和使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属西安电子科技大学。本人保证毕业离校后，发表论文或使用论文工作成果时署名单位仍然为西安电子科技大学。学校有权保留送交论文的复印件，允许查阅和借阅论文；学校可以公布论文的全部或部分内容，可以允许采用影印、缩印或其它复制手段保存论文。（保密的论文在解密后遵守此规定）

本学位论文属于保密在___年解密后适用本授权书。

本人签名：闫国　　日期 2006.2.25

导师签名：　　日期 2006.2.25

第一章 绪论

1.1 引言

在过去的几十年中，随着计算机工业的飞速发展，各行业对计算机的依赖逐渐增加，在这一过程中关于可信处理系统（dependability computing system）的研究日益受到各研究机构的关注。不同的应用领域对各种计算机系统的性能、可靠性、安全性等要求不尽相同，但总体说来，人们希望它们能提供相对更高可信度的服务。操作系统是计算系统的灵魂，它在这一期望中担负着不可推卸的责任。

操作系统被看作是计算机硬件和上层应用之间桥梁，它不仅抽象、优化硬件的使用，而且为上层用户提供服务。根据对系统实时性的要求操作系统被分为分时系统和实时系统，分时系统主要为应用提供便捷的服务，这种操作系统的可信度取决于系统平均响应时间，即能否在单位时间为更多的用户请求提供服务；而实时系统，因为主要应用于如工业制造、信息通信、交通、军事、航空航天等重要领域，系统提交可信服务的能力对人类生命财产的保障、生产生活的质量产生至关重要的影响，因此这种系统中要求平均响应的同时，单个任务响应时间则更为重要[10]。

最初的实时操作系统小而简单，只是带有一定专用性的软件，为用户提供系统初始化的管理和简单的时钟功能。此后，随着上层应用的不断发展，为了满足更多应用的需要，开始出现应用于特定硬件的专用实时操作系统[11]。这些系统虽然满足了一部分应用的迫切需求，但存在很多的局限性如：特定的应用环境和设备要求使得难于移植和重用、很高的开发成本投入和很短的生命周期。但实时上，这些专用的实时操作系统中，除了面向特定硬件和特定应用接口等部分的功能，还有很大一部分多任务功能如：时钟管理、基于优先级的调度、同步/互斥通信等机制是基本相同的。于是，实时操作系统的发展出现了和分时系统殊途同归的局面，即其体系结构基本采用了同一组织模式，无论实时系统还是分时系统，其系统结构都采用层层包裹的虚拟机模式；在功能实现上，采用软件组件的模块化结构。因此随着计算机硬件性能的提高以及上层用户需求的多样化系统随之膨胀，这样从操作系统自身的发展来讲，分时系统和实时系统同时面临着很多的问题：如资源管理困难、功能模块间关系的复杂化、内核规模增加导致系统可靠性降低等，这些问题都日益成为操作系统发展迫切需要解决的问题，正是这些问题的存在导致了目前计算系统软件发展落后于硬件的局面，也使得对操作系统结构的研究和改进日益受到重视[4]。

1.2 项目背景

1.2.1 传统操作系统结构问题

传统的观点将操作系统看成一个虚拟机（virtual machine），而操作系统的主要任务和功能是资源管理。事实上，这种观点容易使人对操作系统的认识产生错觉：认为操作系统研究需要解决的问题的本质是如何进行系统抽象和如何优化资源调度，该思想集中体现在对操作系统内核的研究上。但真正推动操作系统发展的动力是底层硬件和上层的应用，而最终的目的是响应上层应用提出的资源管理和使用请求，为上层应用提供高质量的服务，因此无论是对于系统抽象还是资源的管理都是应上层应用的需求产生的。

纵观操作系统发展历史，从操作系统内核结构的角度可以将其分为三种内核结构模式[1]：巨内核结构（Monolithic Architecture）、微内核结构（Micro-kernel Architecture）、扩展内核结构（Extensible kernel Architecture）。

巨内核体系结构最初的目的是为了使开发人员能够快速的开发出高性能的应用，这种内核对底层硬件提供较高层次的系统抽象，大部分系统功能集成于内核中，并定义统一的应用接口，这样的系统有较好的通用性和执行效率。然而正是这些接口，使得这种体系构架的灵活性降低，导致无法适应特定的应用。随着计算机系统应用领域的迅速扩展，用户应用的多样化和定制化问题会越来越突出。另外，从内核开发者的角度而言，巨内核由于各内部模块交互的复杂性导致维护的困难，虽然理论上可以为各模块定义良好的接口，使得模块之间的交互完全通过接口完成，但是由于各模块的交互太复杂，很难做到完全封装，并保证正确性。

由于巨内核存在的缺陷，促使研究人员提出微内核的结构。微内核的思想是资源核（resource kernel），它的设计理念在于分离系统的设计目标和它的实现机制，内核本身除了对硬件设备提供基本抽象之外为上层提供的抽象越少越好，大部分的系统功能由用户进行更高一级的设计，以满足不用应用的需求。这样使内核体积变小，从而获得较好的可伸缩性。这种方式看起来解决了通用性与灵活性之间的矛盾，但仍然存在缺陷。首先，最小抽象只是一种理想的状态，因为内核进程运行的安全性和效率上的优势，上层应用总是希望更多系统相关的功能在内核态下运行，这样在设计和实现的过程中，越来越多的功能又被集成于内核中，所谓的微内核也不由自主的向巨内核接近；其次，由于许多功能是在内核以外实现的，那么应用和内核的通信将会非常的频繁，这加大了系统的整体开销，从而降低了系统整体运行的可靠性。

扩展内核的结构是在微内核/资源核的基础上产生的，这种内核按照接口划分为三种形式：1）硬件资源接口集，这种方法认为系统不论向用户提供哪种接口，

总是存在某些应用使得提供的接口不能很好的适应这些应用的要求，所以内核直接将底层的硬件资源提供给上层应用，该种内核的唯一职责就是对底层硬件资源进行安全的复用。最典型的例子就是 exokernel 和 Ageis；2）固定接口集，这种内核向上层提供一个抽象接口集合，用户可以通过这些集合实现特定应用的定制，最典型的是 SPIN 和 cache kernel，如 SPIN，用户可以将自己的代码通过接口动态的加入到内核中；3）元接口集，这种内核提供一个元抽象接口层，用户可以使用这些元抽象接口实现自己需要的接口，以实现功能的定制。扩展内核的形式将微内核的问题具体化，似乎解决了本节开始所提出的应用多样化和定制化的问题，但实际上这种思想并没有广泛的应用于操作系统的设计中，因为扩展内核需要重新设计接口和运行机制。而且随着硬件价格的大幅度下降，使得扩展内核当初的设计理念之一："用尽可能少的硬件，做尽可能多的事"发生了动摇。

1.2.2 项目研究工作

对于上文提到的三种操作系统内核结构，无论哪一种思想和方法，一般都认为操作系统是包括在硬件设备和用户之间的软件体系；在其内部各系统功能用不同的功能模块实现；模块之间、操作系统内外部分使用某一种通信机制进行关联。如图 1.1 所示，这种虚拟机的体系结构的最大的特点是封闭和隔离，各系统功能、系统状态及相关参数被严格保护，上层应用对系统功能的使用以及系统底层功能的相互调用都要通过接口实现。这种模式最大限度的保证了操作系统的底层结构安全可靠，但不同层次之间定义严格、固定且唯一的接口也造成了系统运行的瓶颈。用户应用的多样性要求系统抽象越细致越好，这导致接口越来越多并且越来越复杂，系统功能模块之间相互的通信也越来越频繁，从而使得系统结构也越来越复杂，而复杂性的提高必然导致可靠性和性能的相对下降。

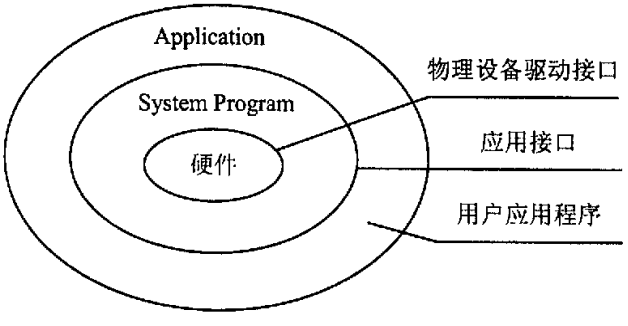


图 1.1 传统计算系统的虚拟机体系结构

针对目前操作系统体系结构存在的可靠性和安全性下降的隐患，西安电子科

技大学软件研究所十五国防预研项目“可信[1]软件构建技术及构建库管理系统”项目组认为这一现象的根源在于操作系统内部功能模块自主管理的体系结构，并提出了基于状态的操作系统体系构架（State-Based OS Architecture）。该结构将系统一部分重要状态数据和状态控制从结构上进行分离，这些重要的状态放在共享内存中；以状态为驱动的资源调度策略建立在这种结构的基础之上。根据 State-Based OS Architecture 项目组构造了一个具体的操作系统——Octopus OS，该系统针对传统操作系统层层包裹的虚拟机结构导致系统的可靠性降低的弊端，效仿章鱼的生理结构，采用各进程自治/闭环的方式执行，任务之间的通信通过共享的数据来实现以避免不同层次数据通信所带来的安全隐患，并提供多个监控进程监控重要状态的形式实现系统的管理。

1.3 论文主要工作和内容安排

本文作者从构建可信系统的角度出发，研究了关于容错的相关内容，并从提高操作系统可靠性出发，对前文所述操作系统的发展以及操作系统调度的现状进行了分析，把握基于状态操作系统结构的核心，提出了适合于 Octopus OS 的调度算法。

文章第二章介绍了自治系统相关的概念、分析了自治操作系统结构出现的原因，并描述了项目组提出的 State-Based OS Architecture，在此基础之上，介绍了 Octopus OS 体系结构及实现方式。

第三章在分析传统调度算法现状的基础之上，提出了 Octopus OS 调度的两点需求，并针对这样的需求提出了相应的解决方案：对于调度策略单一的问题采用闭环的方式实时反馈系统执行的状态以判断所选择的执行策略；针对任务自治执行的特性，在需要选择任务执行策略优化时，采用竞价调度模型。

第四章在前一章讨论的基础之上，提出了应用于 Octopus OS 的调度算法的框架，并给出了具体的实现方式：监控器实时监控系统执行的状态，根据反馈值来判断系统下一步的执行策略，当资源充足时，采用通用调度策略；资源缺乏时，采用竞价调度。与此同时利用 Matlab 仿真讨论了调度算法的优劣，并分析了该算法的不足和可扩展的部分。

第五章描述了基于 X86 平台 Octopus OS 的部分实现，系统选择 Bochs 虚拟机进行模拟，文中主要讨论了硬件平台对进程控制的支持与限制以及相关数据结构的设计与组织。

---

[1] 可信（dependability）最初概念为有能力提供一种服务，这种服务基于某种合理的理由被信任（trusted），而当前如果一个服务被认定为可信的，系统要有能力受避免在可接受范围内的故障（failures）的发生[2]

第二章 基于状态的自治操作系统——Octopus OS

2.1 基于状态的自治系统结构

2.1.1 自治系统

目前，由于很多计算机系统为了完成越来越多，甚至更加复杂的任务，导致了系统本身的复杂性增加，系统复杂性的升高造成系统设计、维护的代价就越来越大，同时造成对系统理解的困难。尤其在很多关键任务（mission critical）系统中，对系统的维护往往要求在极短的时间里定位系统故障并采取正确的处理措施。鉴于这种趋势，很多研究机构都认为目前计算机系统应该向智能化的方向发展，它们应该更多的进行自我管理（自我管理包括如自我配置、自我诊断和治疗、自我优化以及自我保护等行为），从而将研究的重点集中在了自适应和自治系统的研究上。

自治系统（Autonomic computing）概念由 IBM 提出[3]，这个词来源于自主神经系统（autonomic nervous system）。自主神经系统是生物学概念，该系统将生物的意识大脑从那些必须进行的、生死攸关，但是很低级生理活动（如：呼吸、心跳、神经末梢反射等低级神经反射控制）中解放出来。自治系统概念的本意在于希望计算系统能够从那些大量的系统日常管理和操作任务中解放出来，从而将更多的精力和时间放在系统的核心业务上。但自治计算系统的基础必然是系统意识级别的自适应操作系统。

2.1.2 自治操作系统体系结构的出现

自治系统的概念早期多应用于军事控制系统，通常事先设定若干不同场景和对应策略，系统根据执行过程中对现实环境的检测和评价与预置的场景进行匹配，从而调整系统自身行为以达到对应不同情况的目的。后来随着计算机系统在商业领域的应用，使得自适应系统这种策略被专家系统、决策系统、客户管理系统等趋于智能分析类的软件广泛应用[4]。但这类自适应系统存在很多的问题：这种系统的自适应性通常通过对结果起决定性作用的参数的选择和设定来实现，而这些参数时建立在应用人员的业务知识之上的，并非真正的自适应；并且该类系统对自身适应性没有评估和改善的能力，在系统适应性低效的情况下，除非有人为干预，否则该系统的低效性不能改变；这样的系统也没有自我修复能力，一旦系统出现人为（参数设置失误等情况）故障或系统不可测故障，必须在外界干预情况下才能恢复。

一些研究机构在对上述问题进行分析之后发现，对于有些适应性策略，可以在小的范围内进行适应性评价，当应用认可其效果后可将这一策略推广至整个系统范围以获得更大收益。但一旦适应性策略应用范围扩大，则用于适应性管理的开销必然会增加，适应性管理也更加复杂。因而自适应系统出现了自身发展的瓶颈问题。一部分研究人员认为造成这一现象的根源在于自适应系统从一开始就是建立在传统操作系统体系结构上的。传统操作系统体系结构的缺陷在于软件模块的功能性划分，这种结构的系统中一个功能由一个软件模块实现，系统的可靠性和效率提高只能通过调度优化和功能冗余来实现。这对于自适应系统的适应性控制而言其性能显然是难以令人满意的。在这种观点下，自适应系统发展出现了新的思路：操作系统体系结构的自适应变化。

2.1.3 基于状态的（state-based）操作系统体系结构

第一章指出传统的操作的可靠性和效率低是由系统模块化结构的封闭性、复杂性造成的。各系统信息由系统中的模块封闭并分散的管理。模块间缺少全局观的控制和联系，它们之间的关联和协调是完全自主的，在不可测的情况下，协调方中任意一个成员的异常行为都是被动显示。另外，对于像数据库应用中事务级的任务，这种方式无法满足数据应用对于数据完整性、一致性的需求。项目组认为解决这些问题的关键在于操作系统必须将某些对上层应用至关重要的信息进行统一组织管理，开放给应用进程。由此，提出了基于状态的（state-based）操作系统体系结构，指出用状态方式对操作系统信息进行描述，采用反馈控制（feed control）理论思想对系统状态进行管理。这种系统体系结构如图 2.1 所示[5]。

其中系统状态（state）图由操作系统部分重要信息和参数组成，状态组织结构可以部分描述当前系统运行情况，这些重要的状态放在共享内存中。系统内进程间相互关系用一定的数据结构进行描述，通过对系统状态的监视和观察可以了解用户应用的执行情况，对系统状态的控制可以保证用户应用执行的完整性和可靠性。**状态信息**不仅是状态监视器可见的，也是应用进程可见的，进程可以通过状态信息读取实现通信功能；**控制器**用于分析用户的应用需求并实现程序静态分析，设定系统内部相关的初始状态；**监视器**负责系统的全局状态监测和控制，并对系统状态进行评估，定位系统错误，实现系统的容错；**状态改变代理**在非信任进程进行某种操作触发系统状态迁移时检查其操作的合法性，以实现对系统当前相关场景进行保护。如果监控器发现某一个进程状态修改行为造成系统向非法状态迁移，系统将根据被保护的场景进行回滚操作。

系统中信任进程通常为系统控制的进程，非信任进程为用户应用进程。这种结构与传统的操作系统相比，提高了可靠性，它通过监控器和状态代理来实现系

统重要状态的全局监控，为事务级的应用提供良好的支持；同时将原本封装的一些信息开放给上层，而不再通过接口；由于采用了共享内存来存放重要的状态信息，因此减少了进程间通信的开销。

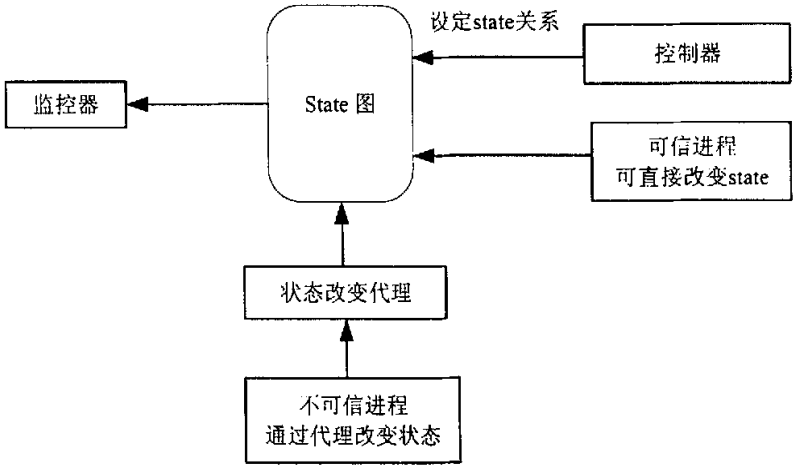


图 2.1 Stated-based 操作系统体系结构

2.2 Octopus OS 简介

Octopus OS[6]是对基于状态操作系统体系结构的实例化。整个系统采用自治的方式实现，在执行的过程中实时监控系统故障，实现系统与应用进程的容错，从而达到系统运行的可靠。

传统操作系统为了方便重用，在设计上非常强调层次化，以为用户抽象一个统一接口的虚拟机。这样的体系结构在通用系统中已经应用的非常成功，但是对于那些可靠性要求非常高的系统如军用、航空等领域，因为这些系统通常面临着十分苛刻的指标、多变的环境，不但要求应用，而且要求操作系统本身达到高可靠性，同时需要系统本身具有良好的错误监测和恢复的能力。而传统操作系统则无法满足，原因主要有两点：系统重要状态数据分散在系统的各层之中，系统各模块之间通过带参数的调用进行交互，导致操作系统各个模块之间的控制流和数据流非常复杂，增加了系统的耦合度，从而增加了系统的错误诊断和错误恢复的难度；由于系统监控进程属于应用的一部分，这样导致应用程序监控自己的局面，一旦应用监控进程自己崩溃，系统整体就会崩溃。

针对上述问题，项目组根据基于状态操作系统体系结构提出一个高可靠的自治操作系统——Octopus OS，该结构模仿章鱼的生理行为，如图 2.2 所示。其中应用本身就像是章鱼的触角，而系统的监控进程好比章鱼的大脑，共享内存为章鱼

神经细胞。对于一些小的错误，应用本身可以独立的处理并自己实现错误的恢复；而对于超出自身控制能力范围内的错误，则通过监控进程来实现。这些控制过程好比章鱼的行为，当章鱼的触角受到小的伤害如擦伤，它会自己治愈；而当其碰到了大的问题，如遇到了天敌或者障碍物，则章鱼的大脑将协调所有的触角来避免大的伤害，或者绕行。

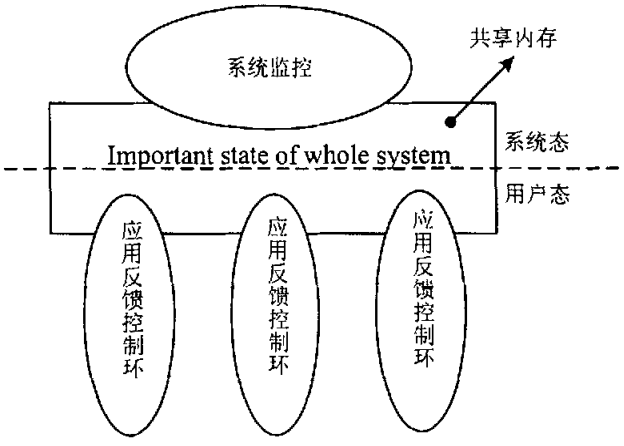


图 2.2 Octopus OS 框架

在 Octopus OS 中操作系统和应用被看作一个整体，系统中所有重要状态使用共享内存进行统一管理，而整个系统的控制由监控器实现；系统的监控进程工作在操作系统特权级上，以提高系统的安全性和可靠性，此处也可以考虑采用多版本，使用多个监控进程，以达到系统监控的安全和可靠。

Octopus OS 的设计采用两层闭环反馈的方式，系统的实现结构如图 2.3 所示。可以看出不论监控进程还是用户应用进程均采用闭环反馈的方式，它们实现方式如下：

应用进程：每个应用进程执行过程分两个步骤。首先，实时地采样任务运行时切片（profiler：任务运行时某一时间点的静态信息），从切片中获得的反馈信息包括两种：1）应用和应用之间交互所用到的数据；2）用于每一个任务的重要的健康指标对应的反馈值。对于第一类数据实现应用之间的交互，它被放到共享内存中；而第二类用于完成系统的容错，这类信息又分为用于用户自己的错误监测和恢复的数据以及进程该反馈点重要的健康指标，前者放在私有内存中，后者放在共享内存中运行时状态模型。接着，利用上一步收集的用于错误检测和恢复的数据与预先设定的指标进行比较，根据比较的结果控制器调整下一个采样周期任务的执行状态，必要时实现错误的恢复。

监控进程：监控进程周期的采集共享内存中任务的运行时状态模型，并将采样的结果与全局的健康指标进行比较，根据比较的结果，监控进程的控制器将通过传递消息给用户进程控制器以实现系统整体的控制与错误恢复。

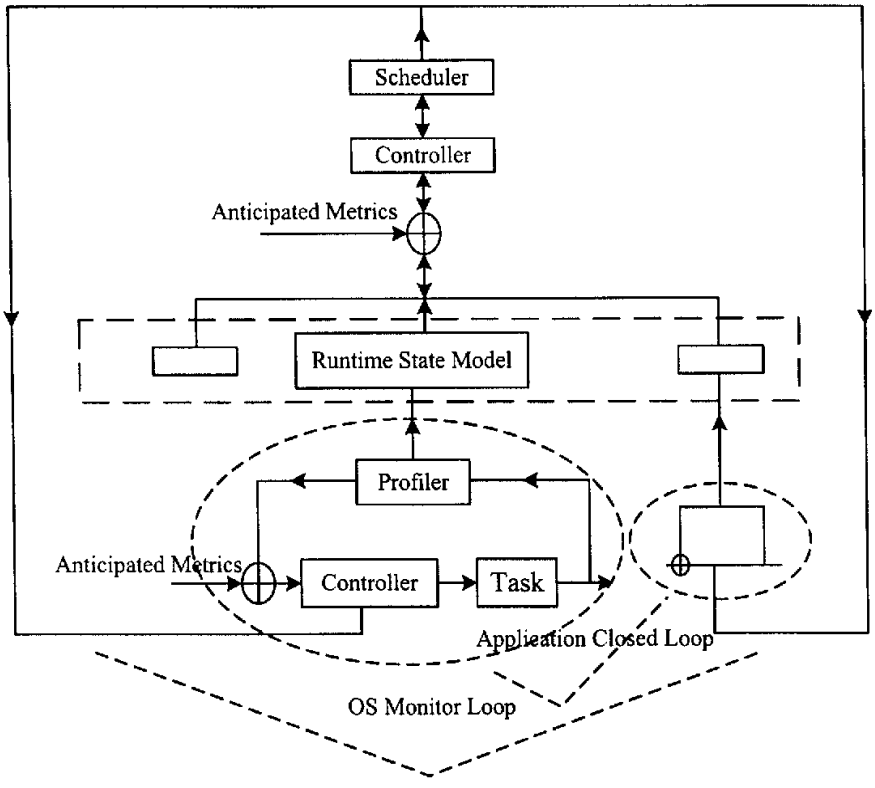


图 2.3 Octopus OS 实现的结构

2.3 本章小结

本章首先通过讨论自治系统的特性，引出了自治操作系统体系结构出现的原因，进而介绍了项目组提出的基于状态操作系统体系结构，并在上述讨论的基础之上，介绍了 Octopus OS 的结构。Octopus OS 是基于状态操作系统体系结构的实例化，它从系统结构体本身出发采用自治的方式执行，并统一管理重要的状态以实现系统的容错从而实现系统和应用运行的可靠性。基于本章的讨论，下一章将根据 Octopus OS 的特性分析其系统调度，并提出相应的调度方案。

第三章 Octopus OS 调度算法的设计

系统的调度通常是一个操作系统的关键的部分之一，调度算法的优劣将会影响到整个系统的性能，同样，对于 Octopus OS 而言调度也很重要，它是解决系统性能和自适应能力的关键。本章将在分析传统调度算法的基础之上，讨论 Octopus OS 调度的需求，并为其提出一个适宜的调度策略。

3.1传统的任务调度算法

任务能够被调度的前提是该任务拥有了执行所需的资源，这类资源包括 CPU、内存、外设、带宽等，因此通常将任务的调度看作是系统资源的分配和管理。其中使用最多的资源为 CPU 时间和内存，由于目前虚拟内存技术非常成熟以及硬件技术的更新，内存的需求并不是那么迫切，但是程序最终的执行是当其获得 CPU 时间后，因此如何决定在所有进程之间分配 CPU 资源则成为调度算法的关键，并且目前很多操作系统对调度算法的讨论也都是基于 CPU 时间。

3.1.1 调度算法的设计准则

操作系统的调度算法随着硬件的发展逐渐复杂化。早期的批处理时代，调度算法非常简单，只要顺序的执行磁带上的任务。当硬件性能提高后出现了多道程序系统和分时系统，为了充分利用 CPU 资源，采用了复杂的调度策略。如何选取调度策略，将影响到整个系统的性能，同时 CPU 调度策略应具有相应的性质以满足系统进程的特性。通常评测一个调度算法考虑到如下准则[10][11]：

1. 公平：确保每个进程获得合理的 CPU 份额；
2. CPU 利用率：通常希望尽可能的保持 CPU 忙碌。如实时系统中，CPU 利用率应该在 40%到 90%之间[12]；
3. 吞吐量：单位时间内执行完的进程数被称为系统的吞吐量。通常根据需求来控制系统的吞吐量；
4. 周转时间：一个进程执行所需要的时间通常是评测系统调度的一个重要指标，从进程提交到进程执行完的时间间隔为周转时间。即周转时间等于：在就绪队列中等待的时间、在 CPU 中执行的时间和 I/O 操作的时间之和，该标准受限于输出设备的速度；
5. 等待时间：通常意义的等待时间是一个执行的任务除了占用 CPU 时间外的其它时间，但由于调度算法并不能影响 I/O 操作时间和其它进程的执行时间，因而等待时间为进程在就绪队列中所耗费时间的总和；

6. 响应时间：在交互式系统中，周转时间并不是最好的指标。进程通常给用户输出一个计算的结果，并继续执行计算。因此使用另一个度量标准——响应时间，它计算从进程提交请求到产生首次响应的时间。

在实时系统中，应用程序的执行结果的正确性不仅与计算的逻辑结果有关，而且与结果产生的时间相关，即数据的处理应在一定的截止期限内完成。因此在实时应用越来越广泛的情况下，调度策略还应考虑

7. 实时性：需要任务在截止期限内完成

通常在设计一个调度算法时要尽可能的兼顾上述的一些原则，但又可能出现冲突。设计者希望最大化 CPU 利用率和吞吐量，最小化周转时间、等待时间、响应时间，在大多数情况下求得一个平均的性能，但是在有些特殊的情况下则需要最优化最大或者最小值。如为了保证用户获得满意的服务，需要最小化“最大响应时间”。

3.1.2 传统的调度策略

系统任务的调度分为非剥夺方式（nonpreemptive）和可剥夺方式（preemptive）对于前者，规定正在执行的进程不能被抢占，直到该进程结束或者发生某事件而阻塞时，才把 CPU 分配给另一个进程。这种方式的调度算法简单且易于实现，但是它通常不够灵活，并不适合于具有多个竞争用户的通用系统，在实时系统中这种方式则比较合理。可剥夺的调度方式则允许将逻辑上可继续执行的进程暂时挂起，并基于某种原则，如优先级原则、短作业优先原则、时间片原则等选择一个进程执行，剥夺方式虽然灵活，但将导致竞争条件的发生，由此也产生了信号量、消息等其它复杂的进程控制理论和方法，尽管如此目前绝大多数的操作系统也都采用剥夺的方式。

3.1.2.1 现代通用系统调度策略

为了适应不同环境，人们提出过多种调度策略，如先来先服务、短作业优先、时间片轮转、优先级调度、多重队列调度、彩票调度等算法。其中最通用的为基于优先级调度和时间片轮转的方式。

优先级调度算法中，为每个进程赋予一个优先级，CPU 总是执行优先级最高的任务，优先级相同时采用先来先服务的策略。该调度策略关键之处为如何定义优先级权值以及优先级的分配，这些值将受系统中进程类型、就绪队列任务数等参数的限制。优先级调度可以是可剥夺或者不可剥夺的：当一个进程到达就绪队列时，在可剥夺方式下系统将其优先级与当前运行进程的优先级比较，如果新进程优先级高于当前任务，则切换上下文，CPU 执行新进程；不可剥夺方式则是简

单的把新进程放到就绪队列中合适的位置。

时间片轮转的方式是一种最简单、最公平的调度。每个进程被分配一个该轮次允许运行的时间段，称作时间片。该策略的关键之处在于时间片长短的设定，如果时间片设的太短，会导致过多的进程切换，降低了 CPU 的效率；而设的太长可能会引起对短的交互请求的响应变差，通常这个值受系统中上下文切换时间的约束，而上下文切换又受限于硬件设备的性能。时间片轮转调度是可剥夺调度：如果在时间片结束时进程还在运行，则 CPU 将被剥夺并分配给另一个进程，如果进程在时间片结束前阻塞或结束，则 CPU 当即切换。

3.1.2.2 实时系统调度

实时系统是那些时间因素非常关键的系统，前文已经提到系统运行结果的正确性不仅需要逻辑结果的正确，而且需要任务在某个时限内完成，否则应用程序执行的结果将毫无意义。实时系统通常分为硬实时和软实时，前者对时间的要求非常严格必须满足任务的时限，而后者偶尔超时也是可以容忍的，因而实时系统的调度算法需要较高的可预测性。

通常考察实时调度算法的方法取决于[12]：1）一个系统是否执行可调度性分析；2）算法是静态的还是动态的。基于这两方面的约束，主要存在的调度策略为静态驱动表和优先级抢占。静态表驱动调度在系统运行前根据各任务的实时要求构造一张任务运行时间表，这张表指明各任务的起始运行时间和运行长度，在运行时调度器只需根据该表在指定的时刻启动相应的任务即可。由于时间表是系统运行前静态生成的，因此可以采用较复杂的搜索算法找到较优的调度方案，但该方式缺乏灵活性，需求一旦发生变化，时间表需要重新生成。这种方式因为具有非常好的可预测性，主要用于航空航天、军事等对系统的实时性要求十分严格的领域。

优先级抢占调度方式与通用操作系统中采用的基于优先级的调度策略基本类似，任务优先级根据一定的原则定制。如在实时系统中应用的非常广泛的 DEF（最早时限优先）调度，该算法的基本思想是：当一个事件发生时，对应的进程就被加入到就绪队列之中，该队列是根据任务截止期进行排序的优先级队列。对于这样的调度主要应用于一些简单、独立、对时间要求不高的嵌入式系统。但随着调度理论的不断成熟和完善，这种方式也会逐渐在一些对实时性要求十分严格的领域中得到应用，目前许多实时操作系统采用的都是这种调度方式。

3.2 Octopus OS 调度算法的需求分析及解决方案

3.2.1 Octopus OS 调度算法的需求

在 Octopus OS 中采用了自治方式实现系统的容错控制和管理，目的是在对可靠性要求非常高的如军事、航天等领域的系统中实现应用和系统执的高可靠性。通常这些领域运行的系统没有人为干预，同时运行的环境是不可预测的，此时资源的管理和分配将会影响到系统的性能，如果调度策略选择不当，可能导致系统崩溃，而无法恢复，因此一个好的调度策略是 Octopus OS 系统的主要任务之一。虽然传统的调度策略已经成功的应用于多种类型系统的任务调度，但是如上文中所提到的调度方案并不适合 Octopus OS 这样的系统，原因主要有两点：

1. 传统的调度策略均为开环的，但对于特定情况尤其是实时任务的调度是建立在复杂的任务特性之上，如任务 deadline、优先级限制、共享资源等，一旦开始任务的调度，就不能改变调度的方式，因此由于系统执行环境的不可预测性，开环调度通常会使一些关键任务不能及时执行，导致发生 miss deadline；同时为了任务被有效的调度，往往需要降低 CPU 的使用率，这样就不可避免的存在系统仍然有空闲资源却发生任务 miss deadline 的情况，甚至是系统的崩溃；
2. 在 Octopus OS 中每一个任务是自治执行的，他们对资源的请求是可控制的，执行也是相对自主的，任务通过判断自身执行时所处的状态提出对资源的请求，如当资源紧缺时，可以根据当前的状况提出多种更少的资源请求量，以满足自身执行所需的资源保障或者放弃，同时某个任务对资源请求的放弃也为系统的恢复、容错提供了时间与空间的保证。因此，为了达到资源的合理分配，需要调度程序提供更进一步的优化选择策略，即根据系统当前实际的资源量和任务的多种资源请求量，以某种原则，选择最优的任务组，实现系统中任务被公平、可靠、有效的调度。而传统的系统调度只是简单的采用某种规则从就绪队列中选择将被执行的任务，无法满足 Octopus OS 调度目标。

3.2.2 闭环方式的任务调度（针对问题一）

上文提到传统的调度无法满足Octopus OS的调度需求的问题一为：因为它们都是开环的。针对这一问题我们考虑采用闭环的方式来实现系统调度。

3.2.2.1 反馈控制模型

控制理论是自动化的基础，而自动化又将人类从繁重的甚至是危险的劳动中解放出来。对于一个实际的系统，由于其内部非线性机理的复杂性和外部环境的

不确定性，使得人们仅凭自己的感觉或经验无法达到高精度的控制要求，而必须依靠理论的指导。这种理论又必须能给出最优或者满意的控制策略使系统达到一个理想的指标。因此，在控制理论中需要建立系统的模型，但由于实际系统的复杂性，往往不能从基本的系统特性直接推导出准确的模型，而必须利用系统的输入和输出数据的反馈，这就形成了反馈控制理论。一个基本的反馈控制模型如图 3.1 所示：

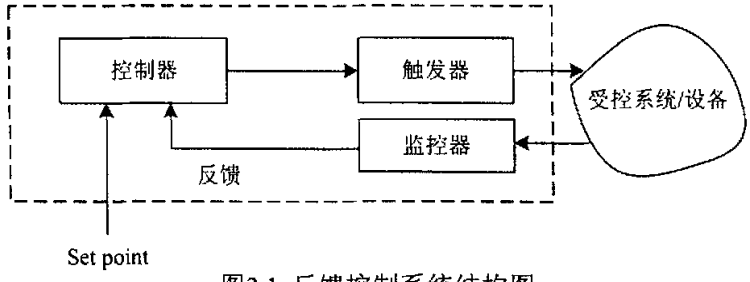


图3.1 反馈控制系统结构图

一个反馈控制系统由控制器、触发器、监控器三部分组成。控制过程涉及到 controlled variable（被观测变量）、set point 和 manipulated variable（经过调整的数据）。其中，controlled variable 由监控器监测到相关参数，set point 用于评测 controlled variable 的正确性，如果发生了偏差或者错误，则经过调整获得 manipulated variable。反馈控制系统经过如下步骤实现：

1. 系统周期的监控 controlled variable，将获得的结果与 set point 进行比较，如果没有错误则跳转到步骤 3，否则，执行步骤 2；
2. 控制器通过基于错误的控制函数算得正确的或者理想的输入值 manipulated variable；
3. 触发器调整控制器的输出值为适合的输入参数，传递到受控系统/设备。

可以看出，基于错误的控制函数设计的好坏将影响到系统运行结果的稳定性与可靠性。在实际的系统中，应用最为广泛的控制函数为比例（Proportional）、积分（Integral）、微分（Derivative）控制调节函数，采用 PID 调节的控制器称为 PID 控制器[13]。通过 PID 调节函数后，输入的误差值将被调整为一个理想的输出，其中比例控制项 P 表示控制器的输出值与输入误差成正比例关系；积分项 I 表示控制器的输出与输入误差值的积分成比例关系，该项用于消除系统执行过程中稳态误差；微分项 D 表示控制器的输出值与输入误差的微分（即误差的变化率）成正比例关系，它用于克服系统误差调节过程中，可能出现的振荡甚至失稳现象。PID 调节以其结构简单、稳定性好、工作可靠、调整方便而称为工业控制的主要技术之一，其基本表达式如公式 3.1 所示[13]

$$y = K_p(\varepsilon + \frac{1}{T_i}\int_0^t \varepsilon dt + T_d \frac{d\varepsilon}{dt}) \qquad \text{式 3.1}$$

其中 y 表示控制器的输出，K_p 为比例系数，T_i 为积分常数，T_d 为微分常数，ε 为误差。公式 3.1 是连续时间 PID 调节方程，但在实际的计算机处理过程中，需要离散的采样系统运行信息，以被计算机控制处理，通常应用计算机实现 PID 控制时，采用公式 3.2 的递推算法[14]：

$$y(n) = K_p[\varepsilon(n) + \frac{K_I\tau}{T_i}\sum_{i=1}^{n}\varepsilon(i) + \frac{T_d}{\tau}(\varepsilon(n-1))] \qquad \text{式 3.2}$$

τ 为采样周期，K_I 为逻辑控制控制系数，它的值为 1 或 0，当 K_I =0 时，表明积分项不起作用。

3.2.2.2 任务调度采用反馈控制模型

Stankovic等人为了实现在不可预测环境下的实时调度性能的保证，采用了将反馈控制技术与实时调度技术相结合的方法，构造了一个基于PID反馈控制的调度体系结构，解决了不可预测环境下CPU实时调度的问题[15]。这种方式完全借鉴了工业控制中的理论模型，通过判断系统当前miss deadline的情况，用MissRatio(t)表示，以决定是否需要控制器来调整系统中当前的执行队列，控制器中存放着事先定义好的PID调节函数。这种反馈控制的思路为：系统刚开始运行时，设定一个采样周期内分配给所有任务的CPU时间数，根据这个值，控制就绪队列中的任务；随后，控制器周期的监控系统当前MissRatio(t)，并将MissRatio(t)与set point点设定的$MissRatio_s$相比较，以决定下一个采样周期内任务对CPU分配的调整，从而调整执行的任务。CPU时间的调整通过公式3.3实现：

$$\Delta CPU(t) = C_p error(t) + C_I \sum_{IW} error(t) + C_D \frac{error(t) - error(t-DW)}{DW} \qquad \text{式 3.3}$$

其中，$error(t) = MissRatio_s - MissRatio(t)$，$C_P$，$C_I$，$C_D$，DW均为PID控制器的可调参数。可以看出，该CPU调整函数用到了比例、积分、微分三项。Stankovic使用PID控制器方式实现任务调度是考虑到任务调度不可预测性而需要动态调节的方式与工业控制的特性基本类似，并且PID控制不需要受控系统的精确的分析模型。虽然这种方式有效的解决了由于CPU资源匮乏引起的miss deadline，但是对于Octopus OS并不完全适用：

1. 如果PID函数在设计时不够理想，并且比例、积分、微分参数设置的不合理，会导致反馈系统（即调度控制部分）的震荡，而反馈控制的稳定也需要经过一段时间才能实现，在这个过程中可能导致系统相关的损失，使得系统可靠性降低。因此，在Octopus OS中完全采用PID的调度方式，显然不合理；
2. 上文中提到的PID调度的方式只是针对CPU资源的分配和控制，但是在一些恶劣的情况下，系统不只仅仅要考虑CPU资源，而且还要受内存、系统电能等资源缺乏的困扰，如果同时考虑这些资源势必复杂化PID函数从而导致1所提到的后果；

虽然Stankovic提出的采用PID的控制方式并不能完全满足Octopus OS的调度需求，但我们将借鉴这样的调度思路，为调度算法引入控制器，根据系统执行情况，实时的获取相关的参数，以判断资源的使用状况，判断此后系统中执行的任务，从而避免了系统执行过程中不可预测性带来的弊端。具体的实现将在第四章中讨论。

控制器获取了系统当前相关参数后，调度器需要选择就绪队列中将被执行的任务，当资源缺乏需要实现不同任务的不同资源请求的优化调度时，传统的简单的调度策略并不能满足，下一节将讨论面对这样的需求时如何做出这样的选择以实现优化。

3.2.3 引入竞价资源分配模型（针对问题二）

传统的调度无法满足 Octopus OS 的系统调度需求的问题二为：传统调度的选择策略简单，在资源缺乏的情况下无法优化系统任务的执行。针对这一问题我们考虑引入微观经济学中的竞价模型来实现资源紧缺情况下任务的调度。

3.2.3.1 微观经济学中的相关概念

基本概念：微观经济学主要研究个体和比较固定的团体的经济行为，它集中于对单个价格和市场以及特定资源在具体使用中的配置的分析。其中欲望指消费者想拥有某种物品的愿望；需求指消费者欲望驱动下、以自身所拥有货币的约束下，做出的一种选择。而需求的形成受到价格、收入和偏好的限制。其中收入是消费者拥有货币的能力；价格是物品价值的货币体现，价格的高低反应了该种资源的稀缺程度；偏好则反映出消费者对物品的喜欢程度或主观评价。消费者对某些物品的喜欢程度越高，表明他对该物品的偏好越大，但由于物品有限，要想得到该物品，则要付出相应的货币。当消费者手中的钱不能完全满足自己的对物品的欲望时，他会根据实际情况，来减少对物品购买，以控制欲望。

影子价格：在微观经济学中存在一个价格的概念——影子价格（shadow price），

它不同于生产价格，生产价格体现了商品的价值，而影子价格是随市场供求关系的变化而变化的，当市场资源缺乏，影子价格会升高，反之会降低。

效用函数：在微观经济学中，我们还必须关注一个重要的函数——效用（Utility）函数[16]，它是一个关于消费者资源请求的函数。在微观经济学的消费者行为理论中，消费者在可供选择的商品中进行选择，并按照从消费的商品中得到尽可能大的满足的方式来进行，这意味着他们知道所面临的各种选择，并且对这些选择做出评价。而消费者从各种不同数量的商品中取得的满足的有关的全部信息都包含在他的效用函数 U 中。效用函数的值域是效用值，它表现了消费者对当前消费的满足，这个值越大，表示用户对当前的消费越满意。效用函数具有连续的一阶和二阶偏导数，并且是一个严格的正则拟凹函数[17]（regular strictly quasi-concave function），它的偏导数是严格的正数，这表示不管那种商品，消费者总是希望得到更多的；二阶偏导数的存在表示用户在有限资金的前提下对当前资源的消费总是会达到一个最满足的程度。消费者的效用函数并不是唯一的，并且是针对某一特定时期的消费。

3.2.3.2 网络资源中采用竞价模型

自上世纪九十年代开始，随着计算机网络飞速的发展，早期的网络资源控制机制并不能满足如音频、视频数据流的传输请求。通常的网络控制协议通过重传的方式处理传输过程中丢失的数据，而对于音频、视频传输中数据包的丢失或者对数据出错重传非常敏感，因此经济学家 H. R. Varian 和 J. K. MacKie-Mason 提出采用竞价的方式来避免网络的拥塞[18]，P. Key 和 D. McAuley 则从计算机网络角度讨论了竞价方式在网络中的实现。这种方式通过监控器实时的监控网络带宽的使用情况，如果网络发生了拥塞，则为带宽资源定义一个合理的影子价格，带宽请求者再根据自己的实际情况适当的选择服务的等级，如按照低音质或者低视频质量的效果发送数据包或者只发送音频数据而忽略视频数据的传送，这样势必降低了音频/视频的质量，但却尽可能的保证了接收方整个音频/视频文件的连贯性和逻辑正确性。竞价的方式利用了经济学中个体与市场共同决定资源价格以无形的控制、调节市场资源配置的特性，避免了系统单方面的控制资源的配给，因为是请求者而非系统更清楚应用的需求，从而实现系统和用户共同调节资源的分配。

3.2.3.3 系统和用户效用值的最大化

上文论述了通过拥塞竞价的方式实现网络带宽的分配，因为资源的分配不但受用户自身的请求数而且受其它请求资源的用户的限制，因此分配的方式会有多种，而如何分配并以怎样的规则实现使得每一个用户的利益最大化，受到多种条

件的制约。从微观经济学的角度来看，则是：希望当前的分配使得获得资源的个体与整个社会的效用值均达到最大。

微观经济学中消费者对物品选择的满意程度量化为用户的效用值，如果想最大化整个微观团体的效用值，则需要最大化每一个消费者的效用值。通常一个请求资源的用户 i 的效用可以表示为公式 3.4：

$$U_i(x_i) = u_i(x_i) - C_i \qquad \text{式 3.4}$$

u_i 是用户 i 的效用函数方程，它与用户的资源请求相关，C_i 是与当时环境相关的非常量的资源消耗。希望最大化系统的效用，则可以看成是最大化的所有用户效用的总和与整个系统的额外花费之差，如公式 3.5 所示：

$$\max(\sum_{i=1}^{n} u_i(x_i) - \sum_{j} C_j(\sum_{i=1}^{n} x_i)) \qquad \text{式 3.5}$$

这个优化的过程可以通过数学的方法来实现，但是依赖于用户的具体的效用函数 $u_i(x_i)$，然而用户的效用函数针对某一特定时期并不是唯一的，并且系统也不知道该函数。[20]的推倒则简化了这一问题，指出该问题可以被分解为优化每一个用户的效用为最大的方式，从而使得整个系统效用达到最大，同时这个优化并不涉及到单个用户的具体效用函数。

假设用户使用的资源的约束参数 t_i 与它的资源请求量 x_i 存在比例关系，即 t_i 和 x_i 决定 C_i，则用户效用最大化问题转为公式 3.6

$$\{\max\ U_i(x_i)\} = u_i(x_i) - t_i x_i \qquad \text{式 3.6}$$

通常根据效用函数的特性，即该函数总是一个单调增加的、凹/凸并且具有连续倒数的函数，因此，$u_i'(x_i)$ 存在，并且当 $u_i'(x_i) = t_i$ 时，用户的效用达到最大。从公式 3.6 可以看出 t_i 限制了用户对物品的获取欲望，经济学中被看作是控制市场物品分配的影子价格。通常 $t_i = p(y)$，表示 t_i 是一个关于负载 y 的函数，y 则为系统中当前资源的负载值。

通过上文的分析，我们则可以将最大化整个系统效用值的过程简化为求得系统当前的影子价格。影子价格与当前请求资源的任务相关，它需要系统实时的反馈来获得，而用户则根据自己的实际需求和影子价格来调整自己所能请求的资源，以实现系统资源分配的优化。

3.2.3.4 Octopus OS 的中使用竞价调度

当发生资源紧缺的情况时，竞价的方式可以像处理网络拥塞一样来处理

Octopus OS 中任务的调度，原因如下：

1. 从概念上看传统操作系统中，资源管理与分配都是以系统为中心的，应用本身则被动的被调度，而 Octopus OS 中任务是自治方式实现的，任务可以根据现场的情况实时的改变自己对资源的请求，因为任务本身而不是系统更了解其所需要的资源，而竞价的调度则为这种需求提供了一个平台。
2. 模型中影子价格是一个与系统中所有任务相关的约束条件，通常如果任务 n miss deadline，是由于前 n-1 个任务占用了资源而导致的，而影子价格则可以控制系统中这 n 个任务对资源的占有。此时系统只要控制影子价格，而任务根据该价格通过公平合理的“购买”来获得资源，即通过任务之间的“竞争”和系统的“宏观调控”实现资源的分配，同时协调任务对资源的请求、最大化系统的效用，也就实现了系统中任务的调度的优化；
3. 竞价的方式易于扩展到多种资源而不只是 CPU 资源的分配，此时系统只要求得当前该资源的影子价格即可。

可以看出，竞价的方式应用于 Octopus OS 这样的操作系统的调度也是可行的，我们借助反馈获得资源的使用情况的同时也算得当前系统中该资源的影子价格，从而实现系统中任务资源的分配。

3.3 本章小结

本章在对 Octopus OS 结构和传统的基本的调度策略分析的基础之上，分析了 Octopus OS 调度的特殊需求，并提出了相应的解决方案。

Octopus OS 目标是应用于可靠性要求非常高的如军事、航空等领域的系统上，对任务的自主执行要求的非常高。通常系统的执行的环境是不可预测的，但又要求系统本身执行的正确性和可靠性。因此单一、固定的调度策略显然不能满足。本文则考虑采用闭环的调度方式，实时的反馈当前系统对资源的使用情况，从而选择一个合适的调度方式。

对于在需要资源分配优化的情况下提出采用竞价的方式实现。这种设计理念源自于微观经济学，其将资源分配的优化转换为求得当前系统的“影子价格”上，当市场中对某资源的请求量增加时，该资源的价格自主的被提高。任务根据自身的实际情况以及对资源价格的判断调节对资源的获取，系统本身则起到宏观调控的作用。这种方式已经被应用于网络带宽的分配，以达到每个任务的 QoS 保证，但是还没有被应用于操作系统资源的分配中，在下一章我们将讨论这一实现。

第四章 Octopus OS 调度算法的实现与分析

4.1 调度算法的构架

对于 Octopus OS 的调度采用基于反馈的调度模式实现，系统根据反馈的结果实时的选择相应的调度策略，该结构如图 4.1 所示。

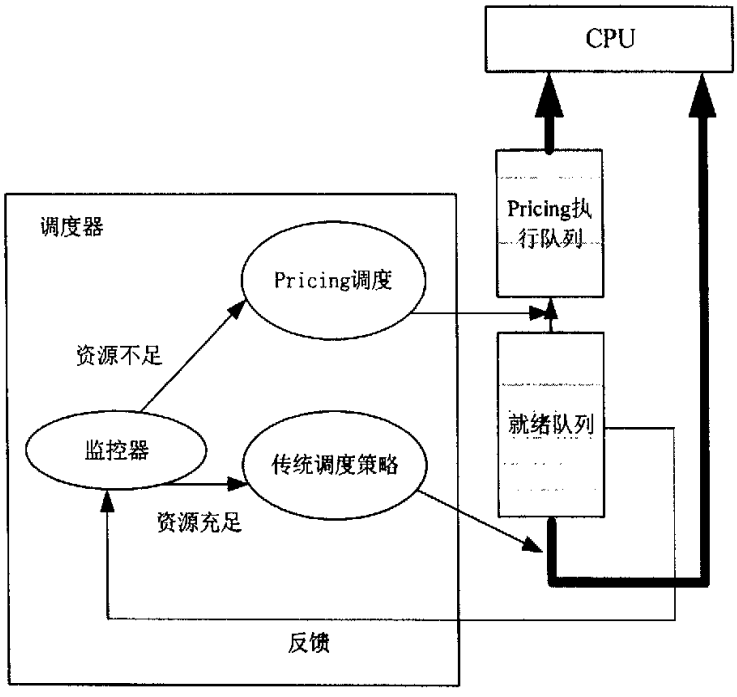


图 4.1 Octopus OS 系统调度构架

Octopus OS 的调度使用反馈的方式是因为系统运行的环境往往不可预测，通过反馈实时观测系统的执行和资源使用状况以提高系统调度的可靠性，同时提高资源的利用率。当资源充足时，采用某种通用资源分配策略，而当资源紧缺时，采用竞价的方式实现系统中自主执行的任务的资源分配的优化。图 4.1 主要考虑 CPU 资源，它是任务执行的最关键资源，但除了 CPU 资源外还有如内存页等其它资源的也必不可少，而这些资源在任务执行的过程中也同样被竞争，因此这个模型也可应用于这些资源的分配上。本章将以 CPU 资源为例讨论该模型的实现，在章节的最后讨论了这种方式在其它资源上的扩展。

4.2 调度算法的实现

4.2.1 反馈控制的实现

4.2.1.1 反馈构架分析

反馈控制是 Octopus OS 调度算法的第一步。反馈结果的判断通过监控器完成，判断的准则是当前的资源是否充足。当资源不足时采用竞价的调度方式；资源充足时，选择传统的调度策略。根据不同的资源竞争情况选择不同的调度策略的优点是：

1. 虽然当资源充足时，也可以采用竞价调度的方式选择待执行的任务，但在系统执行过程中因为软件执行的需要和硬件的支持，资源在绝大多数情况下都是充足的，竞价的方式为了实现资源分配的优化，算法的复杂度必然被提高。而传统的调度策略已经非常成熟，并且简单易用，算法的复杂度很低，如果一直采用竞价的方式势必提高了调度算法的时间复杂度、增加调度的时间；
2. 再者，因为 Octopus OS 会应用到多种领域中，每个领域在系统执行过程中，对任务的执行目标并不相同，当资源充足时，对注重公平的系统采用时间片轮转，而对注重关键任务执行的系统采用优先级调度等，这样可以灵活的改进、扩充调度算法。

4.2.1.2 系统监控器

系统的监控器有两个任务，首先是判断系统资源是否充足。资源使用情况的判断是监控器的主要任务，调度程序周期的调用监控器，判断的结果将会直接影响到系统的性能。通常采用两种判断的方式：事前判断和事后判断。事后判断是在当前采样周期开始，判断上一个时间周期内任务的执行，因为任务的执行有一定的惯性[21]，从而判断当前资源是否缺乏，如在网络中，通过判断上一周期内丢包的数量，来决定网络带宽的使用情况，从而控制发送端这一个周期发送的数据包量。而事前判断则相反，它不依赖于上一个周期的观测数据，而是通过监测当前周期任务对资源的请求，以判断这一个时间周期内资源是否缺乏。在 Octopus OS 中将采用事前判断的方式，因为这种方式具有较好的可预测性，而事前判断对于一些不可预测情况如当前执行周期内到来的任务不能较及时的处理。我们的解决方案：在当前周期开始前，监测就绪队列中的任务，查看任务需要的执行时间总和是否大于一个给定的时间段，这个时间段是相对于本周期内可用时间的子集，

这一思路来自于实时系统中资源预留的方式，如公式 4.1 所示，其中 S_i 表示第 i 个任的执行需要的时间，T 为采样周期，k 为预留系数

$$\sum_i S_i < k * T \qquad (0 < k < 1) \qquad \text{式 4.1}$$

资源预留是为提出资源要求的计算预先保留一定的计算能力或系统资源以保证其使用。预留系统资源首先要求该资源是可以在几个进程或应用程序之间复用的，通常可以用于预留模型的系统资源主要包括处理器和网络带宽等，这两种都是连续资源，除此之外，一些离散资源也可以被预留，如物理内存页和缓冲区等，不过离散资源的预留应该在该资源的最小单位的基础上进行[22]。在这里我们主要考虑 CPU 资源，[22]中的论述保证了 CPU 资源预留的可行性。

系统监控器的第二个任务是为竞价调度收集参数。当资源紧缺时，Octopus OS 采用竞价的方式实现任务的调度，竞价调度的关键是求得资源的影子价格，影子价格反应了系统中资源的稀缺程度，一个进程因为缺乏资源而导致执行的失败是与其它进程相关的，影子价格可以从宏观的角度限制其它进程对资源的占用，而监控器则可以通过周期的监测当前系统资源请求的状况求得获取影子价格的必要参数，具体实现将在下文中讨论。

4.2.2 竞价资源分配的实现

4.2.2.1 任务属性定义

竞价调度模型利用了微观经济学中相关特性，通过为竞争的资源设定价格，并通过任务之间的相互竞争和购买实现资源分配的优化，首先将任务的属性与微观经济学模型相对应：

1. 任务的收入：系统中每一个任务都具有一定的重要性，用任务的收入来体现任务重要性，重要性越高，收入也就越高。通常这是任务与系统达成的一个协议，定义任务的收入为 M= $F(x_1, x_2 \ldots x_i)$，F 是一个受多个参数控制的函数，如任务优先级、执行时间、等待时间等。
2. 价格：根据当前系统资源紧缺的实际情况，计算影子价格。由于系统中待执行的任务在每一个阶段是不同的，而不同重要性的任务其收入又存在差异，因此当前系统中“流通”的钱数必然不同，导致价格会随之变化，以体现当前资源的稀缺程度，通常资源越匮乏，价格应该越高。如可以考虑计算单位资源所要付出的钱数来定义价格，即求出当前任务收入总和 ΣM_i，假设目前可用资源

则价格定义为 $p = M_i / S$；

3. 任务的执行等级：当消费者购买商品时，会根据手中的钱数和商品的价格确定对商品的购买量，如果价格偏高，消费者会降低对物品的偏好减少对物品的购买。这里模拟消费者购买商品的行为，每个任务根据分配给自己收入和当前的价格决定要购买的资源量。因为任务执行的特殊性，所请求的资源量是离散的，我们抽象任务的资源请求量为消费者的偏好，用 x_{ij} 表示，即当第 i 个任务在获得的资源量为 x_{ij} 时可以完成第 j 个执行等级，等级越高，请求的资源越多。

4.2.2.2 资源分配模型

资源不足往往因为系统任务数的增多或者因为所能获得资源的减少而造成的。图 4.2 从资源分配角度描述竞价和非竞价方式下用户获得资源的方式。当资源充足时，任务可以按照最大资源请求量"无偿的"获得资源，即不需要"竞争"，而采用通用的调度策略实现，如图 4.2（a）所示。而当资源紧缺时，系统为每一个任务分配一定的"收入"，并为竞争的资源指定"影子价格"，就绪队列中的任务根据手中拥有的"钱数"来决定自己对资源请求的数量，如图 4.2（b），被选中的任务放到图 4.1 所示的 Pricing 执行队列中。

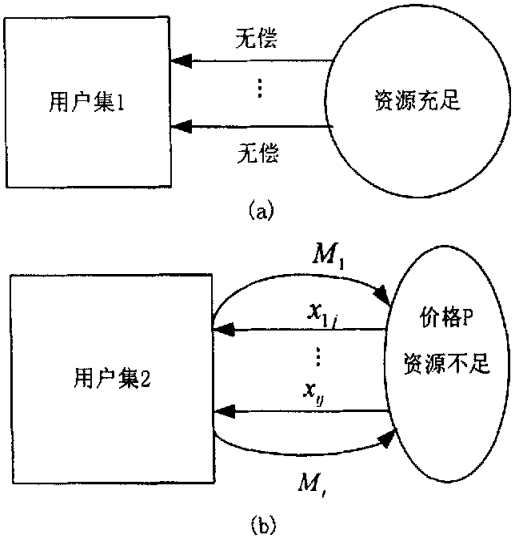


图 4.2 资源分配模型

4.2.2.3 算法的实现

任务根据系统分配给其的钱数选择要执行的等级。具体实现如算法 4.1 所示

```
TaskPricing( )
{
1:      While(任务队列未结束)
          {
2:        While(当前任务 i 的第 j-1 个服务等级仍未被满足)
            {
              deltaMi=Mi-x_ij*P;
              if(deltaMi＝0) goto 1;
              if(deltaMi>0) deltaMi=deltaMi+deltaMi-1; goto1;
              if(deltaMi<0) goto 2;
            }
          }
         While(deltaMn>0)   reallocate();
}
```

算法 4.1 任务竞价算法

首先通过公式 4.2 计算就绪队列中任务 i 所拥有的收入是否能够满足任务此时的资源请求量 x_{ij}，

$$M_i - x_{ij}P = \Delta M_i \qquad \text{式 4.2}$$

其中 j=0，1，2...表示任务的多种执行等级，0 为最高等级，即资源请求最多，随着 j 的增加，任务等级逐渐降解。ΔM_i 有三种情况：

1. 当 $\Delta M_i = 0$，表示当前的钱数恰好可以满足任务的资源请求量 x_{ij}；
2. 如果 $\Delta M_i > 0$，则表示当前任务的收入在满足资源请求后还有一定的盈余；
3. 当 $\Delta M_i < 0$，表示当前任务所拥有的收入无法满足请求量 x_{ij}，利用公式 1 查看是否满足 $x_{i(j+1)}$。对每一个任务反复使用公式 1 直到找到一个合适的执行等级 j，如果最差的等级都不能满足，则表示该任务在下一个时间段不能获得资源。

由于任务的收入是按其重要性分配的，而与任务的执行时间无关，不可避免的出现 $\Delta M_i > 0$ 的情况，并且没有获得执行机会的任务手中的钱也没有被利用。如果 P 是单位资源的价格，显然此时系统资源是没有被充分分配的。收集剩余的钱数，表示为 $\Sigma \Delta M_i$，利用 $\Sigma \Delta M_i$ 对没有完全分配的资源进行一次再分配。遍历就绪队列，如果队列中任务已经达到了自己最高执行等级，则忽略该任务；如果任务的执行等级不是最高的，则提高其执行等级，直到 $\Sigma \Delta M_i$ 接近 ε（一个可接收的值，根据当前系统的硬件执行情况）。

4.3 调度分析

这一节将利用 Matlab 仿真并分析 Octopus OS 的调度算法。其中主要从竞价分配的角度进行了分析，为了对比竞价调度在资源分配优化上的优势，将竞价调度与传统的 EDF 调度方式进行了比较；再者，竞价调度是在资源不足时采用的优化策略，而系统资源充足与否是相对的，这里针对这一点也进行了比较；在竞价调度过程中，钱数的分配是一个关键的参数，其会影响到系统调度的效率，此处仿真两种不同的任务收入方案，讨论了系统运行队列的平均优先级。

4.3.1 仿真参数的设置

仿真过程考虑如下参数的设置：

1）任务属性：首先要指定当前系统中任务的个数，并且针对 Octopus OS 中任务的执行特点，还需要设定每个任务的优先级、任务的服务等级以及该等级对应的资源。这些值均随机产生，此处将系统中执行的任务数控制在 5～15 之间；任务的优先级在 1～10 之间；任务的服务等级数为 2～4 个级别；

2）钱数的分配：对于竞价调度，需要提出一种任务当前收入的分配策略。通常这个值受限于任务的重要性，资源请求数量等，这里将采用任务优先级作为任务钱数分配的约束条件，即 M＝f(P)。实验中，此处是竞价调度一个潜在的扩充点，可以考虑用多个参数约束 M，即 M＝（x_1，x_2，…，x_n），以实时调整任务的执行；

3）效用值：实验中，采用效用值评测竞价调度的结果。通过公式 $\sum et_i * P_i$ 描述调度的效果，这里称其为系统的效用（Utility）。其中 et_i 为所选任务 i 的执行时间，P_i 为任务执行的权值，权值越大，表示该任务执行的重要性也越大。当资源一定时，系统的效用值越大，表明该任务以 et_i 方式执行时，系统所产生的效果越好。

4）资源：实验中有一个重要参数“currentSource”，它表示当前可以获得的资源数。此处，资源不足的情况通常设计为任务最大资源请求总数大于 4 倍的 currentSource，以表示资源的紧缺。

4.3.2 EDF 调度方式与竞价方式的比较

图 4.1 所示，系统根据实时监控的结果选择合适的调度策略。资源不足时使用竞价调度，而资源充足时，采用传统的调度策略，此处选用 EDF 调度方式，用其与竞价调度进行对比，分析评估竞价调度。因为 Octopus OS 通常应用于对可靠性

要求较高的实时系统中，而 EDF 方式是当前流行的实时调度，并且 EDF 方式也兼顾了基于优先级调度的思想。虽然对于 Octopus OS 这样的任务调度需求，EDF 可选择按照每个任务的最高服务等级或者最低服务等级执行就绪队列中的任务，对于前者，如果较高优先级的任务占用了较长的 CPU 时间，那么较低优先级的任务可能会无法被执行，甚至有些任务会被饿死，这种现象在资源不足时，显得更突出；后者可以使尽可能多的任务执行，但是可能会浪费大量的资源，导致系统的效率降低。因此在资源不足时，采用竞价的方式。

我们按照上一节的提出的参数产生 100 组任务在资源不足的情况下分别用 EDF 和竞价的方式来进行处理，计算执行结果的效用值得到图 4.3：

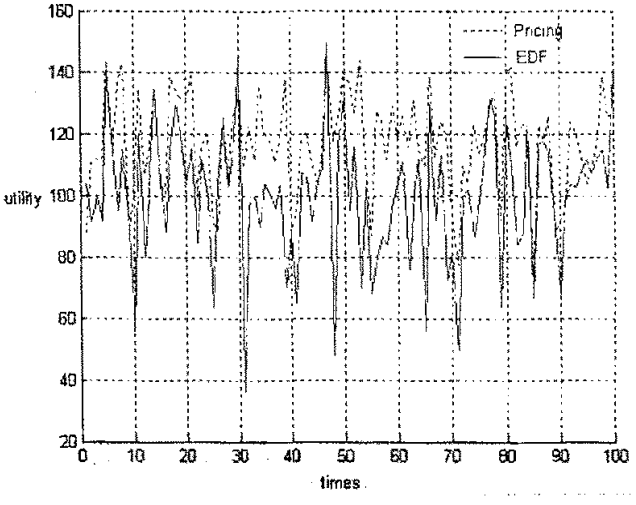


图 4.3 Pricing 与 EDF 方式对比图

此处 W[10]= {2,2,3,3,4,4,5,5,6,6}，P_i=W[i]，W 主要体现系统的公平性，即低优先级和高优先级的任务一样都有获得资源的权利。通过图 4.3 可以看出在资源紧缺的情况下竞价方式的整体的 Utility 的变化幅度要比 EDF 的方式小，并且 Utility 在绝大多数情况下大于 EDF 方式。

4.3.3 比较资源不足和充足的情况

延续上一个实验，竞价的方式虽然有效，但效率较低，何时采用何种方式调度，将影响到系统的整体性能，该问题将转换为如何定义资源的相对充足，这个工作由监控器实现，可以通过多次的采样，针对不同的系统需求，以判断所要选择的调度方式，这个实验则针对这个问题。图 4.4 是采用竞价的方式在三种不同资源情况下 100 组任务的效用曲线图，随着资源的增加，系统的效用值会随之增加：

图 4.5 是 EDF 与竞价方式的在资源充足和不足情况下的比较，可以看出竞价的方式在资源不足的情况下要好于 EDF 的方式，而资源充足的情况下，竞价方式与 EDF 的方式的执行结果是基本相同的，也就是说在资源充足的情况下，我们使用 EDF 就可以满足要求，并且 EDF 方式的效率要高于竞价的方式。而资源不足的情况下，使用竞价的方式可以得到比较好的结果。

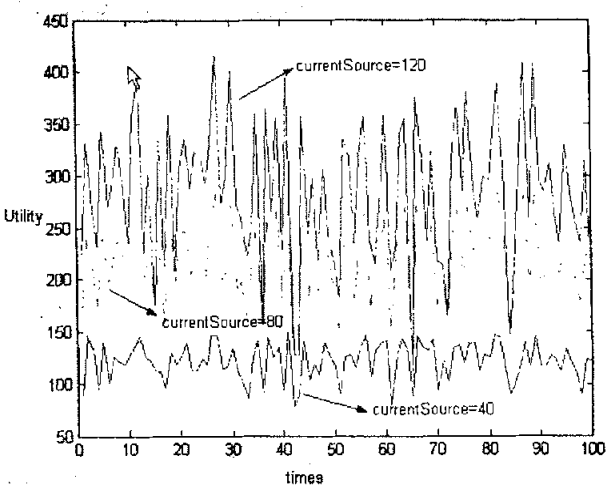


图 4.4 pricing 方式资源变换调度对比图

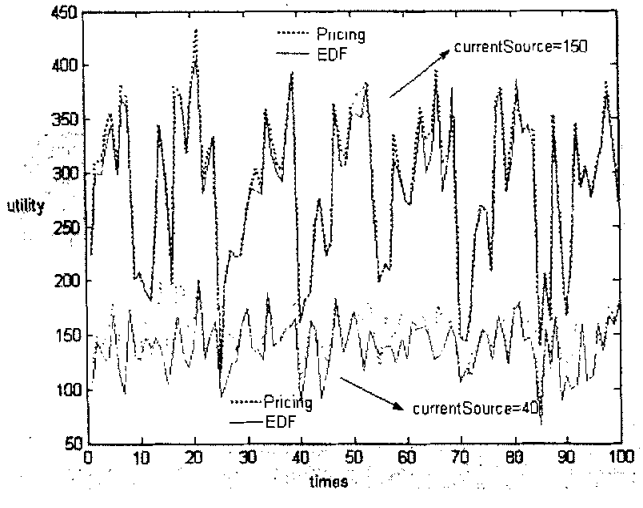


图 4.5 pricing 与 EDF 方式资源变换调度对比图

4.3.4 改变分配钱数的策略

在竞价调度中，一个比较重要的参数是系统中每一个任务的收入，分配给任务的收入决定了该任务被执行的可能性的大小及其服务等级。可以通过改变任务收入的控制函数，来决定系统对任务调度的侧重点，以优先级为例，当不同优先级任务所分配的收入差别不大时，则低优先级获得资源的概率增大；反之，高优先级将会获得更多的资源。此处通过改变 W 的值来改变分配给不同优先级任务钱数的分配策略。图 4.6 中（a）的 W 与上面的实验相同，而（b）中 W[i]= 2^i，纵轴表示当前这组任务的平均优先级。

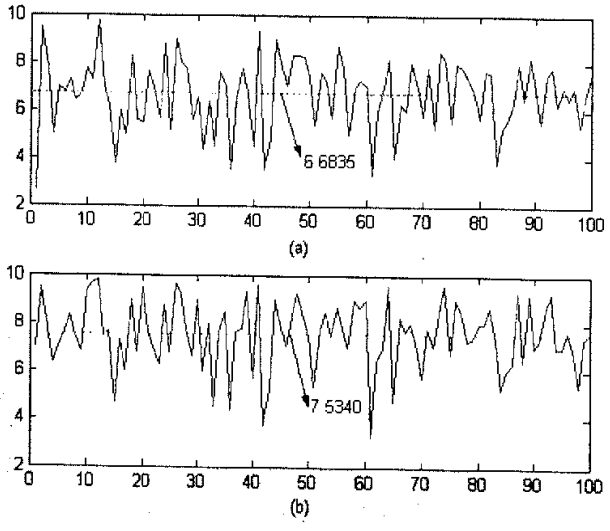


图 4.6 pricing 方式下不同钱数分配策略对比图

可以看出（a）中 100 组任务的平均优先级小于（b）中的，对于（a）更多小优先级的任务得到的执行的机会，而（b）中由于给高优先级的任务分配了更多的资金，其更有可能获得资源。

4.4 本章小结

本章对于前一章提出的 Octopus OS 调度方案给出了具体的解决策略，并通过 Matlab 仿真对该调度方案进行了分析。文中首先给出了整个调度策略的整体框架，并针对该框架进行了具体的分析与实现。监控器周期的监测系统资源的使用情况并为计算影子价格收集参数，当资源充足时，采用传统的调度策略如基于优先级、

时间片轮转以及实时调度 EDF 方式等；在资源有限的情况下，根据当前系统可获得资源，采用经济学的竞价模型进行调度。

在仿真实验中借助 EDF 调度，从三个方面对竞价方式进行了对比、分析。第一个实验可以看出在资源充足和不足的情况下分别采用 EDF 和竞价的调度的方式，可以总能使系统的效用达到一个较大的值。第二个实验从监控器的角度出发，通过判断不同的系统相对可提供资源的情况，决定需要采用的调度策略。第三个实验则从任务执行的角度分析了竞价调度中不同重要性任务收入对整个系统任务执行的影响。

对于 Octopus OS 调度算法待扩展的部分：目前竞价的方式只是考虑到在 CPU 资源上的应用，今后还可考虑应用于内存的管理，而缺页将作为监控器的监控信息；通常管理多资源预留是非常困难的，而竞价这种系统宏观调控、个体公平竞争的资源管理方式使得其可以容易扩展到特定信息和知识的应用领域。

第五章 在 Bochs 虚拟机上实现 Octopus OS

因为 Octopus OS 采用了基于状态的体系结构，在这种结构中系统执行的重要状态放在共享内存中，而传统的操作系统内核由于实现了多种功能，任务的状态繁多，如果采用基于状态的方式管理会非常复杂且难于实现。为了验证 Octopus OS 的思路，项目组考虑在一个实际环境中重新构建一个操作系统框架。这一章将介绍在 Bochs 虚拟机上构建仿真、调试的环境并围绕着系统调度相关的内容进行了讨论，如硬件平台对进程控制的支持与限制以及相关数据结构的设计与组织等。

5.1 仿真环境的创建

5.1.1 虚拟机的选择

考虑 Octopus OS 的模拟选择在虚拟机上而非真实的机器上运行，在降低成本的同时也为调试带来了方便，如通过改变虚拟机执行参数就可以改变 CPU 主频、内存、硬盘等硬件资源，从而实现多种硬件环境的模拟。Octopus OS 目前是一个探索的阶段，需要搭建一个环境来实践该想法，因此选择何种 CPU 结构运行不是主要的。这里选择 x86 体系结构，因为当前的 PC 机多都采用该结构使其应用的非常广泛，因此相关的硬件手册、上层实现的资料也非常多，便于查找和借鉴。

目前流行的 PC 仿真软件主要有三种：VMware 公司的 VMware Workstation、Connectix 公司的 Virtual PC（该软件被微软收购）和开放源代码 Bochs。这三种软件都可以虚拟或仿真 x86 硬件环境，使得在运行这些软件的系统平台上运行多种其它的操作系统。但从使用的范围和运行的性能来说，有一定的区别：Bochs 仿真了 x86 的硬件环境及其外围设备，可以很容易被移植到很多操作系统或者系统结构平台上如 x86，PPC，Alpha，Sun 和 MIPS，由于主要使用了仿真技术，其运行的性能和速度都要比其它两个软件要慢很多；Virtual PC 仿真了 x86 的大部分，而其它部分则采用虚拟技术实现，它的性能界于 Bochs 和 VMware Workstation 之间；VMware Workstation 则仅仅仿真了一些 I/O 功能，而所有其它部分在 x86 实际的硬件上直接执行，当用户在虚拟机上执行一条指令时 VMware 不是用仿真的方法来模拟，而是把该指令传递给实际系统硬件来完成，很显然这种方式的速度和性能是最高的，但可移植性很差。

从应用方的角度来看，如果仿真环境主要用于应用程序的开发，那么 VMware Workstation 和 Virtual PC 是比较好的选择，但是如果需要开发底层系统软件、进行操作系统的开发和调试、编译系统开发等，则 Bochs 是比较合适的，因为使用 Bochs 操作系统的开发和调试、编译系统开发等，则 Bochs 是比较合适的，因为使用 Bochs

可以知道被执行程序在仿真硬件环境中的具体状态、精确时序等相关信息。因此Octopus OS 也选择在 Bochs 虚拟机上模拟。

5.1.2 系统编写中 C 语言和汇编语言的使用

Octopus OS 的编写采用 C 语言和汇编语言结合的方式。在 C 语言中提供了某些汇编级语言操作如按位操作，但为了 C 语言规范的简练和通用，C 标准对许多与具体平台、硬件相关的操作如 CPU 中断、寄存器访问、I/O 指令没有提供相应的约束标准，而通过操作系统库函数和编译器支持实现的功能也只是一部分，并且在编写操作系统的过程中，空间和时间的效率往往非常重要，因此用 C 语言编写操作系统的过程中，需要辅助以汇编语言。通常可以采用两种方式实现 C 语言与汇编的结合：1）C 语言调用汇编代码，因为链接器的存在使得不同的程序段分离编译最后生成可执行文件成为可能，只要遵循一定的规则就可以实现 C 语言和汇编语言的相互调用；而且现在很多汇编工具也吸收了 C 语言预处理的长处，并且数据结构也可以放在头文件中加以定义，方便了两种语言的结合；2）内嵌汇编，这种方式是在 C 语言中直接写入汇编语言，是 C 编译器做的相应的扩展，如 GCC 中提供“asm”[24]语句功能。

在编写操作系统代码中，使用汇编语言通常出于以下几个方面的考虑：

1. 与硬件打交道的程序段通常需要用到一些专用的指令，最常见的是与 CPU 相关的，如开/关中断、寄存器（如设置控制寄存器、某些功能寄存器等）操作等，此外，在同一种系统结构的不同 CPU 芯片中特别是新增的功能，往往会增加一些新的指令，执行这些指令需要用汇编语言编程；
2. 操作系统中实现某些操作的程序段或者函数，在运行时会非常频繁的被调用，因此对该程序段的运行速度要求很高。而用汇编语言编写的程序，在算法和数据结构相同的条件下，其效率通常要远高于用高级语言编写的。如系统调用的进入和返回非常频繁，并且该过程还涉及到用户空间和系统空间之间的切换，而用于这个目的的一些指令需要用汇编语言来编写，以提高效率。
3. 在编写操作系统时，通常需要设置某个寄存器或某个数据域中的指定项，出于效率的考虑，使用汇编语言可以很容易的定位并设置该数据项；
4. 某些特殊的场合，一段程序所占用的空间大小也会显得非常重要，如操作系统的引导程序，主引导程序通常要容纳在磁盘上的第一个扇区中，这个时候必须控制在 512 字节之内，用汇编语言可以很容易的实现。

5.1.3 系统的编译与调试

Octopus OS 选择 GCC 作为编译器，实现系统的编译、连接的工作。GCC 是前端支持多种语言、后端支持多种平台的编译系统，在编译的前端实现词法和语法的分析，词法分析的结果传递给编译系统的后端，而语法分析的结果被翻译成 RTL（寄存器传送语言，这种语言是对具体体系平台做出的抽象，以达到支持不同硬件平台的目的），后端对 RTL 进行优化并翻译成目标机器上的汇编语言，最后触发汇编器和连接器生成目标文件[24]。我们选择 GCC 的主要目的为：GCC 为 C 语言提供了多种扩展，以方便系统内核的编写；并且 GCC 编译系统实现从前端到后端的完全支持，因此可以利用 GCC 对 C 的扩展以及后端工具的支持控制内核代码数据的分布。

Bochs 为用户提供了两种方式调试：一种是 Bochs 自带的调试工具；一种是远程调试，使用 GNU 的 GDB[25]。对于前者 Bochs 系统启动后立即出现调试指示符，调试者可以从 FFFF0h 处开始调试（这是 X86 默认的系统加电后开始执行的地址），或者通过命令行指定要调试的内存地址，此时调试者需要对内核文件的程序段非常清楚，以确定调试的位置；后者在 Bochs 启动后，等待另一个终端启动 GDB 实现调试，调试的工作在 GDB 中完成，这种调试方式直接加载内核镜像文件，跨过系统引导部分。调试者可以选择任何一种方式，通常后者比较直观，也相对简单，但前者可以观察到一个 PC 加电后初始化的过程。

5.1.4 Mutilboot 引导

系统硬件初始化后由 BIOS 将磁盘上主引导记录的 512 字节写到内存的 0x7c00：0000 处，这段代码再将磁盘中内核主体部分加载到内存中相应的位置。这样的加载过程是 x86 体系结构所默认的方式，基于该体系结构的操作系统都需要编写代码完成这项工作，为了使操作系统的编写者只考虑系统本身的实现而不必顾及加载的过程，Erich Boleyn 等人考虑提出一种装入程序标准，该程序完成上述的加载。Multiboot Specification 就是为在 x86 机上运行的 32 位操作系统所制定的装入规范，只要操作系统的编写者遵循这个规范，就可以使用遵循 Multiboot 规范编写的装入程序完成核心的装入工作[26]。Grub 就是遵循 Multiboot 规范编写出来的启动管理器，Octopus OS 选用 Grub 来引导，通过为在 Bochs 上运行的操作系统镜像文件中装入 Grub 来使用，在加载时通过命令指定所加载的 Octopus OS 内核文件。

5.2 进程控制块管理

5.2.1 进程控制块

操作系统中，每一个进程都有一个进程控制块，称为 PCB，它用于存放与该进程相关的信息，以实现进程的管理和控制。最基本的信息应包括如唯一的进程号、进程状态、用于进程切换保存 CPU 相关寄存器的信息（如段寄存器、指令寄存器等）。在 Octopus OS 中系统的 PCB 如表 5.1 所示：

名称	占用字节数	注释
taskNo	1	任务号
taskState	1	任务的状态，表示任务是存活还是死亡
money	4	该任务所拥有的钱数
serverarray[LEVEL]	LEVEL	任务对应的服务偏好
priority	1	任务对应的优先级
register	**	重要寄存器信息
……		其它信息

表 5.1 PCB 控制块结构

在该 PCB 中 register 用于存放重要的寄存器信息的结构，如表 5.2 所示：

名称	占用字节数	注释
esp0	4	堆栈指针寄存器
ess	4	堆栈段寄存器
eip	4	指令寄存器
……		

表 5.2 寄存器存储结构

5.2.2 PCB 与进程堆栈空间

每一个进程除了最基本的 PCB 外，还需要一个堆栈空间，堆栈中存放的信息如：进程执行过程中以及被调用函数的局部变量、被调用函数的地址信息、进程切换的上下文信息等。可以说进程堆栈空间和进程控制块都是进程的重要组成部分，因此在 Octopus OS 中考虑将系统堆栈空间和进程控制块统一存放，数据的管理使用 C 语言中联合的方式，如图 5.1 所示

```
union task_union {
    struct {
        struct task_struct task;   //任务控制块
        union task_union *PCBNext;  //指向下一个空闲块
      } PCBControl;
    unsigned long stack[4096/sizeof(long)];   //堆栈空间
};
```

图 5.1 进程 PCB 与堆栈空间结构

当操作系统为每一个进程分配一个 PCB 时，实际上按照类型“union task_union”分配，每一个进程占用 4K 字节即一个物理页面[1]，这样使得 CPU 加载 PCB 时被放入到一个完整的物理页上，该页底部为 PCB 块和一个指向下一个进程控制块的指针（该指针用于实现 PCB 组织和管理，具体细节将在下面章节描述），顶部为进程的堆栈空间，如图 5.2 所示

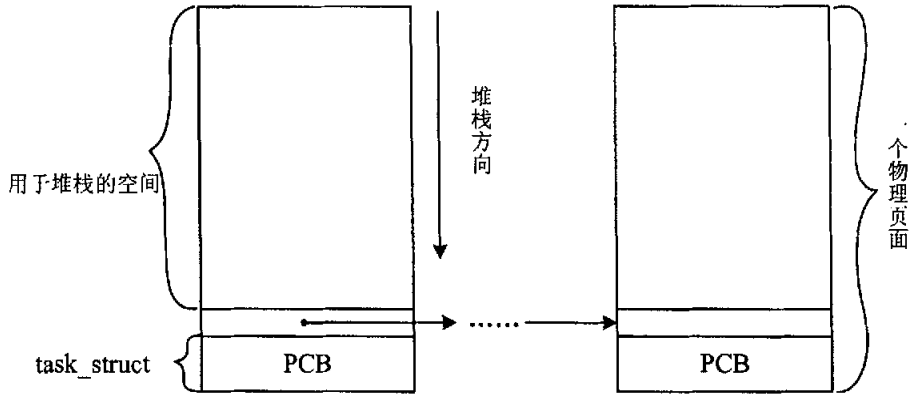


图 5.2 Octopus OS 中进程控制块与进程堆栈空间示意图

可以算出堆栈空间的可用大小为（4K-sizeof(task_struct)- sizeof(union task_union*)），因此对于该进程调用的函数嵌套不能太深，否则将会覆盖进程的 PCB。虽然如此，这种方式带来的好处还是显而易见的，当设置堆栈指针寄存器 ESP 时，可以用 PCB 起始位置加 4K；而且当系统进程在运行的时候通常需要访问当前进程自身的 PCB 如改变当前任务的优先级、分配的钱数等等，这种存取方式可以很容易通过当前的 ESP 获得 PCB 的起始位置，Octopus OS 中获得的方式使用

[1] Intel 在 i386 之后为了管理的灵活引入了页式内存管理方式，这种方式使得物理上离散的空间在逻辑上看似是连续的，而页的大小会影响到内存管理的性能，通常一页为 4K[27]。

与操作，如 $esp \cap 0xfffff000$

因为当为每一个进程分配控制块时，均为 1 个物理页面，即 4KB，因此每一个进程控制块的起始位置均为 4KB 对齐的，而每一个进程的 ESP 又在其 4KB 范围内，采用“与”的方式就可以准确而快速的定位进程控制块起始位置。具体实现用嵌入式汇编方式，如下：

```
asm("andl %%esp ,0;" ":=a"(current): "0"(~4095)); //current 为 task_struct*类型
```

5.2.3 进程控制块的存放

在 Octopus OS 中，进程控制块作为全局变量存放在数据段中，为了更加直观利用 GCC 对 C 的扩展在数据段中加入一个新的节“task”[24]如下所示

```
union task_union \
  __attribute__((__section__(".data.task"))) taskInfo[TASK_NUM];
```

Octopus OS 中通过设置 TASK_NUM 限制系统中可控制的任务数，其中__attribute__是 GCC 对 C 中函数类型、变量类型、数据类型属性扩展的关键字，__section__是变量类型扩展的一种，它在默认的数据段 data 中加入了 task 节，使用这个属性的同时还要在连接脚本中加入相应的语句：

```
.data.task: AT(LOADADDR(.rodata)+ADDR(.data.task)-ADDR(.rodata))
{
    *(.data.init_task)
}
```

这里利用前一个节 rodata 以确定.data.task 在物理内存中的位置。

5.2.4 进程控制块的组织

在 Octopus OS 中通过 TASK_NUM 限制系统所能管理的进程数，如前文所述 PCB 块位于数据段中，因为 GCC 编译的中间结果采用 ELF 格式保存[31]，这种格式中数据段空间用于存放全局 C 变量，它是编译后预留的，不能像堆空间一样动态的申请和释放。由于每个进程均需要一个进程控制块，因此系统为了处理更多的任务，在每一个任务执行结束后需要逻辑释放其所占用的 PCB，以供新进程使用。这种释放可以采用两种方式实现：1）加入标志位，用以标明该 PCB 当前的状态：使用/空闲，查找空闲块时顺序的查找 PCB 表中的下一个，直到找到空闲块为止；2）为每一个 PCB 加入一个额外的指针，该指针总是指向下一个空闲的 PCB。对于前者，每个 PCB 虽然只需要 1B 标志位，但是它的查找效率低，为 O(n)；而后者每个任务的额外空间为 32B，但是链表首指针为当前空闲的 PCB，便于查找。

Octopus OS中考虑采用第二种方式实现，如图5.2所示每个任务包含一个指向union task_union的指针，系统刚启动时为PCB在数据段分配了相应的空间，每一个PCB块连续存放，并通过指针改变各PCB块的相对位置。对于这种方式，考虑初始化PCB表、申请PCB、逻辑释放PCB这三个操作：

1．初始化PCB表

在PCB表中借助一个指向数据段中第一个空闲的PCB，初始化过程如算法5.1所示，初始化后结构如图5.3(a)所示。

```
while(i < 数据段中 PCB 块的个数)
{
  顺序的将 taskInfo 中每一个 task_union 数据项的空闲块指针指向与其
紧邻的数据项
    }
最后一个数据项指为空
PCBFreeList 指向 PCB 表中第一个空闲块
```

算法5.1 初始化任务PCB表

2．申请PCB

占用PCB即将空闲块从PCB表中分离，如算法5.2所示：

```
if ( PCBFreeList 不为空) //如果为空，表示当前没有空闲的 PCB 块
    PCBFreeList = PCBFreeList->next // 调整头指针
返回调整之前头指针指向的 PCB 块；
```

算法5.2 申请PCB块

3．逻辑释放PCB

释放即将使用完的PCB块加入到空闲列表中，为了避免查找的开销，利用头节点，将空闲PCB插入到空闲链表的表头，如算法5.3所示：

```
FreePCB->next = PCBFreeList
PCBFreeList = freePCB   // freePCB 为释放的 PCb 块
```

算法5.3 释放PCB

当经过若干次PCB的占用与释放空闲链表中指针指向已被打乱，如图5.3(b)

所示。

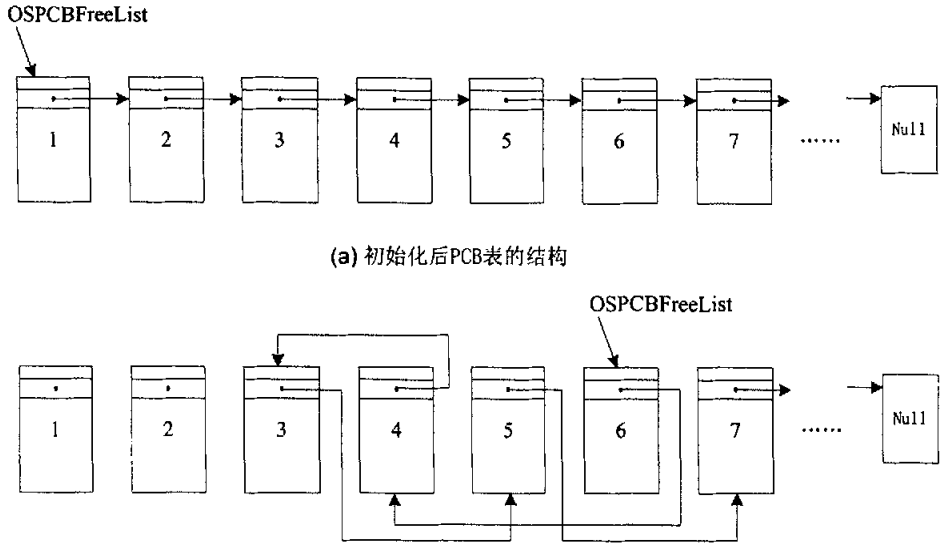


图 5.3 系统 PCB 的组织和管理

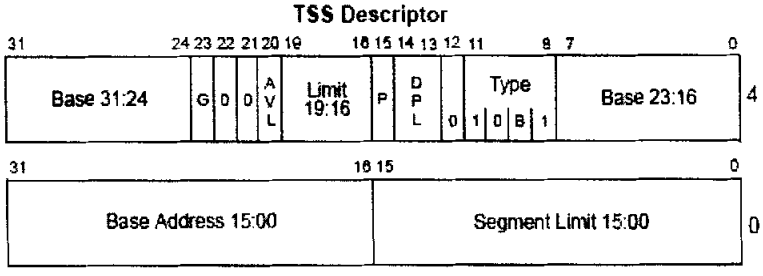
5.3 进程间任务的切换

5.3.1 X86 系统结构从硬件上对进程切换的支持

x86 体系结构的实现中，考虑到从硬件上支持任务间的切换，并为此添加了 TSS（任务状态段）和 TR（任务状态积存器），在 TSS 中存放着一个进程的关键的状态信息，TR 用来指向当前任务的 TSS（TR 与段寄存器 CS、DS 的功能是一样的，它存放 TSS 在 GDT 表中的位置）。与此同时 IDT 表中除了中断门、陷阱门和调用门之外，还定义了一种任务门，任务门中包含新的任务的 TSS 段选择码。当 CPU 因中断而穿过任务门时，CPU 会将当前任务的运行现场保存在自己的 TSS 中，并将旧的 TR 内容用新的替换（新值在任务门“TR 段选择码”域中，如图 5.4 所示），再通过 TR 找到 GDT 表中相应的 TSS，随后将新的 TSS 中内容装入 CPU 中的各个寄存器，从保护旧进程的现场到设置新进程的 CPU 寄存器都是由硬件完成的[27]。X86 提出的这种设计思路虽然很方便，但是由 CPU 自动完成的这种任务切换并不是一条指令实现的，而是通过 JMP、CALL 等基本的指令完成，其执行过程长达 300 个 CPU 时钟周期[29]。任务的切换在操作系统中是非常频繁的，因此

如 Linux 系统就避开了这种切换的方式，并不使用硬件提供的任务切换机制。

在 Octopus OS 中也将采用回避的方式，并不使用任务门，以及 TSS 保存每个进程切换时的寄存器副本，而是将其保存在各个进程自己的系统空间堆栈中，但是因为：x86 CPU 需要软件设置 TR 和 TSS，并且因为中断或者系统调用从用户空间进入系统空间时，会由于运行级别的变化而自动更换堆栈，而新的堆栈指针包括堆栈寄存器 SS 的内容和堆栈指针寄存器 ESP 的内容，则取自当前的 TSS，因此还必须在软件上考虑对 TSS 和 TR 的设置。Octopus OS 在第一个进程中（主要完成 CPU、中断、控制器等相关部分的初始化工作）在 GDT 表中设置唯一的 TSS 项，(TSS 项与其它段描述符项一样共 64 位)，并且该描述符指向唯一的 TSS 空间，TSS 描述符项如图 5.4 所示[27]



AVL Available for use by system software
B Busy flag
BASE Segment Base Address
DPL Descriptor Privilege Level
G Granularity
LIMIT Segment Limit
P Segment Present
TYPE Segment Type

图 5.4 TSS 描述符

GDT 表中每一个描述符有 64 位，提供 TSS 所在位置、描述符权限、段限等相关信息，因此将采用汇编的方式设置其中的值各值将更加的直观、方便。

5.3.2 进程切换的实现

任务切换最关键的是保存当前进程执行过程中 CPU 寄存器的关键信息，并用新进程的对应值取而代之。上文中提到，Octopus OS 中不使用硬件提供的机制实现任务的切换，也不采用 TSS 保存 CPU 的寄存器信息，而是采用软件的方式实现。进程切换的代码采用嵌入式汇编的方式编写如算法 5.4 所示：

其中 1 到 2 之间的语句实现了进程中一些基本寄存器入栈的操作。2 采用将当前堆栈指针寄存器 esp 放到进程自己 PCB 中保存的方式而不是堆栈中是因为：esp

指向当前进程堆栈的栈顶，而进程切换的同时需要堆栈的切换，采用入栈的方式

```
1   push eax寄存器
        ⋮
2   mov esp，当前进程进程PCB中对应域
3   mov 当前进程再次执行的位置1，当前进程PCB中
4   mov 要切换的进程PCB中保存的堆栈指针，esp
5   push 要切换进程的eip
6   ret
7   l:
        ⋮
8   pop eax
```

算法 5.4 系统任务的切换

势必引起混乱，因此必须单独保存 esp。3 保存被切换进程的 eip，这里 eip 选择 7 所在位置，当该进程再次被调度时，将从此处开始执行，完成寄存器的恢复操作。4 用新进程的堆栈指针内容更新 esp，可以看出当前堆栈是新进程的，而指令寄存器仍然指向旧进程，5，6 的配合则完成了向新进程的切换：ret 是 CPU 提供的由硬件实现的语句，它自动将当前栈顶内容放到指令寄存器 eip 中，而 5 为这一切换做了准备。至此，完成了从旧进程往新进程的切换，CPU 将从新的 eip 位置开始执行。这一执行过程如图 5.5 所示，数据保存和替换的过程由 1 到 6 顺序执行。

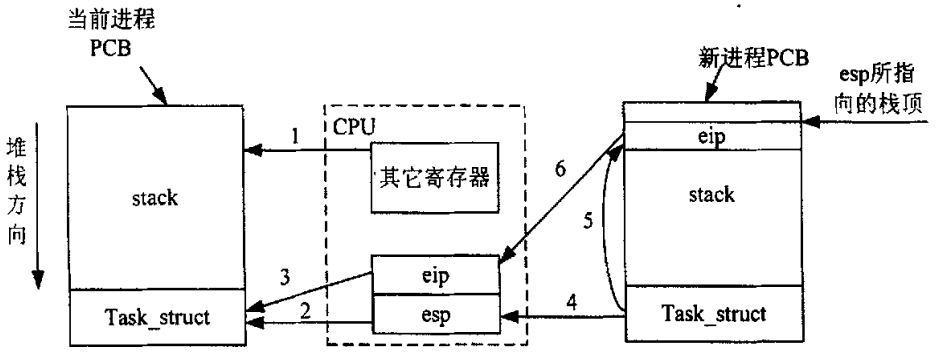


图 5.5 进程切换示意图

结束语

本文首先通过分析操作系统的发展现状，提出了传统操作系统在结构上存在的问题，从而引出了基于状态（state-based）操作系统——Octopus OS。

基于状态的系统结构模型试图采用控制理论的思想对操作系统状态进行管理，提高系统行为的可预测性和可信度。在 Octopus OS 中，状态由一组重要的系统信息组成，它们被部分开放给上层应用进程，系统中的每个进程均采用自治的方式实现：用户应用进程周期的采样自身的执行行为，并根据采样结果调整局部或者全局状态信息，必要时进行错误恢复；系统进程则周期的监控其它进程的运行时模型，以实现系统容错从而达到整体的可靠。

在 Octopus OS 中，系统调度问题是解决系统自适应能力和系统整体可靠性的关键，文章在对 Octopus OS 特性以及传统的系统调度策略分析的基础之上，提出了 Octopus OS 系统调度的需求，并针对这样的需求提出了相应的解决方案。该方案改变传统开环的系统调度策略，采用闭环反馈的方式实现系统调度的实时监控。再者，对于 Octopus OS 这样的系统，由于任务的自治执行的特性，使得每一个等待资源的任务可以根据系统资源的供给情况调整对资源的获取，文中根据这一特性借鉴微观经济学中的竞价模型，引入影子价格，通过任务之间的竞争，实现资源分配的优化。在算法分析中，采用 Matlab 仿真，通过与 EDF 调度方式比较分析了竞价资源分配的效果。

为了验证基于状态操作系统的思路，将在实际的环境中搭建一个模拟的平台，但由于 Octopus OS 结构的特性，目前流行的操作系统内核无法满足其模拟的需求，因此本文的最后利用 Bochs 虚拟机，并基于 X86 体系结构，讨论了模拟平台的搭建，并且描述了实现中关键的数据结构以及相关的重要操作。

本文为 Octopus OS 系统调度提出了相应的解决方案，仍然存在不足之处和待扩展部分如：系统调度过程中的监控器，其目前只是通过设定一个比例参数作为判断资源充足与否的控制变量，以及任务“收入”的分配也只考虑以优先级作为控制参数，在今后的工作中可以引入如机器学习、人工智能的相关领域知识，使得系统根据实际的目标系统收集相关的参数，从而提高系统控制的效率和可靠性。文中提出竞价的资源分配方式不仅仅可以应用于 CPU 资源上，也可考虑将其应用于如内存、系统能源中，以更加符合 Octopus OS 应用需求。

由于时间关系和限于作者水平，对涉及的很多问题都未能一一深入，文章虽然经过多次修改，但必然仍存在许多不完善的地方，望各位评审老师指正。

致谢

本文是我攻读硕士学位期间所做工作的总结，在此谨向所有帮助、关心、爱护过我的老师、同学、朋友和亲人致以诚挚的感谢。

首先，感谢我的导师王力教授，王老师渊博的知识使我受益匪浅。王老师无论是在学习上的指导以及生活方面给予我的关怀和帮助让我终生难忘。在此谨向王老师表示衷心的感谢。

感谢课题组负责人刘西洋老师的悉心指导以及为我提供的良好的科研环境，刘老师深厚的理论功底、广博的知识、敏锐的科学洞察力以及高度的敬业精神都极大地影响和鼓舞着我；同时感谢李航博士的耐心帮助，论文中的许多内容、观点都得益于李航博士的扎实的专业知识和无私的援助，没有他的支持和鼓励，我的论文不会得以顺利完成。在此对刘老师的支持与李航的帮助致以深深的感谢。

再者，我要感谢我的爸爸，他在计算机工程上的经验以及严谨的态度极大的影响了我，并在我遇到困难的时候，给予我鼓励与帮助。

同时感谢课题组的同学和宿舍舍友，感谢张彦春、陈甫鹃、张苗、师姐曾岸林，他们在学习与生活上给了我很多照顾与支持，特别要感谢陈甫鹃，他广博的计算机领域的知识使我受益匪浅，当我遇到问题与困难的时候总是热心的、无私的帮助我，从他那里我学到了很多知识，和这些同学在一起的日子我会铭记在心。

最后，深深感谢我的家人，他们无私的奉献和理解是我巨大的精神支柱，还有男友以及中学同学杨俊敏、惠瑜娜的关心和支持，所有这些使我能够专心于课题研究，顺利完成课题和论文工作。

谨在此向所有帮助过我的人表示衷心的感谢！

参考文献

[1] 李航. Operating System Past And Possible Future. 西安 .西安电子科技大学软件工程研究所 .内部资料. 2004.2

[2] Algridas A., Laprie J.C., Brian R., Carl L.. Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing, 2004, 1(1): 11～33

[3] http://www-128.ibm.com/developerworks/tw/autonomic/

[4] 何华，State-based 操作系统体系架构下的资源调度问题的研究与设计，西安电子科技大学软件工程研究所，内部资料，2005

[5] 李航，基于状态的操作系统模型，西安电子科技大学软件工程研究所，内部资料，2004.6

[6] Hang Li, Xi-Yang Liu, and Ping Chen. Octopus Architecture: A new attempt to achieve reliable OS. In WICSA 2005.11

[7] V. Raghavan, Challenges in Fault Adaptive Systems, Fault Adaptive Systems Technology (FAST), SRDS Workshop, DARPA/ITO, Oct. 2001.

[8] Shooman, Martin L. Reliability of computer systems and networks : fault tolerance, analysis and design. New York : Wiley, c2002.

[9] Pradhan, Dhiraj K. Fault-tolerant computer system design. Upper Saddle River, N.J. : Prentice Hall PTR, c1996.

[10] Abraham Silberschatz , Peter Baer Galvin, Greg Gagn, Operating System Concept , Sixth Edition.高等教育出版社, 2002

[11] Andew S.Tanenbaum, Albert S.Woodhull 著.王鹏,尤晋元等译校, Operating Systems Design Implementation, Second Edition. 电子工业出版社, 1998

[12] C.M.Krishna, Kang G.Shin 著 戴琼海译.Real-Time Systems. 清华大学出版社. 2004.9

[13] Katsuhiko Ogata 著;卢伯英,于海勋等译. Modern Control Engineering. 电子工业出版社. 2000

[14] 疏松贵，计算机控制系统理论与应用，科学出版社，1988

[15] Stankovic JA, Lu C, Son SH, Tao G. The case for feedback control real-time scheduling. In: Werner B, ed. Proc. of the 11th Euromicro Conf. on Real-Time Systems. Los Alamitos: IEEE Computer Society. 1999. 11~20.

[16] 亨德森,匡特著. 苏通译. 中级微观经济理论：数学方法. 北京大学出版社, 1988

[17] Stephen Boyd, Lieven Vandenberghe, Convex optimization. New York : Cambridge

University Press, 2004.

[18] J. K. MacKie-Mason and H. R. Varian. Pricing the Internet. In B. Kahin and J. Keller, editors, Public Access to the Internet, pages 269–314. MIT Press,Cambridge, MA, USA, 1995.

[19] P. Key, D. McAuley, P. Barham, and K. Laevens. Congestion pricing for congestion avoidance. Technical Report MSR-TR-99-15, Microsoft Research, Cambridge, U.K., February 1999

[20] F. Kelly, A. Maulloo, and D. Tan. Rate control in Communication Networks: Shadow Prices, Proportional Fairness and Stability. Journal of the Operational Research Society, 49(3):237~252, March 1998.

[21] Rolf Neugebauer and Derek McAuley. Congestion prices as feedback signals: An approach to QoS management. In Proceedings of the 9th ACM SIGOPS European Workshop, pages 91~96, Kolding, Denmark, September 2000

[22] Shuichi Oikawa and Ragunathan Rajkumar. Scalability in a Real-Time Kernel. The 4th International Workshop on Real-Time Computer Systems Applications October 27-29. 1997.

[23] Labrosse Jean. μC/OS-II The Real-Time Kernel. Second Edition. CMP Books, 2003

[24] Arthur Griffith, 胡恩华, GCC 技术参考大全, 清华大学出版社, 2004

[25] http://bochs.sourceforge.net/doc/docbook/index.html

[26] http://www.gnu.org/software/grub/manual/multiboot/

[27] IA-32 Intel® Architecture Software Developer's Manual, Valume1: Basic Architecture. 2000

[28] IA-32 Intel® Architecture Software Developer's Manual, Valume3: System Programming Guide. 2000

[29] 毛德操，胡希明，Linux 内核源代码情景分析，浙江大学出版社，2001

[30] 赵炯，Linux 内核完全注释，机械工业出版社，2004

[31] Randal E.Bryant, David, O'Hallaron, Computer Systems A programmer's Perspective, 电子工业出版社, 2004

在读期间所做工作

科研工作：

参加西安电子科技大学软件研究所“十五”国防预研项目——可信软件重用及构件库管理系统（编号 41310501）。

发表论文：

闫园，李航，刘西洋 一个自治系统资源管理的研究与设计 . 西安电子科技大学 2005 年学术年会，已录用