高级操作系统

Advanced Operating System

Distributed Systems

Concepts and design

朱 青 Qing Zhu,

Department of Computer Science, Renmin University of China

zqruc2012@aliyun.com

Chapter 5

Operating System Support 操作系统支持

Teacher: Qing Zhu 朱 青

Department of Computer Science,

Information School,

Renmin University of China

zqruc2012@aliyun.com

Textbook

- **Distributed Systems – Concepts and Design,**
  George Couloris, Jean Dollimore, and Tim Kindberg
  Edition 3,4, © Pearson Education 2001, 2005

- **The lecture is based on this textbook.**

第5章 操作系统支持

- 5.1 简介
- 5.2 操作系统层
- 5.3 保护
- 5.4 线程
- 5.5 线程与进程
- 5.6 通信与调用
- 5.7 虚拟化
- 5.8 代码迁移
- 5.9 处理器任务分配

5.1 简介 Introduction

- Distributed systems act as resource managers for the underlying hardware, allowing users access to memory, CPUs, peripheral devices （外围设备）, and the network. Much of this is accomplished by operating systems and network operating systems.
- 分布式系统作为硬件层之上的资源管理器，容许用户访问存储器， CPU，外围设备和网络，这是由操作系统和网络操作系统支持完成的。

Virtual machine 虚拟机

- Because multiple processes can run at the same time on a hardware device, an operating system provides a virtual machine, giving the user the impression of control of a system. This includes protecting each user process from interference by another process.
- 由于多处理器可以同时运行在一个硬件设备上，操作系统可以提供一个虚拟机，给用户一个系统控制的感觉。这包括保护每个用户进程避免其他进程的干扰。
- To accomplish this, operating systems typically use two levels of access to the system resources. The OS has access to everything, while user processes are more limited.
- 操作系统使用两级访问系统资源，当用户进程受限时，操作系统可以访问这些设备。

Network operating system网络操作系统

- Both UNIX and Windows are examples of network operating systems.
- UNIX和WINDOWS NT 都是网络操作系统的例子
- The defining characteristic is that the nodes running a network operating system retain autonomy(自治) in managing their own processing resources. In other words, there are multiple system images, one per node.
- 网络操作系统的独特的特点：运行于其上的结点能独立的管理自己的进程资源。即，在网络上的每一个结点上都有系统映像。

Distributed operating system分布式操作系统

- One could envisage(正视) an operating system in which users are never concerned with where their programs run, or the location of any resources. There is *a single system image*.
- 可以想象存在一种用户不用关心程序的运行地点或资源位置的操作系统。只有一个单一系统映像。
- An operating system that produces a single *system image* like this for all the resources in a distributed system is called a *distributed operating system* .
- 如果一个操作系统像这样对分布式系统中的所有资源只生成单一的系统映像，那么，这个系统就是一个分布式系统。

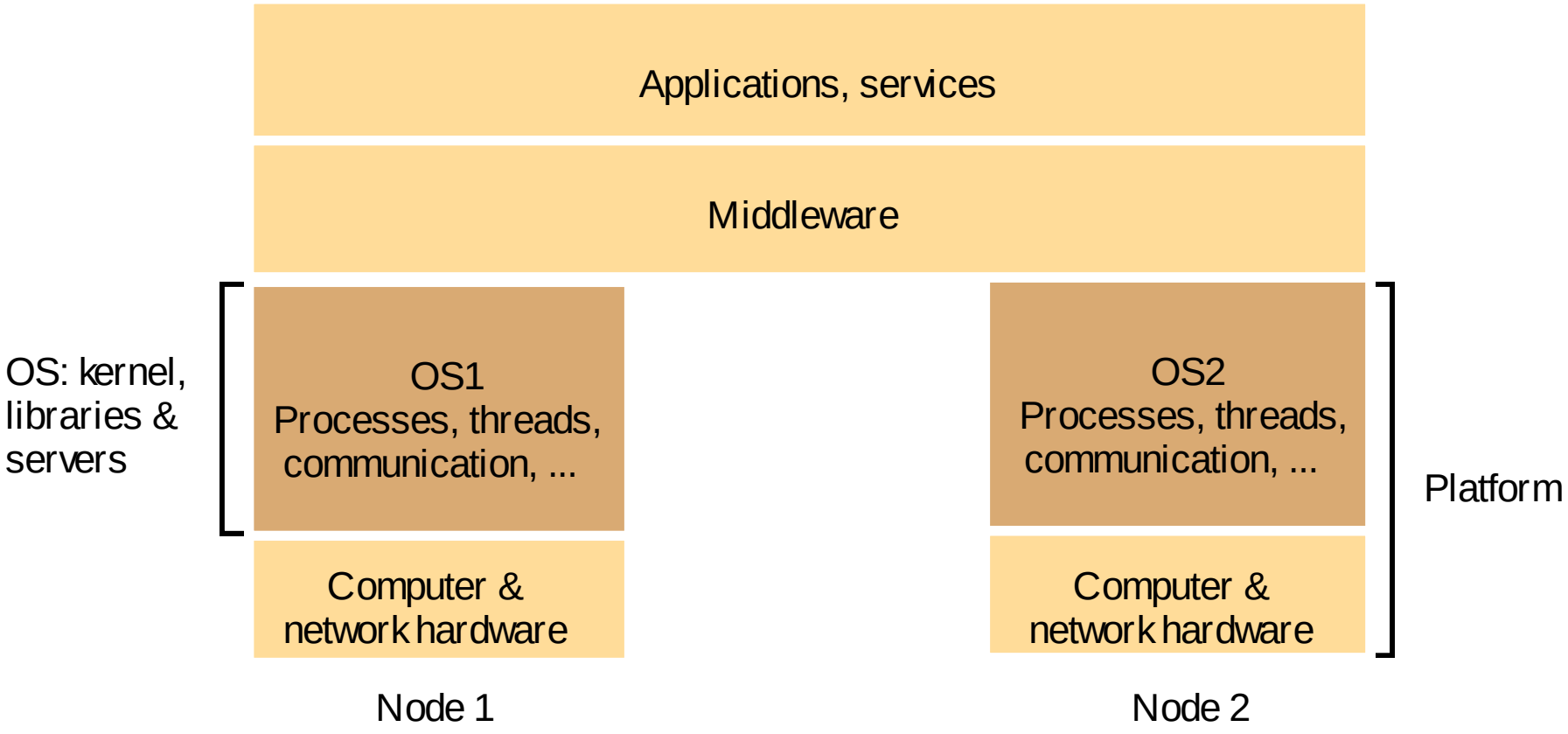
Distributed operating system 分布式操作系统

- The combination of middleware and network operating systems provides an acceptable balance between the requirement for autonomy, on the one hand, and network-transparent resource access on the other.
- 中间件和网络操作系统的结合提供了独立性和网络资源的透明访问之间的平衡。
- The network operating system enables users to run their favorite word processor and other standalone applications.
- 网路操作系统使用户能运行字处理程序和其它独立的应用程序
- Middleware enables them to take advantage of services that become available in their distributed system.
- 中间件使用户能享受到分布式系统提供的服务。

第5章 操作系统支持

- 5.1 简介
- 5.2 操作系统层
- 5.3 保护
- 5.4 线程
- 5.5 线程与进程
- 5.6 通信与调用
- 5.7 虚拟化
- 5.8 代码迁移
- 5.9 处理器任务分配

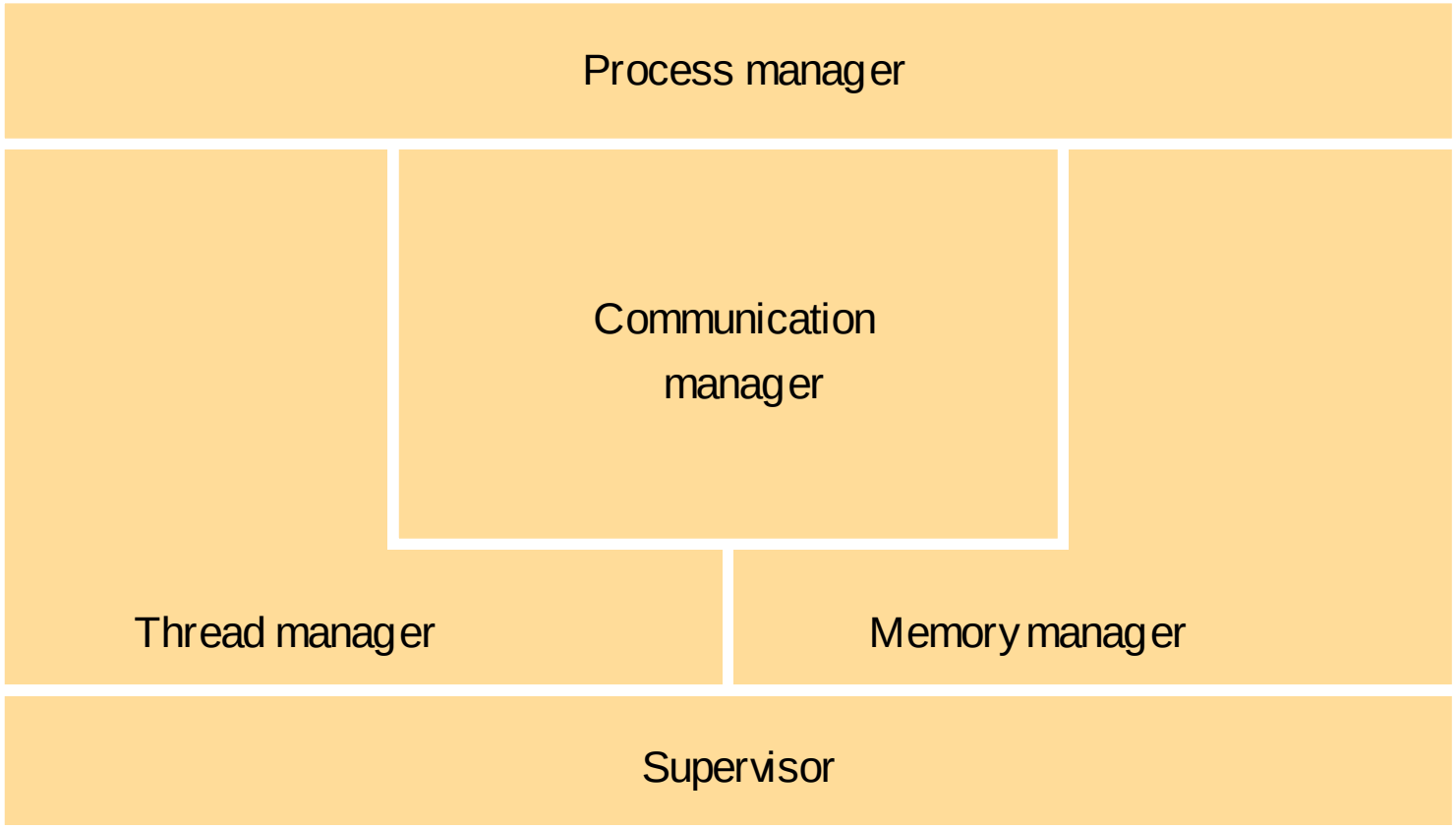
5.2 操作系统层 System layers



两个结点上的操作系统层是如何支持一个公共的中间件层的，从而，为应用和服务提供一个分布式基础设施。

- Kernels and server processes are required at least the following of them:内核和服务器应具备的特点：
- Encapsulation 封装：提供一个有用的服务接口以防问物理资源，资源服务的细节对用户是隐藏的。
- Protection 保护：资源需要被保护以防止非法访问。
- Concurrent processing 并发处理：客户应该可以共享和并发地访问资源。
- Communication 通信：资源管理器在网络上或计算机内接收操作参数并返回结果。
- Scheduling 调度：当调用一个操作时，在内核和服务器上调用其操作。

Figure 6.2
Core OS functionality 操作系统内核功能



Core O/S Functions

- Process management includes creating, managing and destroying processes. Every process has an address space and one or more threads.
- 进程管理：创建、管理和撤销进程。每一个进程有它的定位空间和一个或多个线程
- Thread management includes creating, synchronizing and scheduling threads.
- 线程管理：创建、同步和调度线程。
- Communications management includes all communications between threads in the same computer and may include remote processes.
- 通信管理：同一计算机中线程之间的通信，可以包括远程进程。
- Memory management includes control of physical and virtual memory.
- 内存管理: 物理内存和虚拟内存的控制

The O/S Supervisor 操作系统监督器

- The Supervisor： dispatches interrupts, system call traps and exceptions, the control of memory management unit and hardware caches, and manipulation of the processor and floating point registers.
- 监督器：负责处理中断、系统调用陷阱和其他异常，同时控制内存管理单元和硬件缓存以及处理器和浮点寄存器操作。

第5章 操作系统支持

- 5.1 简介
- 5.2 操作系统层
- 5.3 保护
- 5.4 线程
- 5.5 线程与进程
- 5.6 通信与调用
- 5.7 虚拟化
- 5.8 代码迁移
- 5.9 处理器任务分配

5.3 保护 Protection

- All resources must be protected from interference.
- 所有资源必须被保护以防止干扰。
- Note that the threat to a system's integrity does not come only from maliciously contrived code. Benign code that contains a bug or which has unanticipated behavior may cause part of the rest of the system to behave incorrectly.
- 对系统完整性的威胁并不仅仅来自于恶意编制的程序代码。非恶意编制的代码也有可能因为存在某些错误或具有未预料的行为而导致系统工作异常
- Protection might also prevent the bypassing of required activities such as login authentication and authorization.
- 保护也考虑注册鉴定和授权。

Kernels and Protection

- Most processors have a hardware mode register that permits privileged instructions to be strictly controlled. Generally, there is a supervisor mode for privileged instructions and a user mode for unprivileged instructions.
- 大多数处理器都有硬件模式的寄存器，用来决定能否执行特权指令。通常，有一个监督模式用于授权指令和用户模式用于非授权指令。
- Separate address spaces are allocated, and only the privileged kernel can access the privileged spaces. Usually a system call trap is required to access privileged instructions from user space.
- 地址空间被分开，只有特权内核可以访问特权空间。通常一个系统调用的陷阱需要从用户空间访问授权指令。

Protection Overhead

- Protection comes at a price, including the processor cycles to switch between address spaces, the supervision and protection of system call traps, and the establishment, authentication and authorization of privileged users and processes.
- 保护需要系统开销，包括：地址空间之间的切换占用处理器的处理周期，系统调用陷阱的监督和保护，对于授权用户和进程授权的制定和鉴定。
- Since all overhead consumes expensive resources, it is always a key concern of IT managers.
- 因为这些消耗大量资源，在IT管理中这是值得关心的问题。

第5章 操作系统支持

- 5.1 简介
- 5.2 操作系统层
- 5.3 保护
- 5.4 线程
- 5.5 线程与进程
- 5.6 通信与调用
- 5.7 虚拟化
- 5.8 代码迁移
- 5.9 处理器任务分配

5.4 线 程（thread）

- 1线程（thread）的引入
- 2线程分类
- 3多核与多线程

线程（thread）的引入

- 进程（process）=>进程映像=代码、数据、栈、PCB
- 进程概念的两个特点：
  - 资源分配的单位：资源的控制或所有权属于进程。操作系统执行保护功能，以防止进程之间发生不必要的与资源相关的冲突
  - 调度/执行的单位：进程沿着一条执行路径（轨迹）执行。其执行过程可能与其他进程的执行过程交替进行

线程

- 传统操作系统中这两个特点集中地实现于进程机制
- 两个特点本质上独立，可分开处理：
  - 用进程（任务）作为资源所有权单位(the unit of resource ownership),
  - 用线程（thread）/ LWP（Light Weight Process，轻量级进程）作为调度/执行/分派单位(the unit of dispatching)

线 程

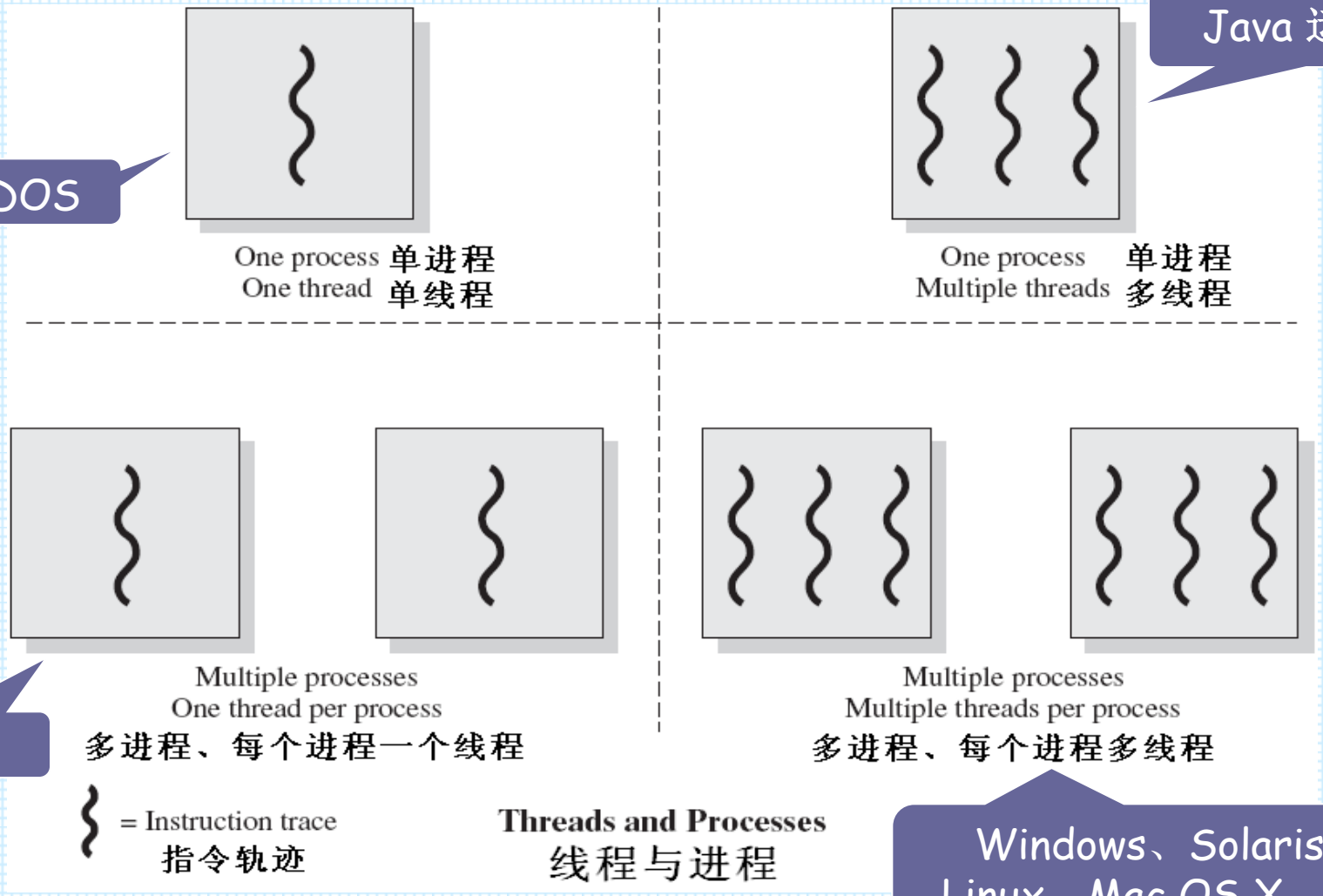
- 在没有线程概念的系统中，进程是
  - 资源分配的单位
  - 调度/执行的单位
- 主要问题1：
  - 进程切换的开销大——每次切换都要保存和恢复进程所拥有的全部信息(PCB、有关程序段和相应的数据集等)

线 程

- 主要问题2：
  - 进程占用的资源多——多个同类进程需占用多份资源，而一个进程中的多个同类线程则共享一份资源
- 线程：一个进程内的基本调度单位
  - 一个进程可有一个或多个线程

多线程（Multithreading）

- 多线程：是指OS支持在一个进程中执行多个线程的能力



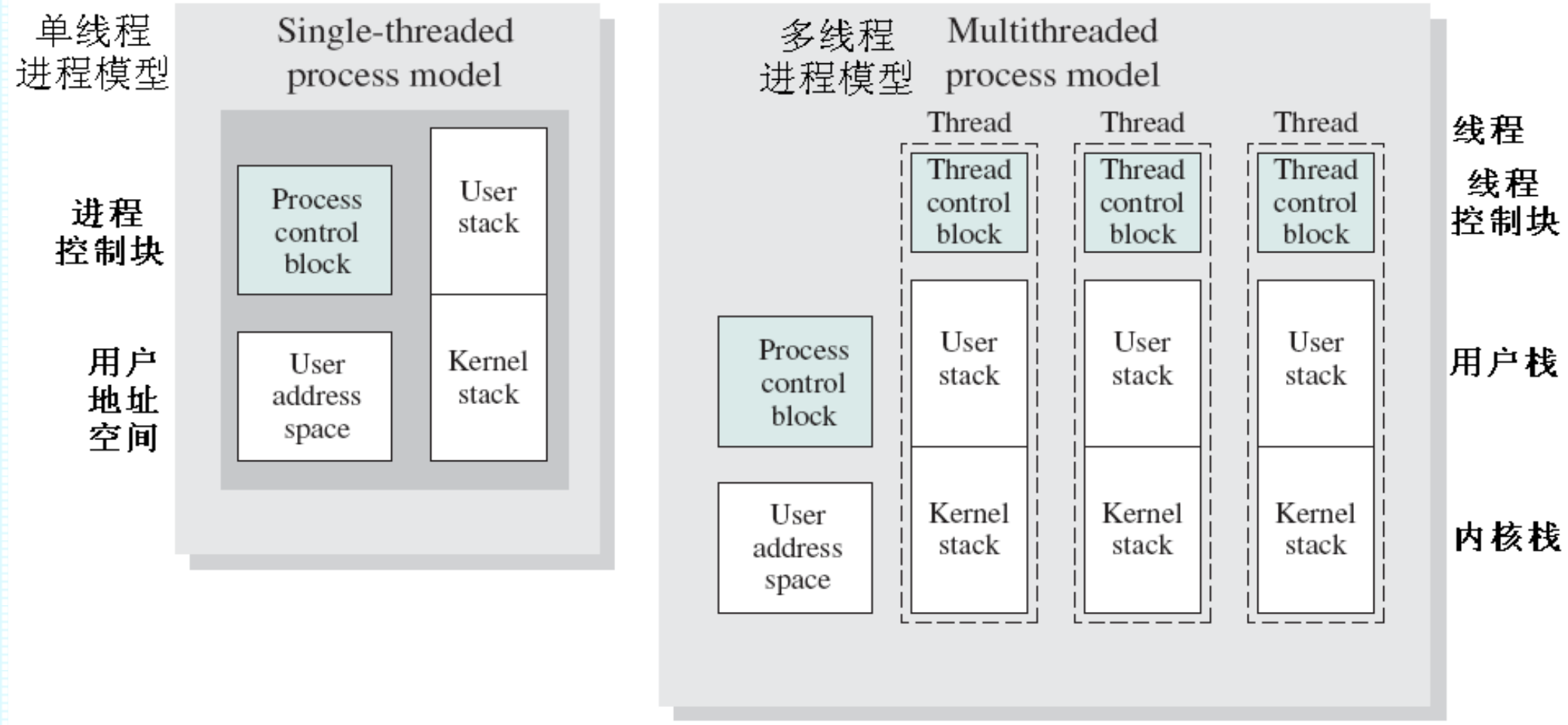
多线程环境下的进程和线程

- 进程/任务：资源分配和保护的单位
  - 拥有用于保存进程映像的虚地址空间
  - 受保护地访问处理器、其他进程、文件和I/O资源

多线程环境下的进程和线程

- 线程：分派的执行单位
  - 执行状态（运行、就绪等）
  - 保存的线程上下文（非运行时）
  - 一个执行栈
  - 独立的用来存储局部变量的静态存储空间
  - 对进程的内存和其他资源的访问（与同一进程内的其他线程共享这些资源）

单线程和多线程的进程模型



Single-Threaded and Multithreaded Process Models

单线程与多线程进程模型

线程的优点(与进程比较)

- 创建速度快（在已有进程内）
- 终止所用的时间少
- 切换时间少（保存和恢复工作量小）
- 通信效率高（在同一进程内，无需调用OS内核，可利用共享的存储空间）

线程的应用

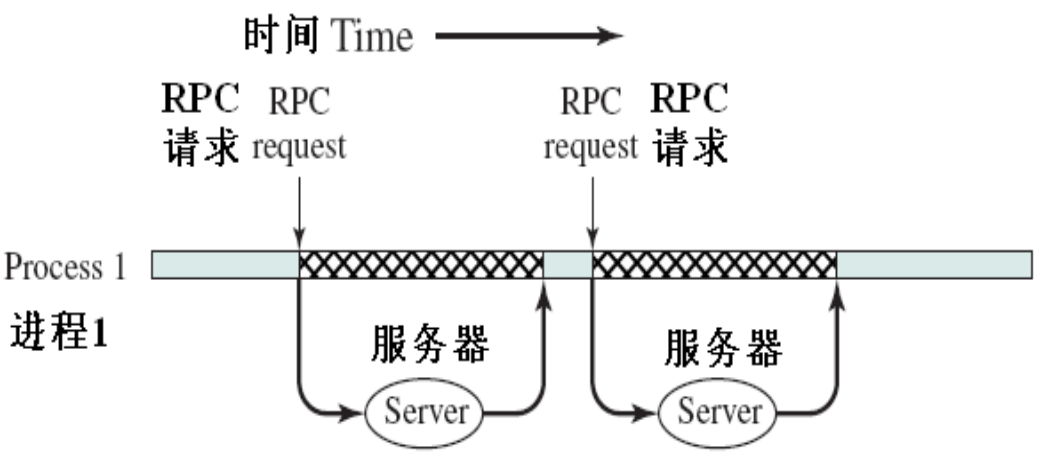
- 若应用程序可按功能划分成不同的小段，或可划分成一组相关的执行实体，则用一组线程（比用一组进程）可提高执行效率（尤其是在多处理器和多核系统中）
- 单用户多核/多处理器系统的典型应用：
  - 服务器中的文件管理或通信控制
  - 前台和后台操作（如前台与用户交互、后台更新数据）
  - 异步处理（如字处理与周期性备份）
  - 加速执行（在多核/多处理器系统中的并行）
  - 模块化程序结构（涉及多种活动或多个I/O源和目的地的程序）

线程的功能特性（执行特征）

- 线程不拥有资源，且进程与其所有线程共享代码和地址空间。但是线程拥有自己的寄存器上下文和栈空间（用来存储局部变量和调用参数）
- 挂起状态、终止状态是进程级的概念
  - 挂起一个进程，则该进程的所有线程也挂起（共享地址空间）
  - 终止一个进程，则该进程的所有线程也终止（共享代码段）

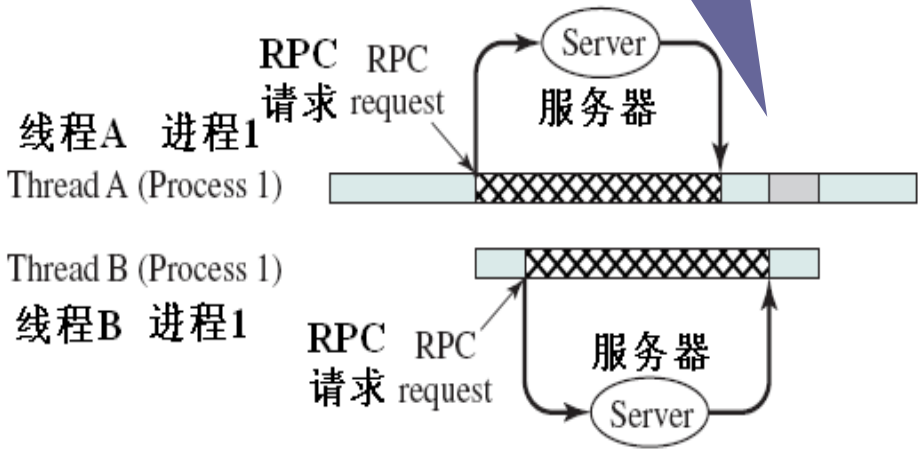
例1：使用线程的RPC

一个线程阻塞不会导致整个进程阻塞



(a) RPC using single thread

使用单线程的RPC



(b) RPC using one thread per server (on a uniprocessor)

每个服务器使用一个线程的RPC（在单处理器上）

Blocked, waiting for response to RPC 阻塞，等待RPC响应

Blocked, waiting for processor, which is in use by Thread B 阻塞，等待正在被线程B使用的处理器

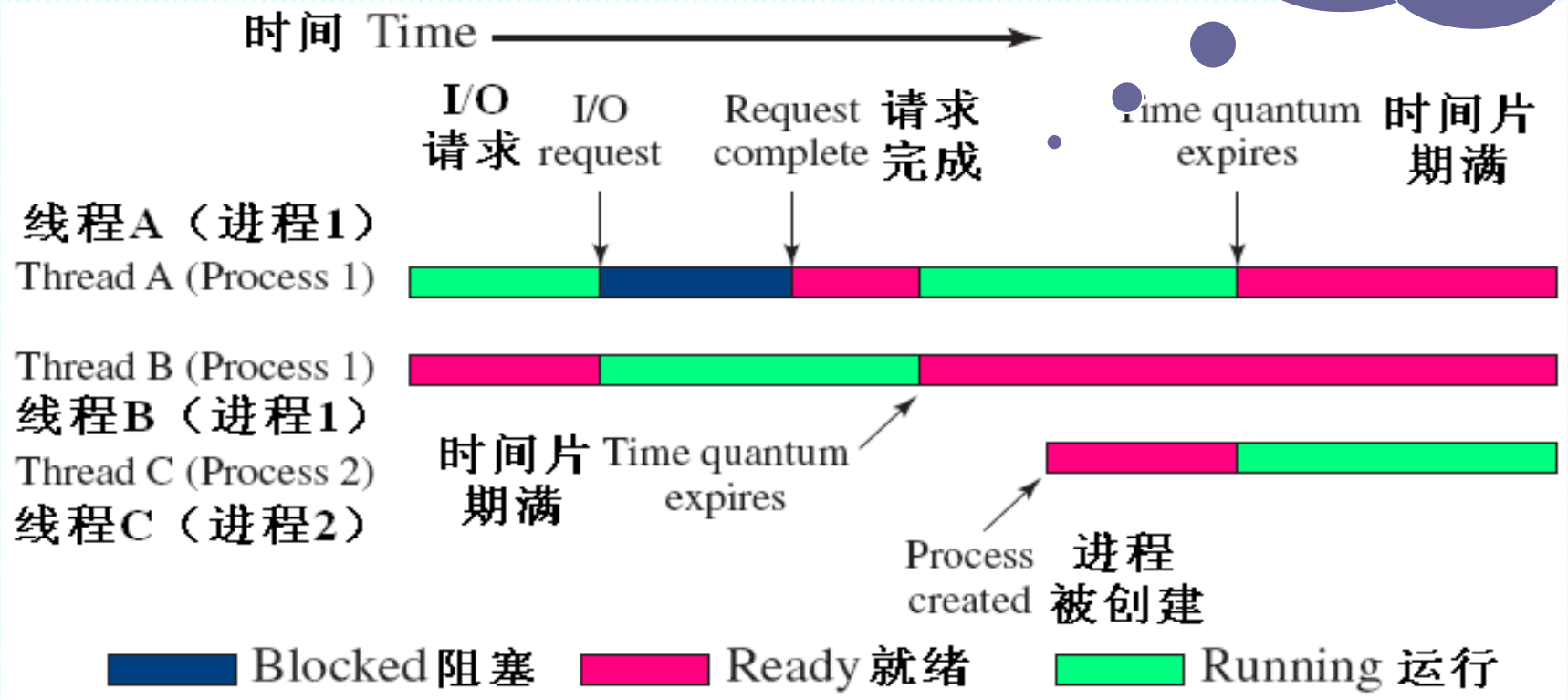
Running 运行

Remote Procedure Call (RPC) Using Threads

使用线程的远程过程调用（RPC）

例2：单个单核处理器上的多线程

多个进程中的多个线程可交替执行



Multithreading Example on a Uniprocessor

单处理器上的多线程例子

线程同步

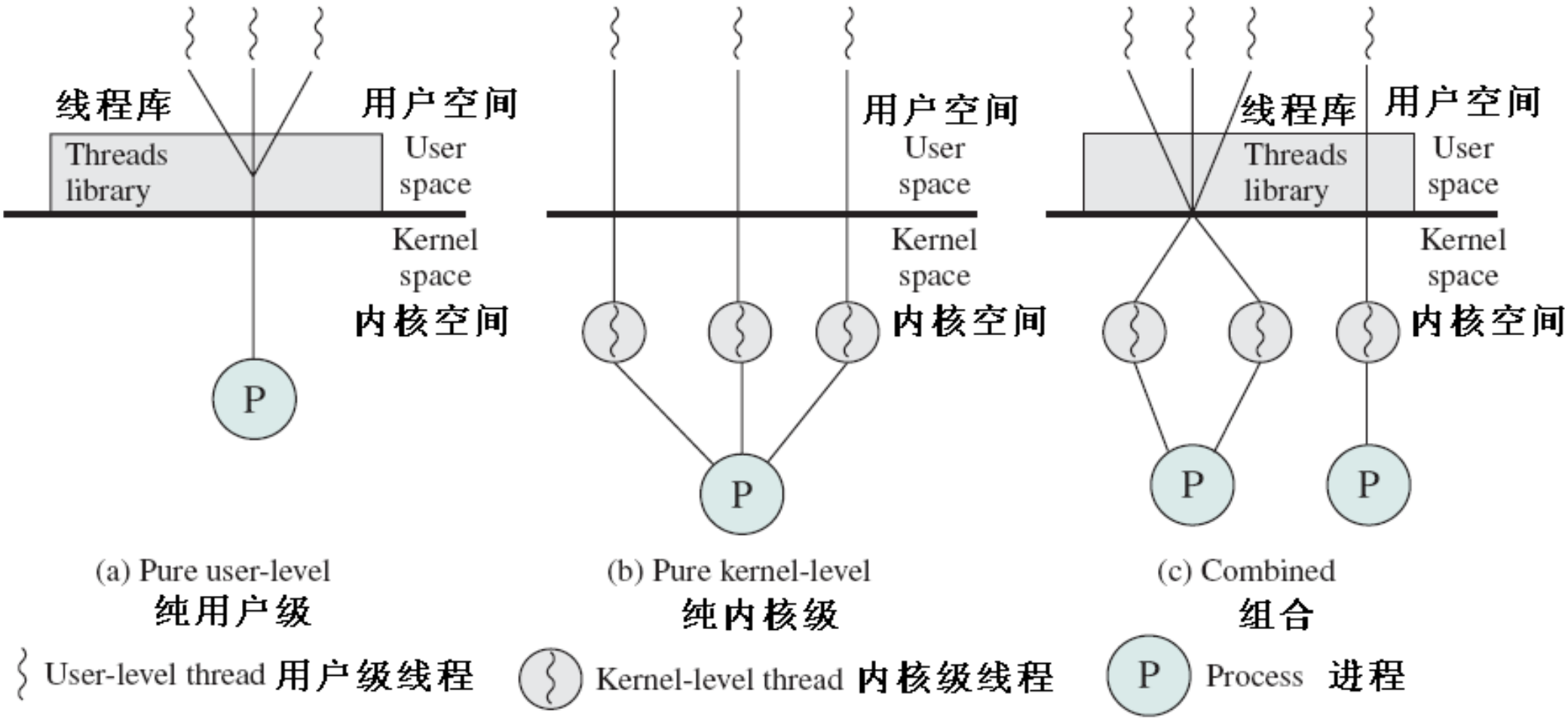
- 同一进程的多个线程共享同一个地址空间和其他资源
- 线程同步
  - 需要对各个线程的活动进行同步，以便它们互不干涉且不破坏数据结构
  - 线程同步机制与进程同步机制相同

线程分类

用户级与内核级线程

- 用户级线程(User-level Threads, ULT)
- 内核级线程(Kernel-level Threads, KLT)
- 混合方法/组合途径(Combined Approaches)

线程分类

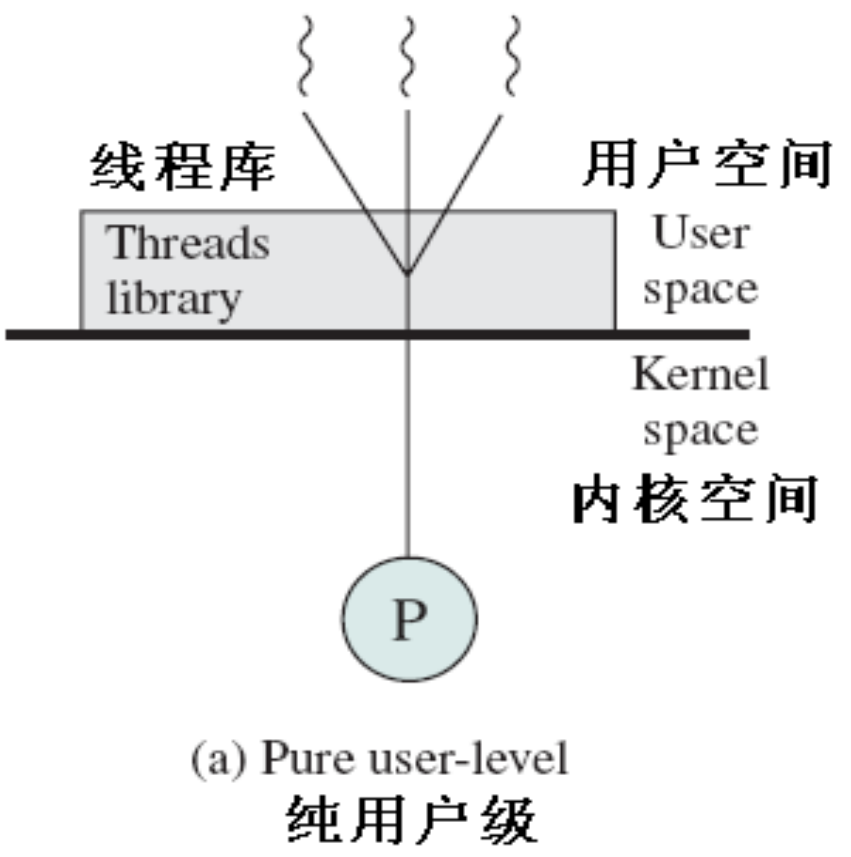


User-Level and Kernel-Level Threads

用户级与内核级线程

用户级线程(ULT)

- 线程管理均由应用程序完成（线程库）
- 内核不知道线程的存在
- 优点：
  - 线程切换不需要模式切换
  - 调度算法可应用程序专用
  - ULT不需内核支持，线程库可在任何OS上运行
- 缺点：
  - 一个线程阻塞会导致整个进程阻塞
  - 不能利用多核和多处理器技术
- 实例：GNU Portable Threads



(a) Pure user-level

纯用户级

线程调度与进程调度

- 以用户级线程为例，说明线程状态变化与进程状态变化的关系。进程B有两个线程，其状态如图a（进程B处于运行态：线程2正在运行，线程1处于就绪状态）
- 可能发生的变化
  - 线程2中执行的应用程序代码执行了系统调用，阻塞了进程B。如图b，但是，线程2是执行状态。
  - 系统调用完成后，时钟中断把控制权交给内核，内核确定当前正在运行的进程B的时间片用完，将进程B转为就绪态，并切换到另一个进程。如图c
  - 线程2执行到某处，它需要进程B的线程1所执行的某些动作的结果。线程2进入阻塞状态，线程1从就绪转换为运行但进程B自身仍保留在运行状态。如图d

线程调度与进程调度

Possible transitions from 4.6a:

4.6a→4.6b
4.6a→4.6c
4.6a→4.6d

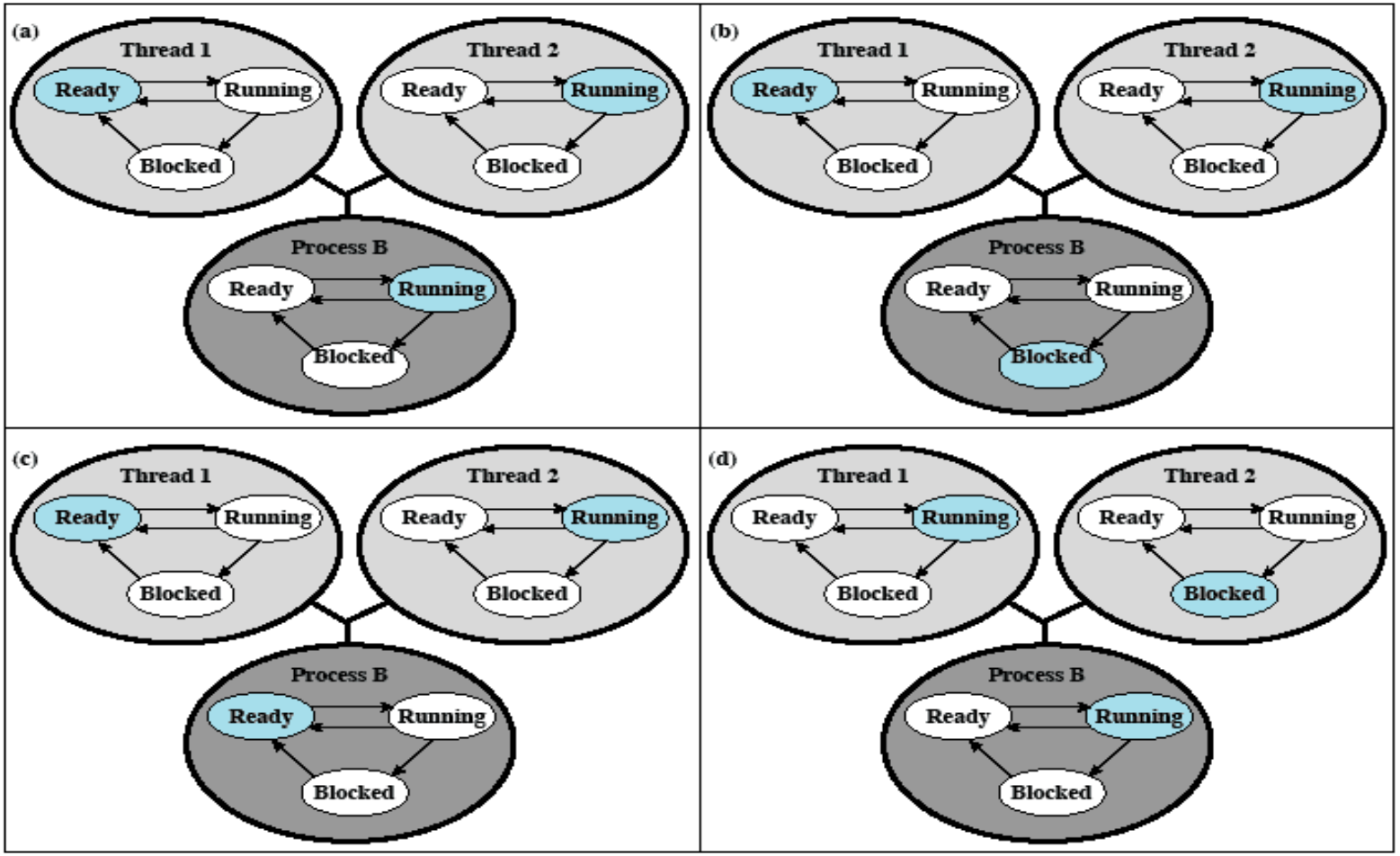


Figure 4.6 Examples of the Relationships between User-Level Thread States and Process States

用户级线程（User Level Thread）

- 由应用程序完成所有线程的管理

  通过线程库(用户空间)

  一组管理线程的过程
- 核心不知道线程的存在
- 线程切换不需要核心态特权
- 调度是应用特定的

线程库

- 创建、撤消线程
- 在线程之间传递消息和数据
- 调度线程执行
- 保护和恢复线程上下文

对用户级线程的核心活动

- 核心不知道线程的活动，但仍然管理线程的进程的活动
- 当线程调用系统调用时，整个进程阻塞
- 但对线程库来说，线程仍然是运行状态

  即线程状态是与进程状态独立的

用户级线程的优点和缺点

优点：

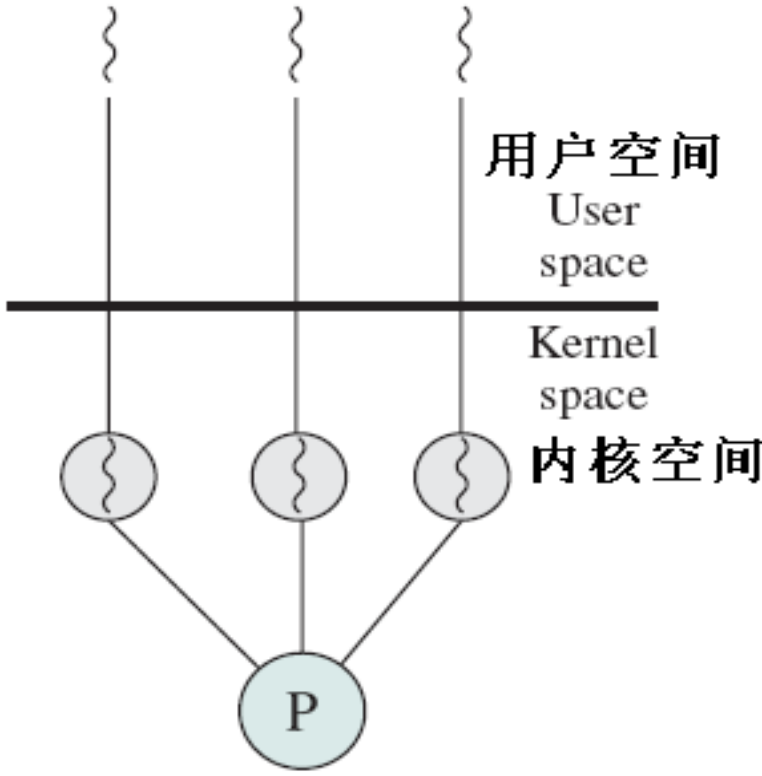
- 线程切换不调用核心
- 调度是应用程序特定的：可以选择最好的算法
- ULT可运行在任何操作系统上（只需要线程库）

缺点：

- 大多数系统调用是阻塞的，因此核心阻塞进程，故进程中所有线程将被阻塞
- 核心只将处理器分配给进程，同一进程中的两个线程不能同时运行于两个处理器上

内核级线程(KLT)

- 线程管理由内核完成（提供API）
- 调度基于线程进行
- 实例：Windows、Linux、Mac OS X、iOS
- 优点：
  - 线程阻塞不会导致进程阻塞
  - 可以利用多核和多处理器技术
  - 内核例程本身也可以使用多线程
- 缺点：
  - 线程切换需要内核的状态切换

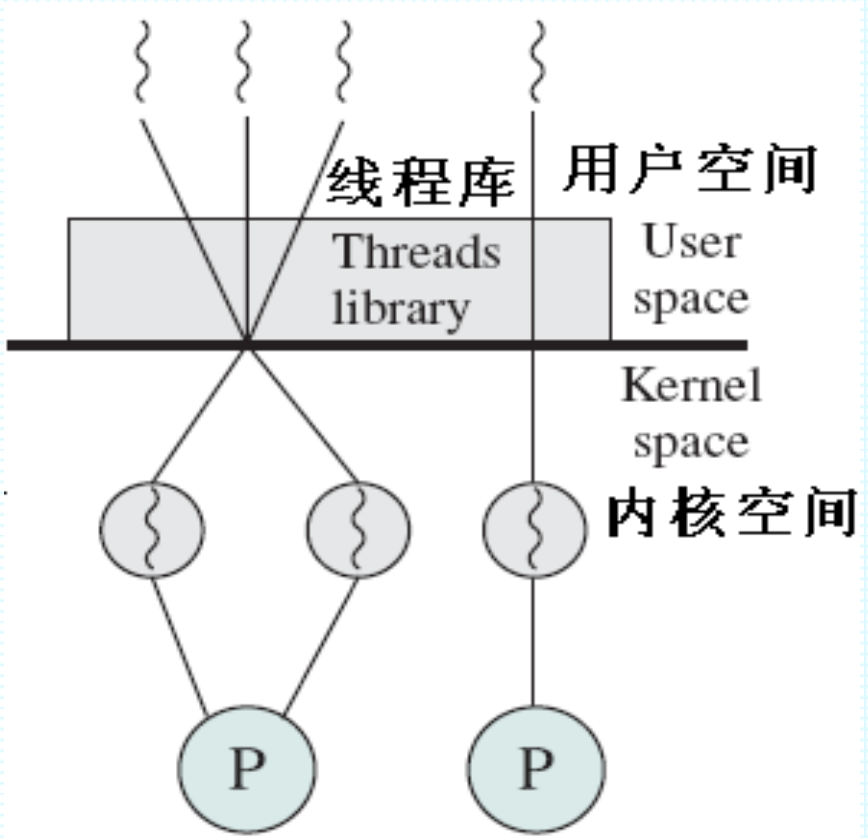


(b) Pure kernel-level

纯内核级

混合方法

- 线程创建在用户空间完成
- 线程调度和同步在用户空间进行
- 应用程序的m个ULT（用户级）被映射到n（≤m）个KLT（内核级）
- 实例：Solaris、Windows 7



(c) Combined

组合

其他方案

线程与进程间的关系

线程:进程	描述	示例系统
1:1	每个进程有唯一线程	DOS、传统Unix
M:1	一个进程可拥有多个线程	Windows NT、Solaris、Linux、Mac OS X、iOS
1:M	一个线程可在多个进程环境中迁移	RS(Clouds)、Emerald（翡翠）
M:N	M:1+1:M	Trix

多核与多线程

- 处理器现在发展到了多核（CPU）甚至众核（GPU）的阶段
  - 2013年XXX研究院提供理论峰值达9.7万亿次的GPU高性能集群计算服务，集群使用了16个6核12线程的Intel至强E5645 CPU处理器、2个4核8线程的Intel至强E5620 CPU处理器、9个448核的NVIDIA Tesla C2050 GPU运算卡
- 对称多处理(SMP)
  - 多个处理器可以执行相同功能（故称“对称”），内核可运行在任一处理器上
  - 每个处理器可从可用进程和线程池完成自身的调度工作
  - 对称的多核是SMP的特例，是片上的SMP

多核系统上的软件性能

- 阿姆德尔（Amdahl）定律

$$\text{加速比} = \frac{\text{单处理器上程序的运行时间}}{\text{在}N\text{个并行处理器上程序的运行时间}} = \frac{1}{(1-f)+\frac{f}{N}}$$

- 其中：N为处理器/核数、f为本质上并行的代码所占的百分比；执行时间的$(1-f)$部分的代码是串行的。
- $N \rightarrow \infty$，加速比$\rightarrow 1/(1-f)$
- 例如：$f=0.9$（即只有10%的代码本质上是串行的）、$N=8$（8核），则加速比≈4.7。$f=0.9$、$N=\infty$时，加速比=10

第5章 操作系统支持

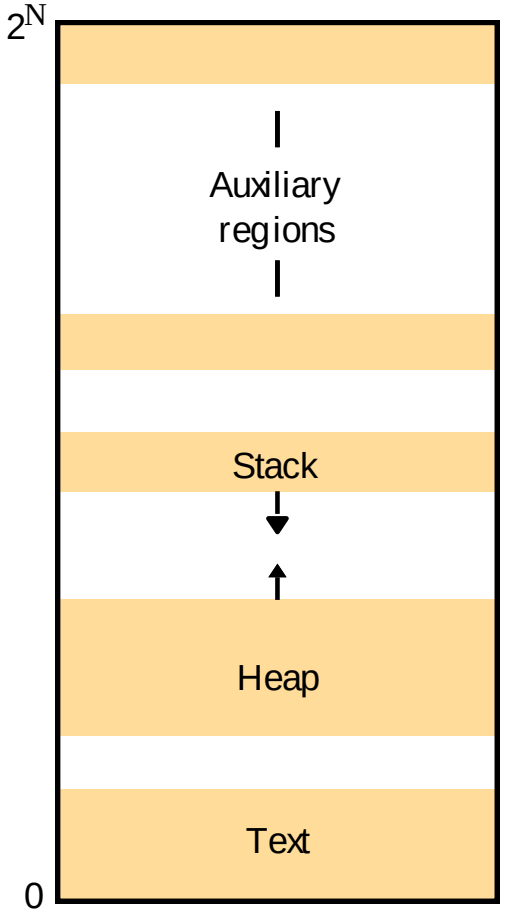
- 5.1 简介
- 5.2 操作系统层
- 5.3 保护
- 5.4 线程
- 5.5 线程与进程
- 5.6 通信与调用
- 5.7 虚拟化
- 5.8 代码迁移
- 5.9 处理器任务分配

5.5 线程与进程

Process

- A typical process includes an execution environment and one or more threads.
- 一个典型的进程包括一个执行环境和一个或多个线程。
- An execution environment includes an address space,
- 执行环境包括地址空间。
- thread synchronization and communication resources such as semaphores and communication interfaces such as sockets,
  - 线程同步和资源通信，例如，信号量，基于socket的通信界面，
- and higher level resources such as open files and windows.
  - 高级资源，如打开文件按和窗口。
- Since earlier processes may have allowed only a single thread, the term multi-threaded process is often used for clarity.
- 因为早期的进程只容许唯一的线程，多线程进程的术语以示区别。

Figure 6.3
Address space 地址空间



辅助空间

栈

堆

正文(固定不可变)

Address Spaces

- As shown on the previous diagram, an address space is a management unit for the virtual memory of a process. It consists of non-overlapping regions accessible by the threads of the owning process.
- 地址空间是进程虚存管理单元。是一个可以被进程的线程访问的连续的内存区，它由非重叠区间组成。
- Each region has an extent (lowest virtual address and size), read/write/execute permissions for the processes threads, and whether it grows up or down. 每个区域包含如下性质：
  - 范围（最低的虚拟内存地址和区域大小）
  - 对本进程的线程的读/写/执行权限
  - 是否能够向上或向下扩展
  - It is page oriented, and gaps are left between regions to allow for growth.
  - 此模型是基于页的，而不是基于段的。

Unix Address Spaces

- Address spaces are a generalization of the Unix model, which had three regions:
  - A fixed, unmodifiable text region containing program code
  - A heap, extensible toward higher virtual addresses
  - A stack, extensible toward lower virtual addresses
- An indefinite number of additional regions have since been added.
- UNIX模型分为三个地址区间：
- 正文区：程序代码区
- 堆：向高地址虚拟扩展。
- 栈区：向低地址虚拟扩展。
- 不确定值的附加区域

Stack 栈

- Generally there is a separate stack for each thread. Whenever a thread or process is interrupted, status information is stored on the stack that will permit the process or thread to continue from the point at which it was interrupted.
  通常，每个线程有一个独立的栈。当进程或线程中断时，状态信息被存储在栈中。容许进程或线程从这个断点继续执行。 Usually, memory allocated to the stack is recovered when the process or thread retrieves the information and resumes, as interrupts occur and resume in a last-in, first out (LIFO) manner.
- 当进程或线程恢复信息时，内存分配栈，恢复信息状态。中断发生和恢复是采用先进后出的栈式。

File Regions文件区间

- A file stored offline can be loaded into active memory. A mapped file is accessed as an array of bytes in memory. Such a file can reduce access overhead dramatically, as it is orders of magnitude faster to access memory than disk files.
- 离线文件被放在活动内存中
- 映射文件：是一个在内存中可访问的字节数组。当它快速访问内存而不是磁盘，文件可以减少动态访问。

Shared Memory Regions共享内存区

- Sometimes it is desirable to share memory between processes, or between a process and the kernel. Reasons for sharing memory include:
- 进程之间共享内存、或进程和内核之间。共享内存的原因：
- Libraries that might be large and would was memory if each process loaded a copy
- 库：如果每一个进程都复制到内存，占用内存空间太大。
- Kernel calls that access system calls and exceptions
- 内核：访问系统调用和异常处理
- Data sharing and communication between processes on shared tasks
- 共享工作的进程之间，数据共享和通信

Process Creation进程的创建

- An operating system creates processes at need. In a distributed environment, there are two independent aspects of the creation process:
- 操作系统在需要时创建进程。在分布式环境中，新进程的创建有两个独立的方面：
- Choice of a target host
- 选择目标主机。
- Creation of an execution environment and an initial thread within it
- 创建可执行环境和一个初始化线程。

Choosing Host for Process进程主机的选择

- A transfer policy decides whether to situate the process locally or remotely.
- 移动策略：决定运行在本地还是运行在远程。
- A location policy determines which node should host the new process. Location policies may be static or adaptive. Static policies ignore the current state of the system and are designed based on the expected long-term characteristics of the system.
- 定位策略：决定哪个结点作为新进程的主机。定位策略可以是静态的或是可适应的。静态策略忽略系统当前状态，其设计是基于系统长期特征的。

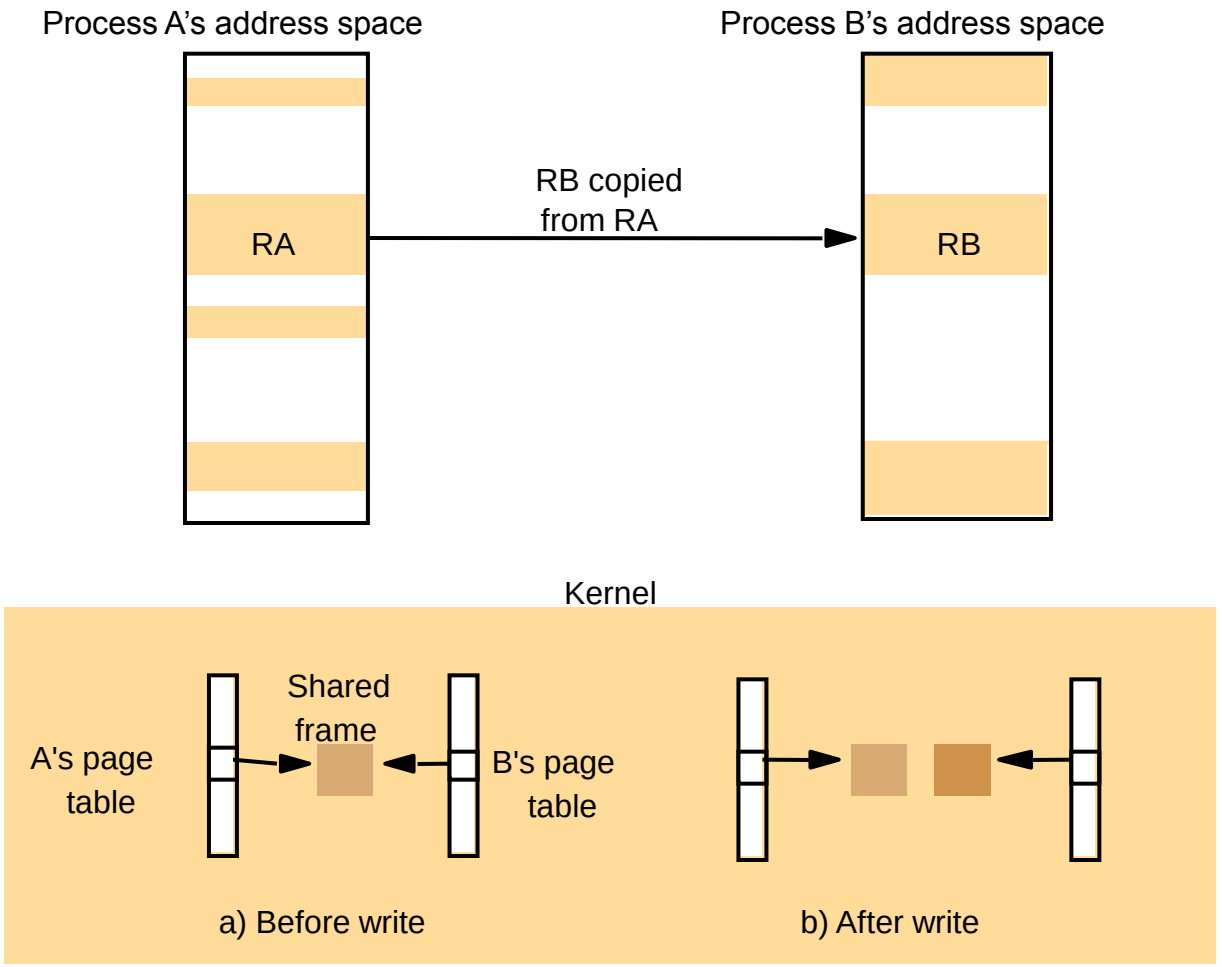
Dynamic Location Policies动态定位策略

- Load sharing policies use a load manger to allocate processes to hosts.
- 负载共享策略：使用负载管理器分配进程给主机。
- Sender-initiated policies require the node creating the process to specify the host.
- 发送方启动的策略：需要结点创建进程并指定主机。
- Migratory policies can shift processes between hosts at any time.
- 迁移策略：在任何时间主机间可以迁移进程。

Creating Execution Environment创建执行环境

- Once a host is selected, a new process requires an address space with initialized contents and default information such as files. The new address space can be defined statically, or copied from an existing execution environment.
- 一旦选择了主机，新进程需要一个包含地址空间和初始化信息的执行环境。新的地址空间可以是静态的(Create)，或者是已存在的执行环境的复本（fork）。
- If copied, content may be shared and nothing written to the new environment until such time as a write instruction occurs for either process. Then the shared content is divided. This technique is called copy-on-write (next slide).
- 如果是复本，那么共享内容被划分，这种技术叫写时复制。

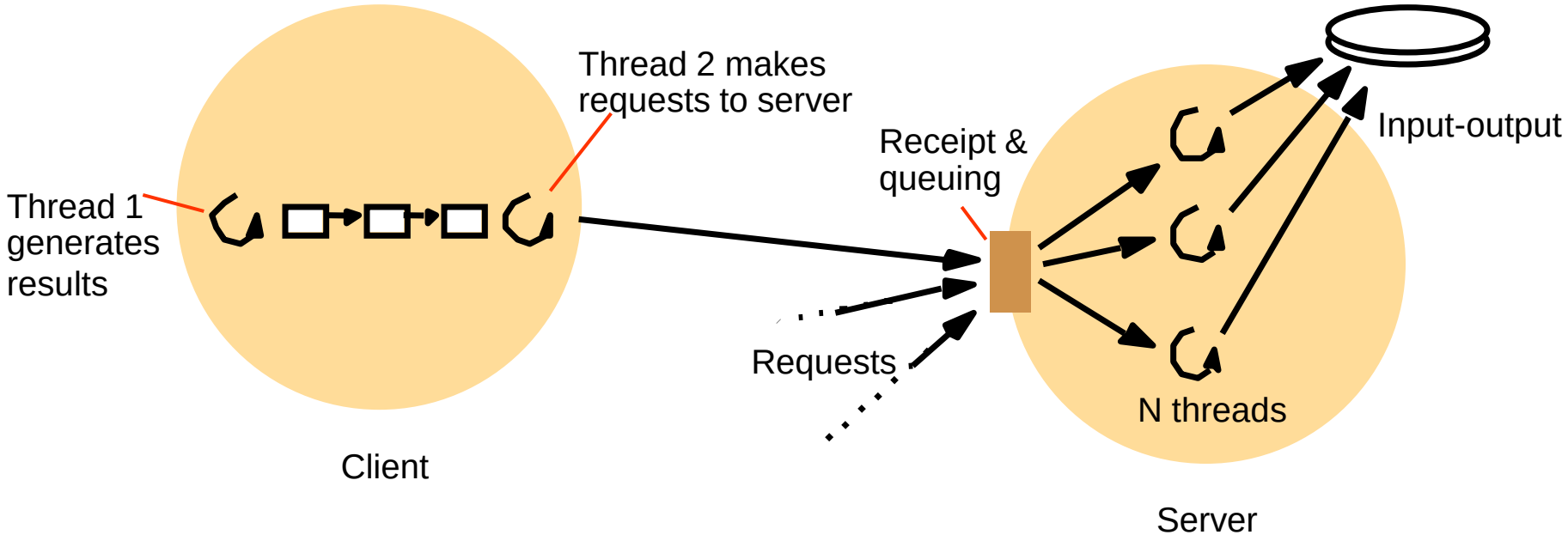
Figure 6.4
Copy-on-write写时复制



Threads线程

- A single process can have more than one activity threads at the same time. For example, it may be performing an activity while at the same time needing to be aware of background events. There may also be background activities such as loading information into a buffer from a socket or file. Servers may service many requests from different users at the same time.
- 一个进程可以同时有一个或多个线程。例如，同时执行一个背景活动，如，从socket 或文件传送信息到存储器。在同时，服务器可以服务来自不同用户的许多请求。

Figure 6.5
Client and server with threads



Multi-threaded Server Architectures多线程服务器结构

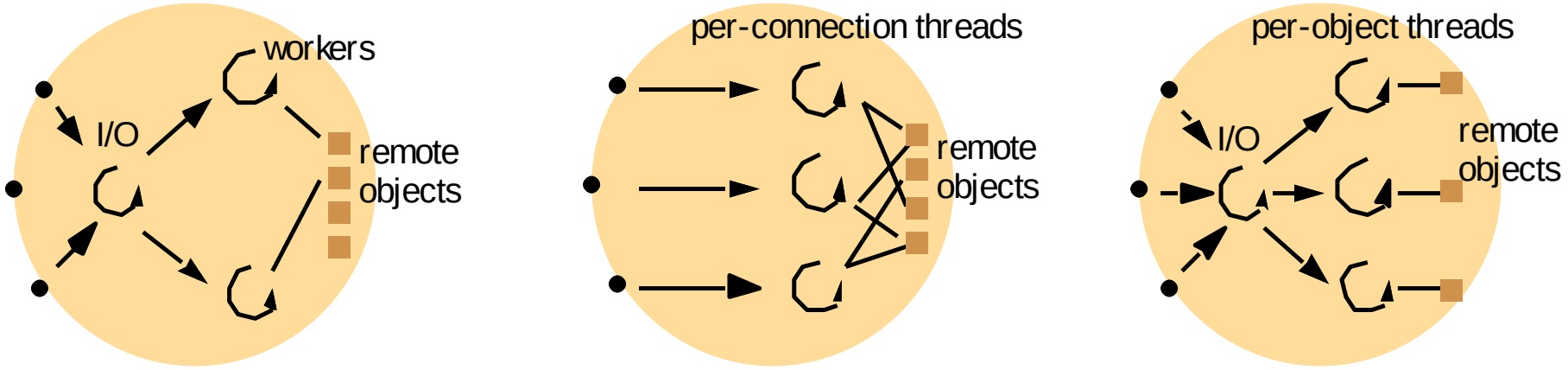
- Worker Pool Architecture—a predetermined fixed number of threads is available for use as needed and returned to the pool after use.
- 工作池体系结构：由服务器创建一个固定的“工作”线程池以便处理请求。
- Thread-per-request Architecture—a new thread is allocated for each new request, and discarded after use.
- 请求线程体系结构：为每一个请求分配一个新的线程，工作线程完成后，会自动终止。

Multi-threaded Server Architectures (2)多线程服务器结构

- Thread-per-connection Architecture—a new thread is allocated for each new connection. Several requests can use that thread sequentially. The thread is discarded when the connection is closed.
- 连接线程体系结构：一个新的线程每次连接时被创建。几个请求可以顺序的使用线程。连接断开线程终止。

- Thread-per-object Architecture—a new thread is allocated for each remote object. All requests for that object wait to use that thread. The thread is discarded when the connection to the object is destroyed.
- 对象线程体系结构：为每个远程对象分配一个新进程。所有的对象请求等待使用线程。当目标连接断开时，线程自动撤销。

Figure 6.6
Alternative server threading architectures (see also Figure 6.5)



a. Thread-per-request
请求线程体系结构

b. Thread-per-connection
连接线程体系结构

c. Thread-per-object
对象线程体系结构

几种服务器线程体系结构

What is a thread? 什么是线程

- It is a programs path of execution.
  - Most programs run as a single thread, which could cause problems if a program needs multiple events or actions to occur at the same time.
  - 线程是程序的执行。大多数程序运行是单进程。如果程序需要多事件或动作同时发生，就会产生问题。
- The Java Virtual Machine allows an application to have multiple threads running concurrently.
- Java虚拟机容许应用同时运行多线程。

What is a thread (cont.)什么是线程

- Multi-threading literally means multiple lines of a single program can be executed at the same time.
- 多线程
  - However, it is different from multi-processing because all of the threads share the same address space for both code and data, causing it to be less overhead.
  - 多线程与多进程的不同因为所有的线程共享相同的地址空间（代码和数据），使得它减少代价。
- So, by starting a thread an efficient path of execution is created while still sharing the original data area from the parent.
- 当与父进程共享原数据时，启动线程，创建有效路径的执行。

Figure 6.7
State associated with execution environments and threads

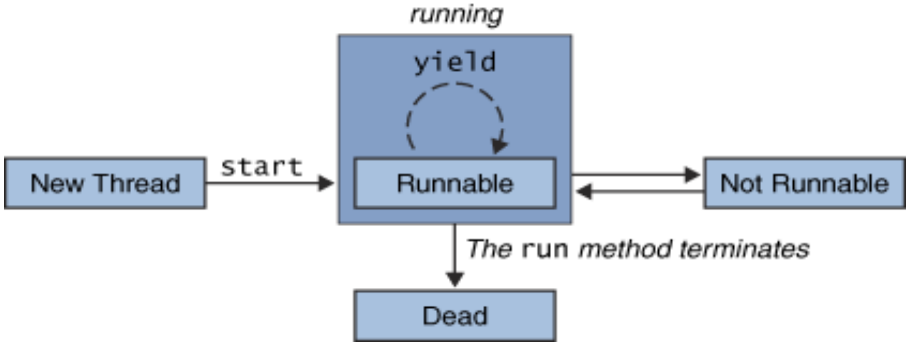
与执行环境和线程相关联的状态 p165

Execution environment	*Thread*
Address space tables	Saved processor registers
Communication interfaces, open files	Priority and execution state (such as *BLOCKED*)
Semaphores, other synchronization objects	Software interrupt handling information
List of thread identifiers	Execution environment identifier
Pages of address space resident in memory; hardware cache entries	

Thread states 线程状态

- Running State: A thread is said to be in the running state when it is being executed.
- 运行态：当它被执行时，处在运行状态。

- Ready State (Not Runnable): A thread in this state is ready for execution, but is not currently executed. Once a thread gets access to the CPU, it gets moved to the Running state.
- 就绪态：线程准备执行，但是当前不执行。一旦线程得到CPU的访问权，它就转成运行态。



Thread states (cont.)

- Dead State: A thread reaches the “dead” state when the run method has finished execution.
- “死”状态：当运行方式执行完成，线程到达“死”状态。

- Waiting State (yielding): In this state the thread is waiting for some action to happen. Once that action happens, the thread gets into the ready state. Threads in the waiting state could be sleeping, suspended, blocked, or waiting for a monitor.
- 等待态：在这个状态线程等待事件的发生。一旦时间发生，线程转成就绪态。等待态的线程可以睡眠、挂起、阻塞或监控等待。

Usages of Threads 线程的使用

- Threads are used for all sorts of applications, from general interactive drawing applications to games.
- 线程可以用于各种应用软件，从交互式图形应用到游戏。

  - For instance a program is not capable of drawing pictures while reading keystrokes. So the program either has to give full attention to listening to keystrokes or drawing pictures, otherwise one thread can listen to the keyboard while the other draws the pictures.
  - 例如，一个程序当读键时，不能画图。所以，程序或者读键或者画图，而一个线程可以读键另一个线程可以画图。

Usages of Threads (cont.)

- Another good usage of threads in on a system with multiple CPUs or cores.
- 线程可以很好的使用于多CPUs或多核的系统

  - In this case each thread would be run on a separate CPU, resulting is true parallelism instead of time sharing.
  - 在这种情况下，每个线程可以运行一个单独的CPU, 结果是真正的并行而代替时间共享。

Figure 6.8
Java thread constructor and management methods
Java线程构造器和管理方法

Thread(ThreadGroup group, Runnable target, String name)

Creates a new thread in the *SUSPENDED* state, which will belong to *group* and be identified as *name*; the thread will execute the *run()* method of *target*.

创建一个初始状态为挂起的新线程，它属于group 组，表示名为name，这一线程会执行target为run方式。

setPriority(int newPriority), getPriority()

Set and return the thread's priority.

设置和返回线程优先级

run()

A thread executes the *run()* method of its target object, if it has one, and otherwise its own *run()* method (*Thread* implements *Runnable*).

如果目标对象有run方法，线程执行其目标的run方法，否则，执行自己的*run()* 方法。

Java thread constructor and management methods
Java线程构造器和管理方法

- *start()*
- Change the state of the thread from *SUSPENDED* to *RUNNABLE*.
- 将线程的挂起状态转化为运行状态
- *sleep(int millisecs)*
- Cause the thread to enter the *SUSPENDED* state for the specified time.
- 将线程转换为挂起状态，并持续指定的时间
- *yield()*
- Enter the *READY* state and invoke the scheduler.
- 进入就绪状态并唤醒调度管理
- *destroy()*
- Destroy the thread.
- 结束线程

Figure 6.9
Java thread synchronization calls
Java线程同步调用

thread.join(int millisecs)

Blocks the calling thread for up to the specified time until *thread* has terminated.
在指定时间内阻塞调用线程，直到线程被终止。

thread.interrupt()

Interrupts *thread*: causes it to return from a blocking method call such as *sleep()*.
线程中断。使其从导致它阻塞的方法sleep返回。

object.wait(long millisecs, int nanosecs)

Blocks the calling thread until a call made to *notify()* or *notifyAll()* on *object* wakes the thread, or the thread is interrupted, or the specified time has elapsed.
阻塞调用线程直到notify() 或 notifyAll() 方式唤醒了线程，或者线程被中断，或超出了指定时间。

object.notify(), object.notifyAll()

Wakes, respectively, one or all of any threads that have called *wait()* on *object*.
分别唤醒一个或多个在object上的调用wait()方法的线程

Creating Threads in Java
用Java创建线程

- There are two ways in which Java allows for the creation of a thread.
- Java容许创建线程有两个方法：
  - Extending a class.
  - 扩展类
  - Implementing an interface.
  - 实现一个界面交互

Creating Threads (cont.)

- The first is to simply extend the Thread class and overload the run() method.
- 首先一个是简单扩展线程类和加载运行方式。

```
import java.lang.*;
public class Example extends Thread
{
    public void run()
    {
    ...
    }
}
```

Creating Threads (cont.)

- The second is to implement the Runnable interface. Using this we can create the same class.
- 第二是执行运行界面，使用这个方式我们创建同样的类

```
import java.lang.*;
public class Example implements Runnable
{
    public void run()
    {
    ...
    }
}
```

Synchronization 同步

- **There are two ways to synchronize threads both of which are based on the synchronized keyword, blocks and methods.**
- 有两种方法同步线程，基于同步键或阻塞的方法。
  - **A synchronized method is one which specifies synchronized as part of its declaration**
  - 同步方法的声明：

```
synchronized void f() { /* body */ }
```

Synchronization (cont.)

- Block synchronization takes an argument of which object to lock. This allows any method to lock any object.
- 阻塞同步采用的是对象加锁的参数方式。容许锁住对象。

```
synchronized(this) { /* body */ }
```

- One thing to note is that the synchronized keyword is not part of a method's signature, so it can't be inherited when subclassed.
- 值得注意的是：同步键不是签名方法的一种，所以它不能继承给子类。

Suspending Threads 挂起线程

- **Like the stop method, suspend() was also deprecated due to allowing deadlock to happen in some cases.**
- 象停止方式，挂起方式同样也是禁止各种死锁的发生
  - **Similarly to the stop() method we need to add something to the class and the run method to allow the thread to suspend itself when asked, and resume when requested.**
  - 与停止方式类似，我们需要添加类和运行方式，当需要时，容许线程观其自己，当被请求时，予以恢复。

Suspending Threads (cont.)挂起线程

- There are two ways in which to suspend a thread.
- 有两种方法挂起线程。
  - The thread calls wait() on itself.
  - The thread calls sleep().
  - 线程调用： wait() ， sleep().
- Then to resume the thread notify() or notifyAll() must be called.
- 然后，恢复线程调用notify() or notifyAll()

```
private volatile boolean susThread = false;
public void suspendThread(boolean val) {
   susThread = val;
   notify();
}
```

Suspending Threads

```
public void run() {
   Thread myThread == Thread.currentThread();
   while(myThread == exampleThread) {
       try {
         Thread.currentThread().sleep(1000);
         synchronized(this) {
             while(susThread)
                          wait();
         }
       } catch(InterruptedException e){}
   }
}
```

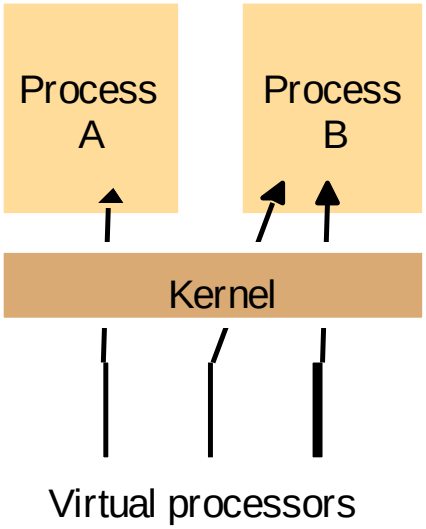
Suspending Threads (cont.)

- In the example it is important to realize that the check should be done after the sleep() in run() so that when the thread is resumed it doesn't immediately go back to the wait state.
- 在实例中，重要的是：当运行中的sleep()之后，应该设置检查，以便当线程被恢复时，它不能很快地恢复等待状态。

Example: FastThreads快速线程包

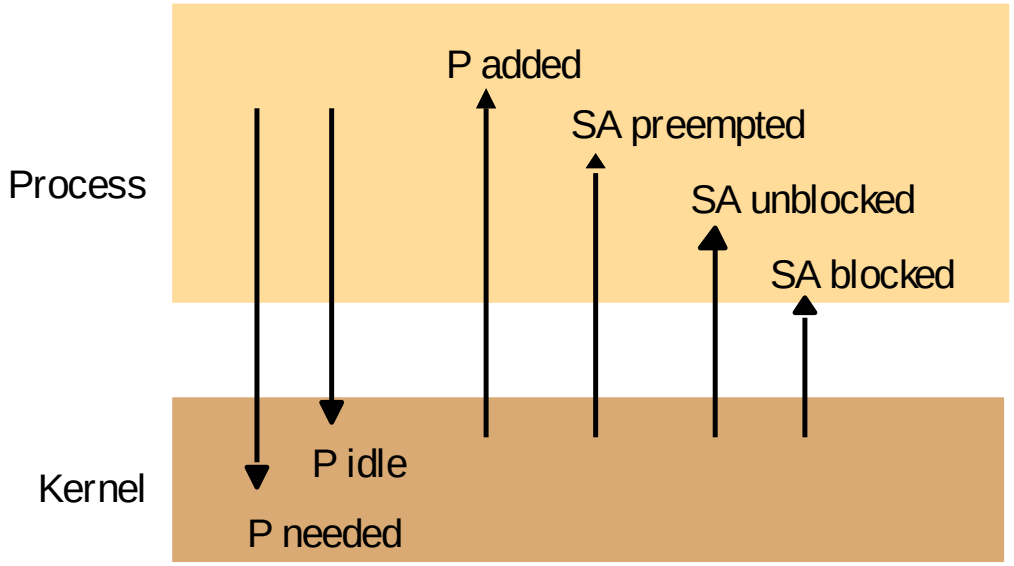
- FastThreads is a hierarchic, event-based thread scheduling system. It manages a kernel on a computer with one or more processors and a set of application processes. Each process has a user level scheduler, while the kernel allocates virtual processors to processes.
- 快速线程包是一个层次的，基于线程调度的系统。它用一个或多个处理器和一组应用进程管理计算机的内核。每一个进程由用户级别的调度器，同时，用户分配虚拟机给进程。
- Part a of the next slide shows a kernel allocation processes on a three-processor machine.
- 下面是包含三个处理机的内核分配进程。

Figure 6.10
Scheduler activations



A. Assignment of virtual processors to processes

将虚拟处理机分配给进程



B. Events between user-level scheduler & kernel
Key: P = processor; SA = scheduler activation

在用户级调度器和内核之间的事件

Kernel Notifications

- Figure 6.10b shows:
- that the process notifies the kernel when a virtual processor is idle and no longer needed or when an extra virtual processor is needed.
- 当虚拟机空闲并不在需要时，或者当进程请求获得额外的虚拟机时，进程将通知内核。
- that the kernel notifies a process when a scheduler activation notifies that process' scheduler of an event. There are four types of scheduler activation events, shown on the next slide.
- 当调度器激活进程事件调度活动，进程将通知内核

User-Level Scheduler notification

- A new Virtual Processor Allocated
- 虚拟处理器已分配。
- Scheduler Activation Blocked
- 调度器激活（SA）被阻塞
- Scheduler Activation Unblocked
- 调度器激活（SA）被解除阻塞
- Scheduler Activation Preempted
- 调度器激活（SA）被抢占

第5章 操作系统支持

- 5.1 简介
- 5.2 操作系统层
- 5.3 保护
- 5.4 线程
- 5.5 线程与进程
- 5.6 通信与调用
- 5.7 虚拟化
- 5.8 代码迁移
- 5.9 处理器任务分配

5.6 通信与调用 Invocation performance

- Invocation performance is a critical factor in distributed system design. The more designers separate functionality between address spaces, the more remote invocations are required.
- 在分布式系统的设计中，调用性能是一个非常关键的因素。如果在地址空间之间分离的功能余额越多，就必须越多的使用远程调用。

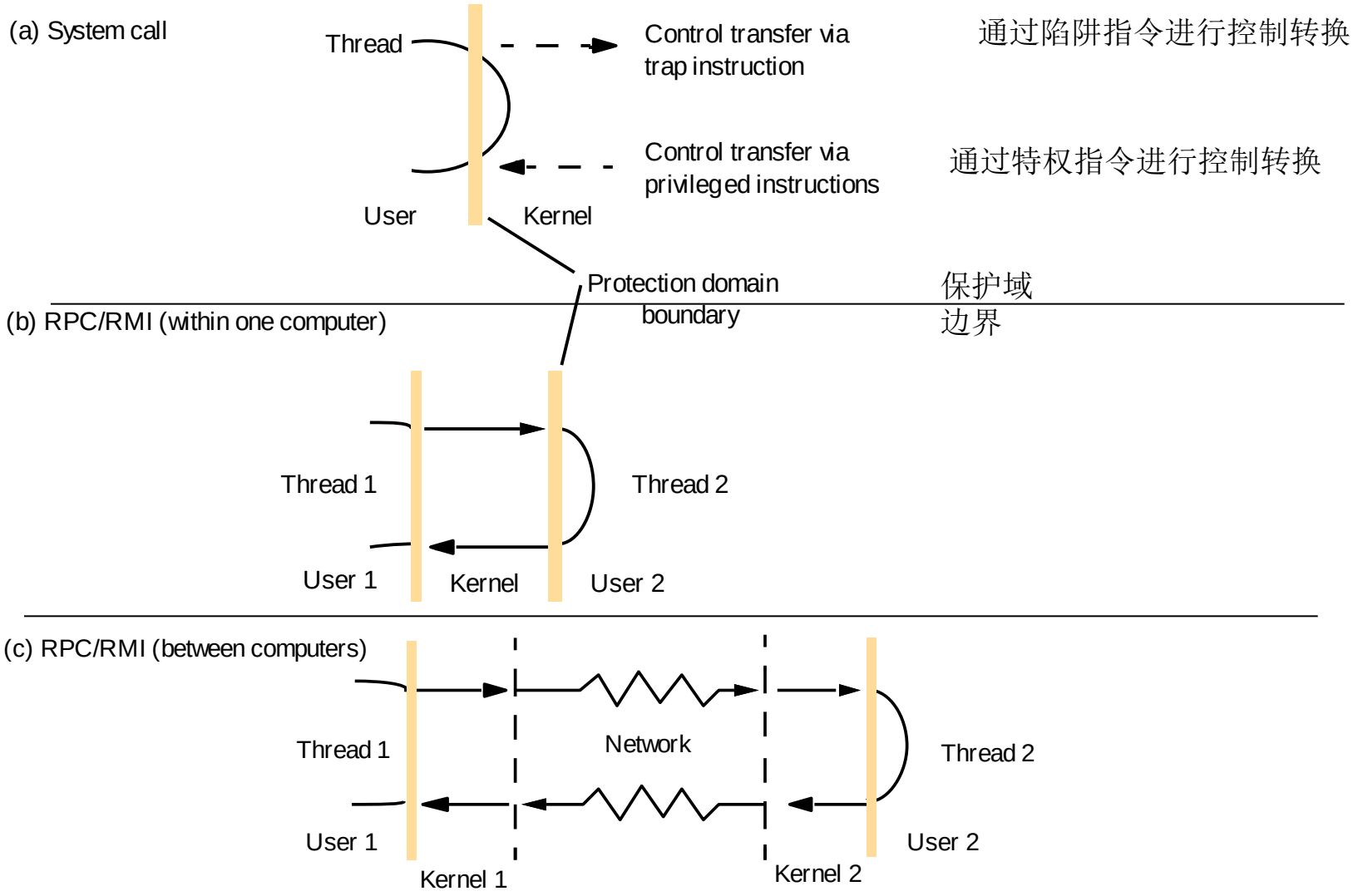
Invocation

- The invocation of a local process takes place within local memory and may involve a few tens of instruction cycles.
- 本地进程调用发生在本地内存，会涉及几十个指令周期。
- Invocation of a remote process involves network activities and possibly access to files, and may require billions of instruction cycles in processors running at speeds measured in gigahertz.
  远程调用涉及网络活动和可能的文件访问，在进程运行速度达到千兆赫时，需要几十亿条指令周期
- These invocation activities are external to the desired processing activities and increase costs without adding value.
- 这些调用活动是进程活动的外部表现，不增加值时，代价增加。

Latency 等待时间

- Null Invocation costs are the delays required to set up communications and the non-goal performing overhead of invocations. These fixed overhead costs measure the latency of the connection.
- 空调用的代价是设置通信延时或无目标的执行调用。这些固定的代价是连接反应时间的测量。
- Substantial efforts go into minimizing and reducing latency costs in distributed applications. Slide 6.11 shows some of the delays for different types of invocations.
- 在分布式应用中，努力反应时间最小化或减少反应时间。Slide 6.11 显示不同类型调用的延时。

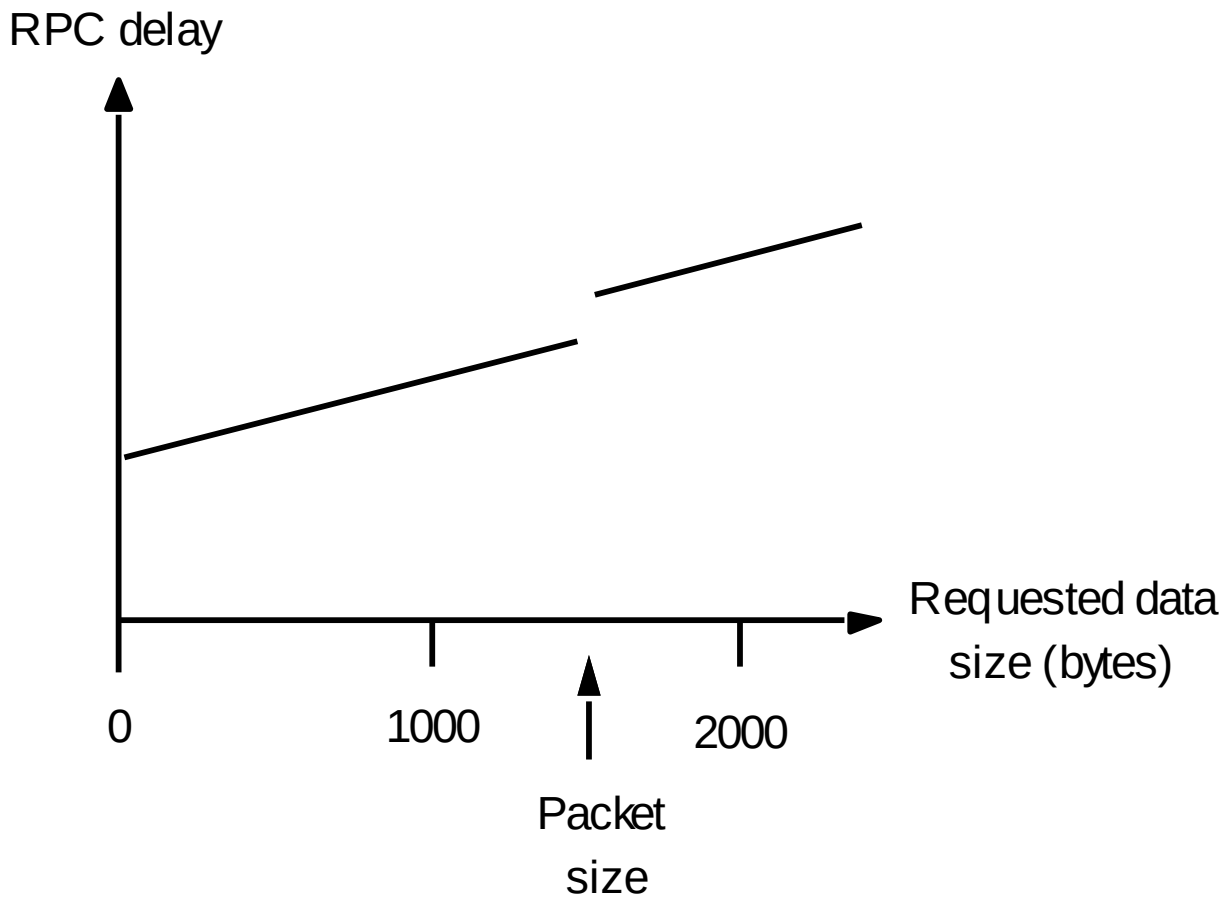
Figure 6.11
Invocations between address spaces
地址空间之间的调用



Invocation Costs as a percentage of throughput
吞吐量的百分比作为调用代价

- As more goal work is accomplished for a fixed amount of overhead, the overhead is less of a concern as a percentage of total costs.
- 当多目标工作固定开销下被完成时，开销很少关心总代价的百分比.
- With very small data sizes, most of the system time may be spent in overhead activities.
- 数据量小，开销大多是系统时间。
- With large data transfers, the overhead costs may be negligible as a percentage of total costs.
- 大数据量传输，作为总代价的百分比的开销是不容忽视的.
- Slide 6.12 illustrates this as a graph of RPC delays against packet size in RPC transfers.
- Slide 6.12 RPC（远程过程调用）包传输延时示意图

Figure 6.12
RPC delay against parameter size



RPC延迟与参数数据量大小的关系

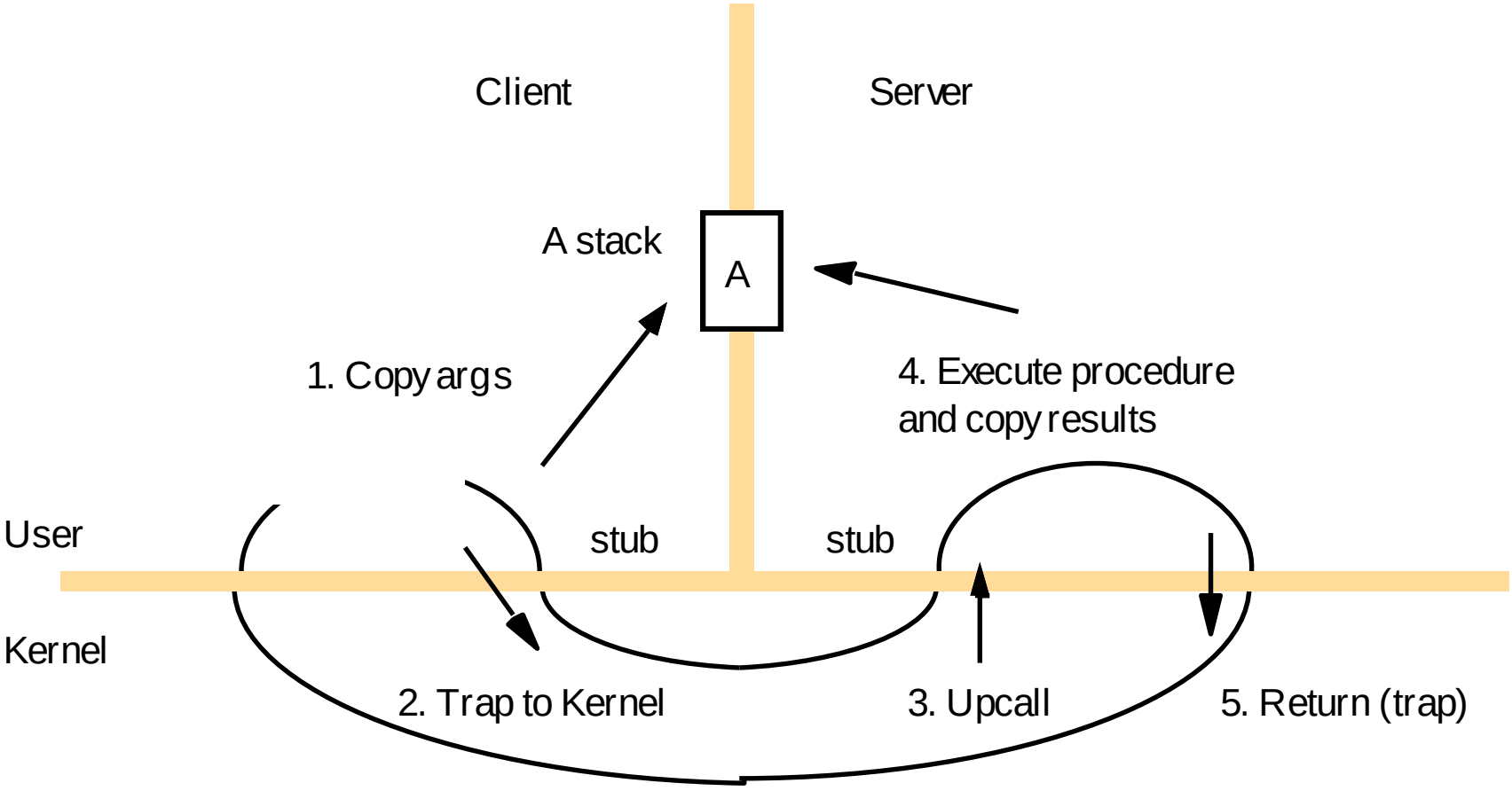
RPC delay against parameter size
RPC延迟与参数数据量大小的关系

- The delay is roughly proportional to the size until the size reaches a threshold at about network packet size. Beyond that threshold, at least one extra packet has to be sent, to carry the extra data.
- 在数据的大小达到一个同网络数据包大小相近的阈值之前，延迟和数量的大小基本上成正比。超过这一阈值之后，为了传输额外的数据，系统至少要多传一个额外的包。
- Depending on the protocol, a further packet might be used to acknowledge this extra packet. Jump in the graph occur each time the number of packet increases
- 根据协议，为了确认这个额外的包，还需要传一个数据包。每次传输包的数量增加时，在图上便会出现一个跳跃。

Lightweight RPC

- One way to reduce the overhead of a remote invocation is to share some of the costs. While process invocation is expensive, some activities can be done once and then reused, while others must be done for each communication.
- 减少远程调用开销的一种方法是代价的共享。当进程调用昂贵时，其他进程进行每次通信，一些活动进行一次或重用。
- Lightweight RPC attempts to minimize overhead by sharing process activities within parent processes that are pre-established. Overhead costs can be reduced by as much as two-thirds.
- 轻型RPC试图通过共享父进程先前订制的活动最小化开销，开销代价可以减少2/3。

Figure 6.13
A lightweight remote procedure call

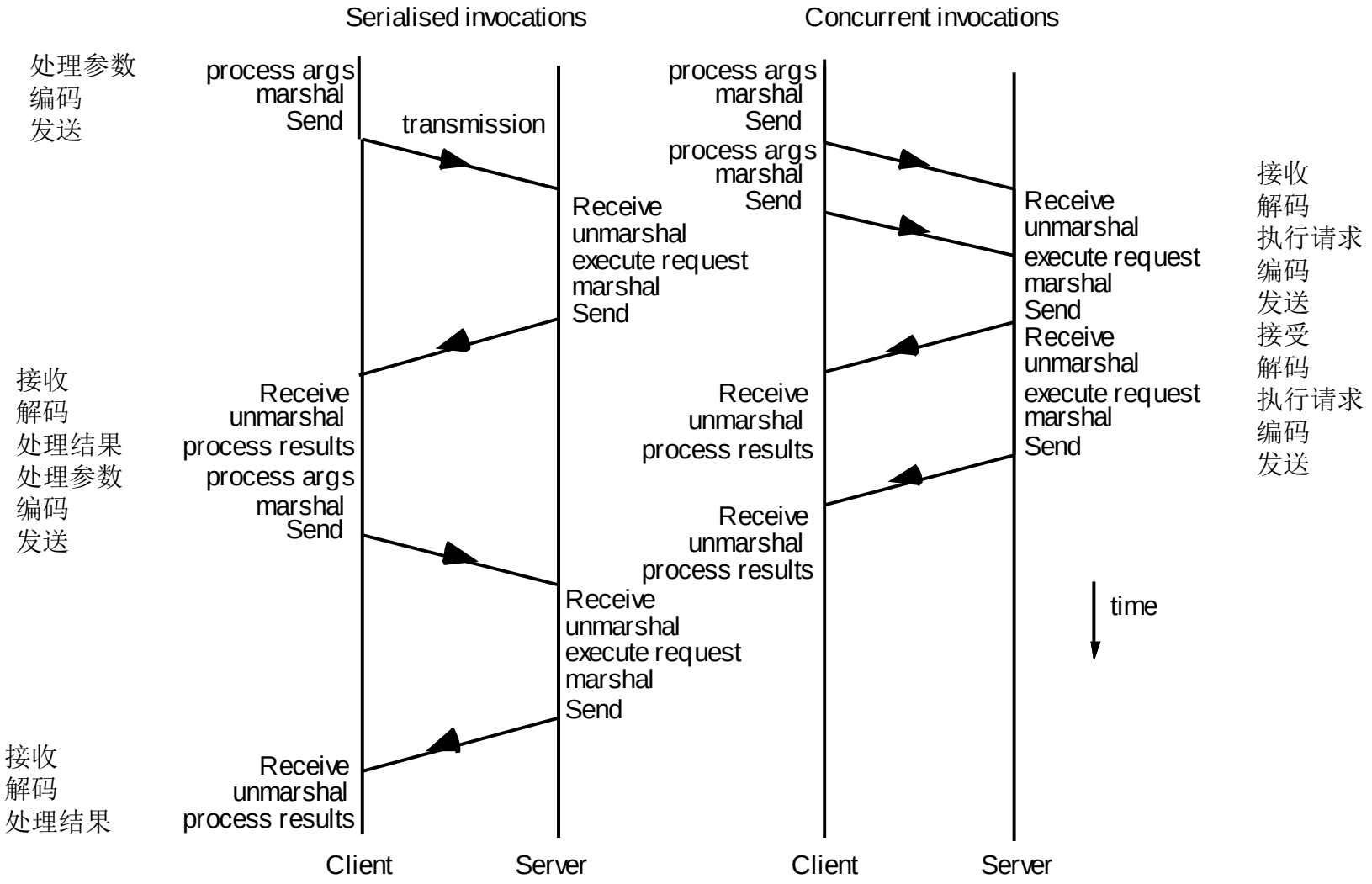


轻型远程调用进程：共享区域A栈

Concurrent Invocation 并发调用

- Another approach to reducing communication overhead is to reduce the number of messages sent to establish the connection and to continue processing while communicating instead of suspending threads until a message is received.
- 另一种减少通信代价的方式是减少建立连接或继续处理时信息发送的数量，当通信代替线程挂起直道消息被接收。
- Serialized and concurrent invocations are compared on the next slide.
- 下面是串行与并行调用的比较。

Figure 6.14
Times for serialized and concurrent invocations
串行调用和并发调用的时序



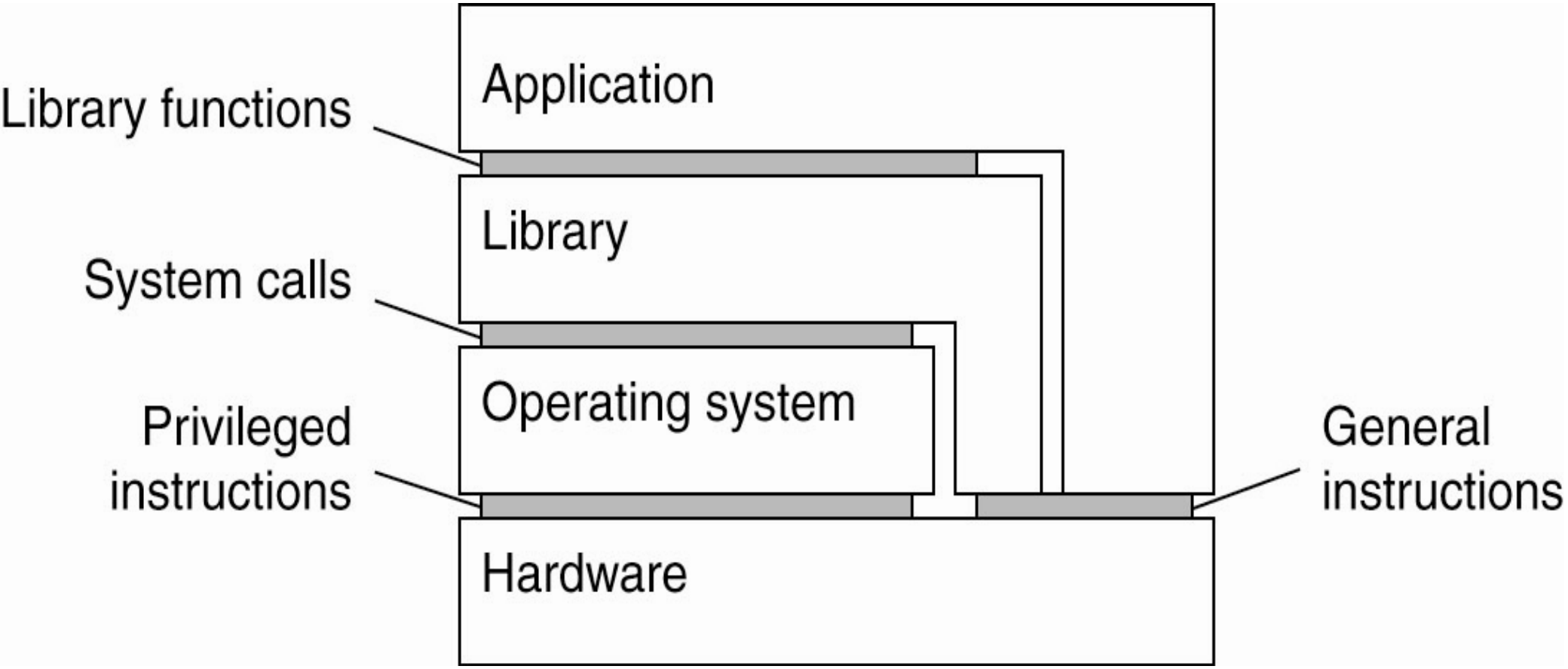
第5章 操作系统支持

- 5.1 简介
- 5.2 操作系统层
- 5.3 保护
- 5.4 线程
- 5.5 线程与进程
- 5.6 通信与调用
- 5.7 虚拟化
- 5.8 代码迁移
- 5.9 处理器任务分配

5.7 虚拟化

- 虚拟化：扩展或替换一个已有界面来模仿另一个系统的行为。
- 在分布式系统中的作用：
  - 尽管硬件和底层系统软件变化较快，高层软件（中间件和应用程序）要稳定得多
  - 通过让应用程序运行在自己的虚拟机上，从而减少平台和机器的种类，提供高度的移植性和灵活性

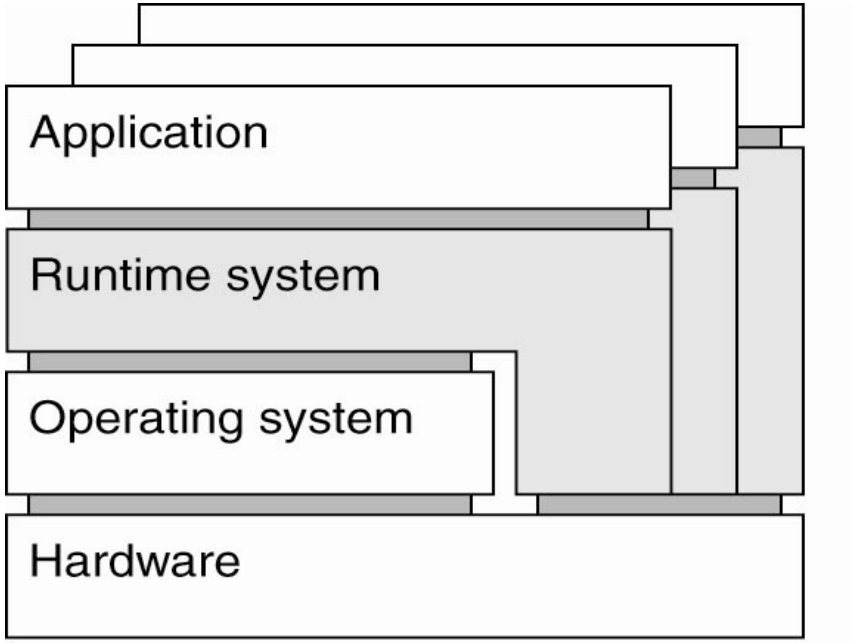
虚拟机体系结构



计算机系统多种界面

虚拟化的方式（一）

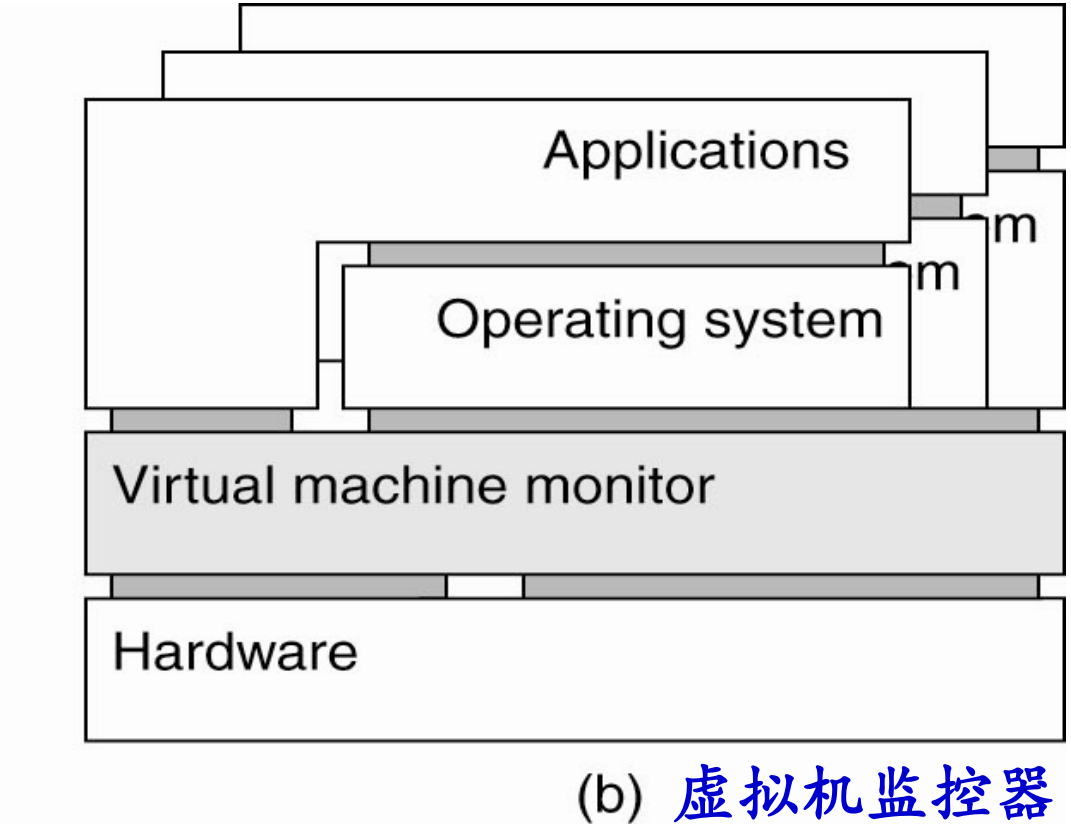
进程虚拟机，混合多个应用程序和运行时系统



(a) 构建一个运行时系统

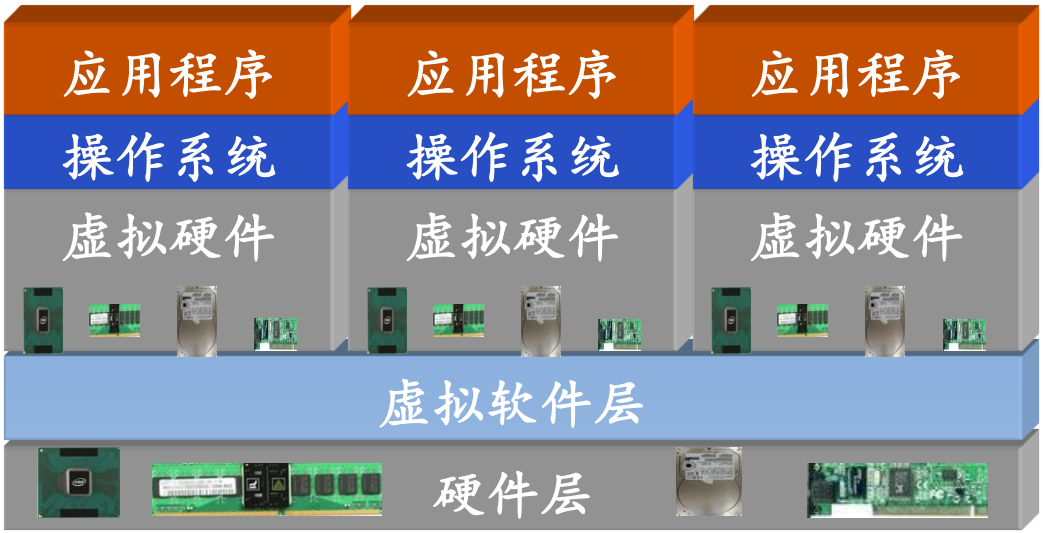
虚拟化的方式（二）

虚拟机监视器，混合多个应用程序和操作系统



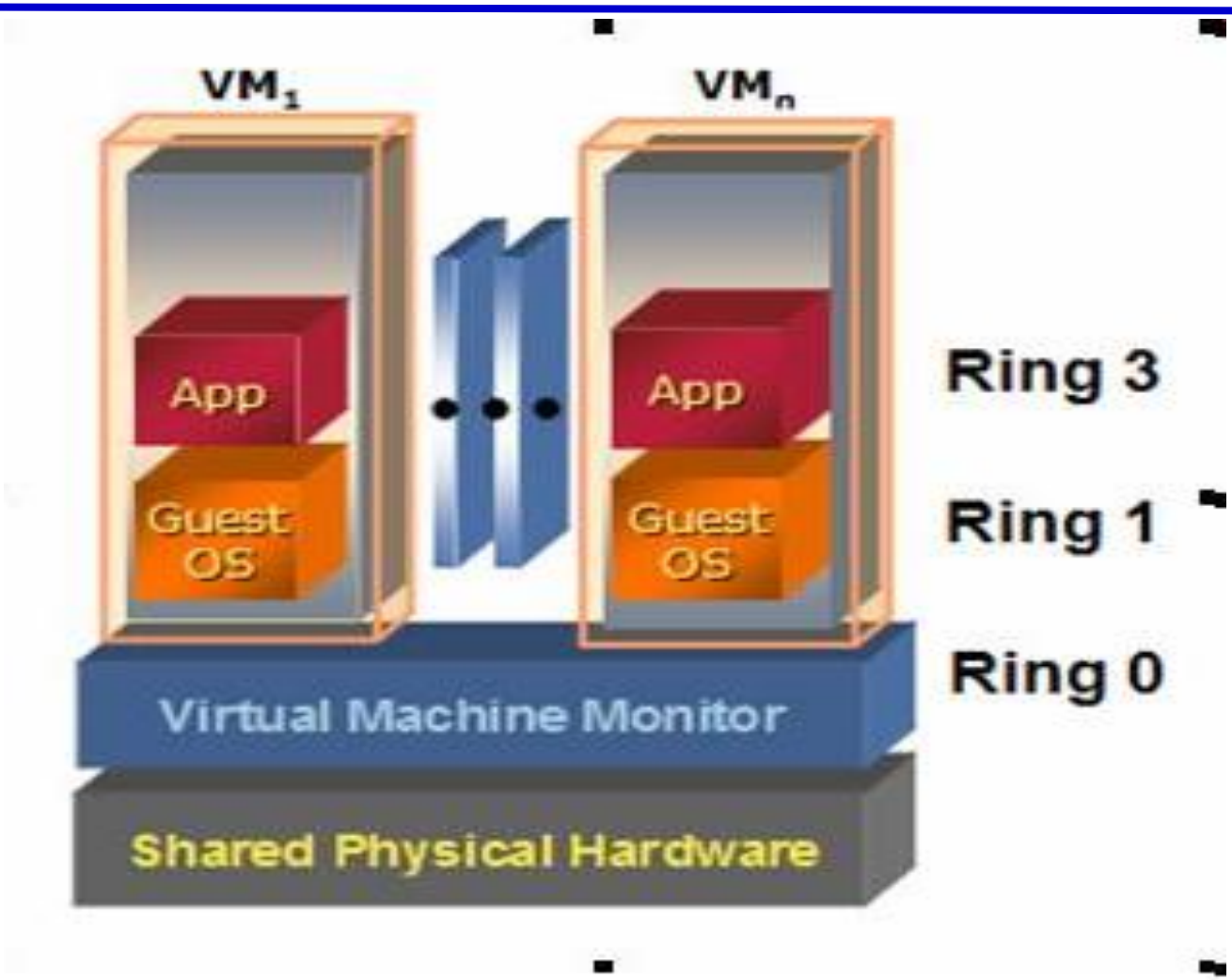
(b) 虚拟机监控器

虚拟化技术简介

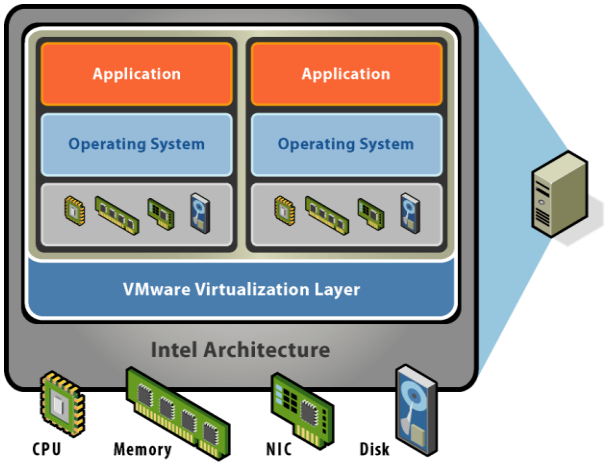


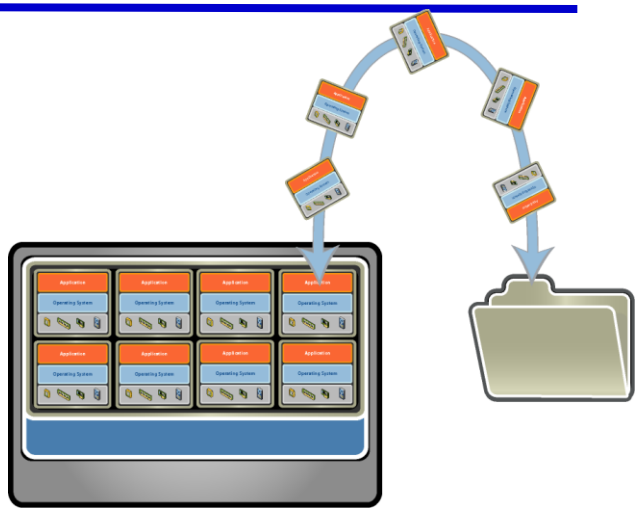
将一台服务器当做N台服务器来使用

虚拟机管理器


VM1
VMn
App
Guest OS
App
Guest OS
Ring 3
Ring 1
Ring 0
Virtual Machine Monitor
Shared Physical Hardware

虚拟化的关键特征





分区

- 在一个物理机上运行多个OS
- 更充分利用服务器资源
- 支持高可用——分区之间可以组建集群（负载均衡、双机容错）

隔离

- 从硬件层面隔离系统故障和安全威胁
- 在虚拟机之间动态的分配CPU、内存等系统资源
- 保证服务可用

封装

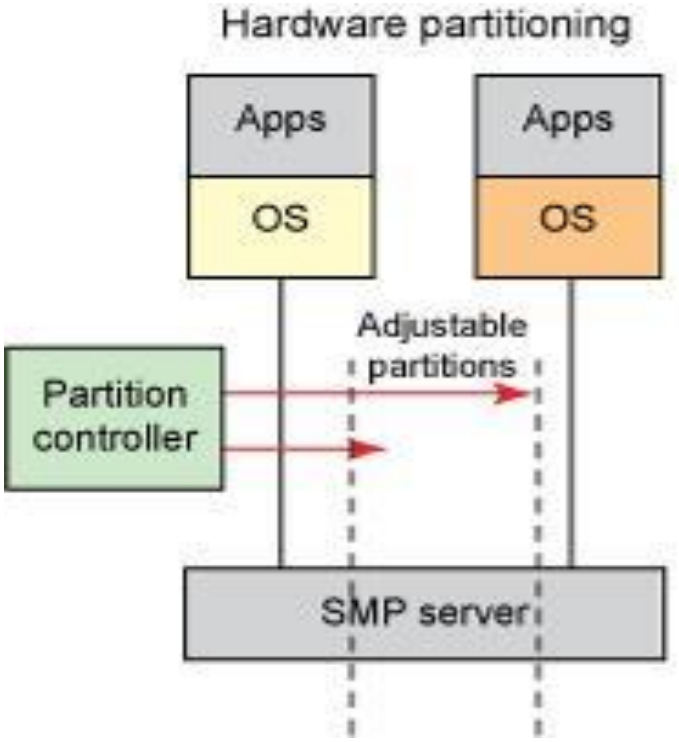
- 将虚拟机封装成与硬件配置无关的文件
- 随时对虚拟机进行快照
- 通过简单的文件拷贝对虚拟机进行迁徙

虚拟化的优势

- 虚拟化前
  - 每台主机一个操作系统
  - 软件硬件紧密地结合
  - 在同一主机上运行多个应用程序通常会遭遇沖突
  - 系统的资源利用率低
  - 硬件成本高昂而且不够灵活
- 虚拟化后
  - 打破了操作系统和硬件的互相倚赖
  - 通过封装到到虚拟机的技术，管理操作系统和应用程序为单一的个体
  - 強大的安全和故障隔离
  - 虚拟机是独立于硬件的，它们能在任何硬件上运行

硬件分区技术

- 特征
  - 将硬件资源被划分成数个分区，每个分区享有独立的CPU、内存，并安装独立的操作系统。
- 缺点
  - 缺乏很好的灵活性，不能对资源做出有效调配。



第5章 操作系统支持

- 5.1 简介
- 5.2 操作系统层
- 5.3 保护
- 5.4 线程
- 5.5 线程与进程
- 5.6 通信与调用
- 5.7 虚拟化
- 5.8 代码迁移
- 5.9 处理器任务分配

5.8 代码迁移

代码迁移

定义：将程序（或执行中的程序）传递到其它计算机。

- 进程迁移：就是将一个进程的状态，从一台机器(源机)转移到另一台机器(目标机)上，从而使该进程能在目标机上执行。
- 这个概念主要来自于对大量互连系统间负载平衡方法的研究，但其应用已超出了这个领域。

代码迁移

迁移动机：

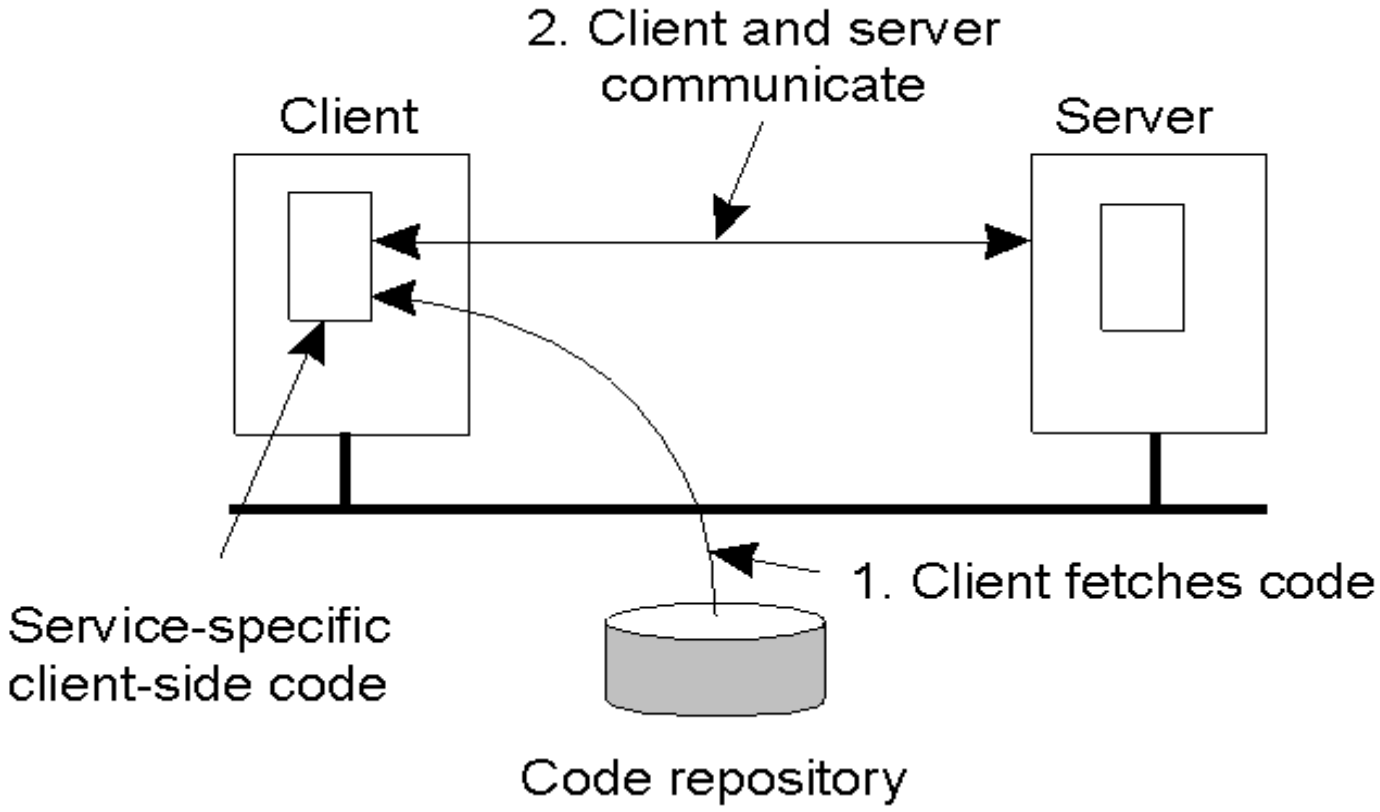
- 实现负载均衡
  - 将进程从负载重的系统迁移到负载轻的系统，从而改善整体性能
- 改善通信性能
  - 交互密集的进程可迁移到同一个节点执行以减少通信开销
  - 当进程要处理的数据量较大时，最好将进程迁移到数据所在的节点

代码迁移

- 可用性
  - 需长期运行的进程可能因为当前运行机器要关闭而需要迁移
- 使用特殊功能
  - 可以充分利用特定节点上独有的硬件或软件功能

代码迁移-灵活性

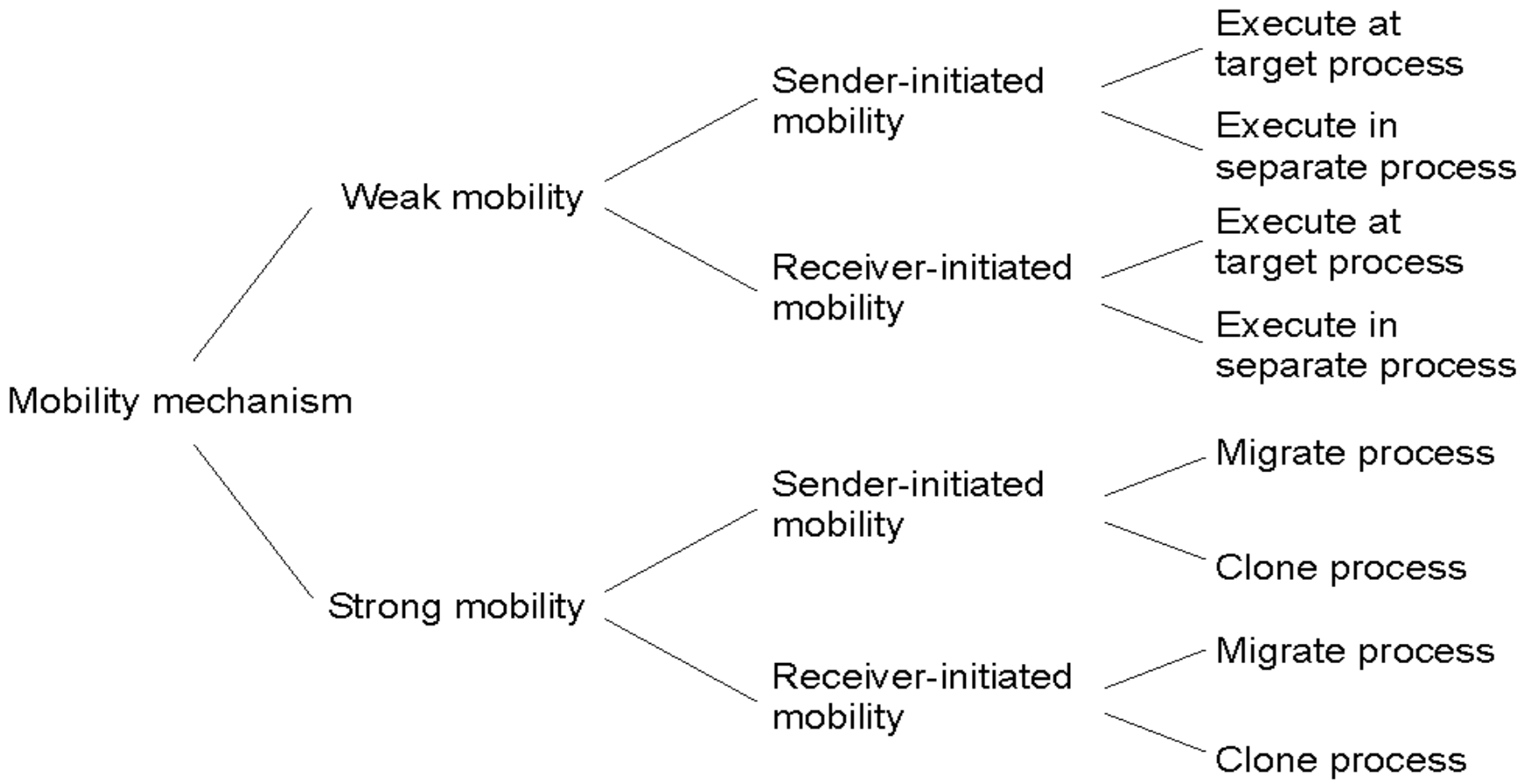
- 客户首先获取必需的软件，然后调用服务器



代码迁移模型

- 代码迁移的不同方法
- 进程组成：
  - 代码段：正在运行的程序的所有指令
  - 资源段：包含进程需要的外部资源的指针
  - 执行段：存储进程的当前执行状态量：私有数据、堆栈和程序计数器
- 弱可移动性与强可移动性
  - 弱可移动性：只能传输代码段以及某些初始化数据，程序以初始状态重新执行，简单
  - 强可移动性：还可以传输执行段，较难实现
- 发送者启动与接收者启动
  - 发送者启动：计算服务器，需要访问服务器资源，客户端需要注册验证
  - 接收者启动：可以是匿名的，Java applet，提高客户端性能

代码迁移模型


Mobility mechanism
Weak mobility
Strong mobility
Sender-initiated mobility
Receiver-initiated mobility
Sender-initiated mobility
Receiver-initiated mobility
Execute at target process
Execute in separate process
Execute at target process
Execute in separate process
Migrate process
Clone process
Migrate process
Clone process

迁移与本地资源

- 进程对资源绑定：按标志符（URL）、按值和按类型
- 资源对机器绑定：未连接（数据文件）、附着连接（数据库）和紧固连接（本地设备）

资源对机器绑定

进程对资源绑定	未连接	附着连接	紧固连接
按标志符	MV (or GR)	GR (or MV)	GR
按值	CP (or MV, GR)	GR (or CP)	GR
按类型	RB (or MV, CP)	RB (or GR, CP)	RB (or GR)

- **MV**：移动资源
- **GR**：建立全局系统范围内引用
- **CP**：复制资源的值
- **RB**：将进程重新绑定到本地同类型资源

- 迁移代码时，根据引用本地资源方式不同所采取的不同做法

进程迁移的动机

进程迁移在分布式操作系统中很重要，主要有以下几个原因：

- ① 负载共享。通过将进程从负载较重的节点迁移到负载较轻的节点，使系统负载达到平衡，从而提高整体执行效率。
- ② 减少通信开销。可以将相互间紧密作用的进程迁移到同一节点，以减少它们相互作用期间的通信耗费。
- ③ 可获得性。运行时间较长的进程在出现错误时可能需要迁移。那么，一个想继续的进程既可以迁移到另外的系统，也可以推迟运行，待错误恢复后在当前系统中重新开始。
- ④ 利用特定资源。

进程迁移机制

在设计一个进程迁移机制时要考虑许多问题，其中包括：

- ① 由谁来激发迁移？
- ② 进程的哪一部分被迁移？如何进行迁移？
- ③ 如何处理未完成的消息和信号？

进程迁移机制

1. 迁移激发

由谁激发迁移取决于迁移机制的目的。

- 若其目的在于负载平衡，那么，通常由操作系统中掌管系统负载的组件决定什么时候进行迁移；
- 若其目的在于获得特定资源，那么，可由需要资源的进程自行决定何时进行迁移，这种迁移也称为自迁移（Self-migration）。

对前一种情况，整个迁移作用以及多系统的存在，对进程都可以是透明的；对后一种情况，进程必须了解分布式系统的分布情况。

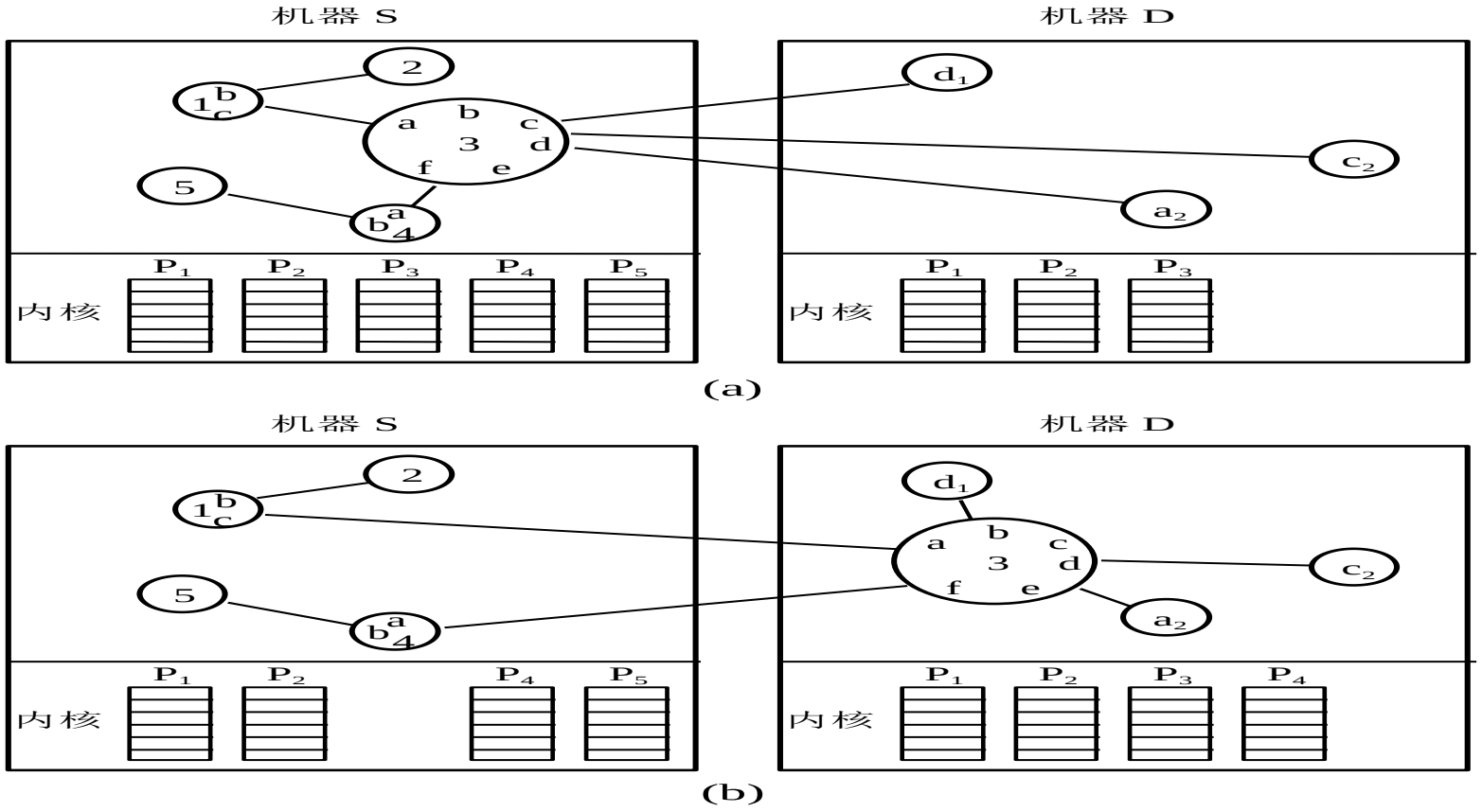
进程迁移机制

2. 迁移什么

- 当迁移一个进程时，必须在源系统上破坏该进程，并在目标系统上建立它。这才是进程的移动，而不是复制进程。
- 因此，进程映像，至少包括进程控制块，必须移动。另外，这个进程与其他进程间的任何链接也必须更新。

进程迁移机制

进程迁移举例：



进程迁移机制

从执行的角度看,进程控制块移动的困难之处在于进程地址空间的传送和进程打开文件的传送。考虑第一个问题——进程地址空间的传送，假设用虚拟存储策略(分段或分页)，有两种方法：

- ① 迁移时传送整个地址空间。
- ② 仅传送那些在主存中的地址空间部分，在需要时再传送虚拟地址空间中的段。

进程迁移机制

3. 消息和信号处理

对于那些未完成的消息和信号，可通过一种机制进行处理：在迁移进行时，暂时存储那些完成的消息和信号，然后将它们直接送到新的目的地，有必要在迁移出发的位置将正在发出的信息维持一段时间，以确保所有的未完成消息和信号都被传送到目的地。

第5章 操作系统支持

- 5.1 简介
- 5.2 操作系统层
- 5.3 保护
- 5.4 线程
- 5.5 线程与进程
- 5.6 通信与调用
- 5.7 虚拟化
- 5.8 代码迁移
- 5.9 处理器任务分配

5.9 处理器任务分配

- 分配算法的设计原则
- 分配算法的实现问题
- 分配算法实例

分配算法的设计原则

分配算法的设计原则：

- 局部性算法和全局性算法
- 发送者启动算法和接收者启动算法
- 确定性算法和启发性算法
- 集中式算法和分布式算法
- 最优化算法和次优化算法

局部性算法和全局性算法

设计问题与迁移策略有关：

- 当一个新进程被创建时，系统需要决定它是否在创建它的机器上运行。若该机器繁忙，那这个新进程就必须迁移到其它机器上去运行。
- 对于是根据本机局部信息还是全局信息来决定新进程是否迁移？
- 比较：
  局部算法简单，但远远达不到最优；
  而全局算法需要付出巨大的代价来换取一个性能稍微好一点的结果。

局部性算法和全局性算法

对于是根据本机局部信息还是全局信息来决定新进程是否迁移？

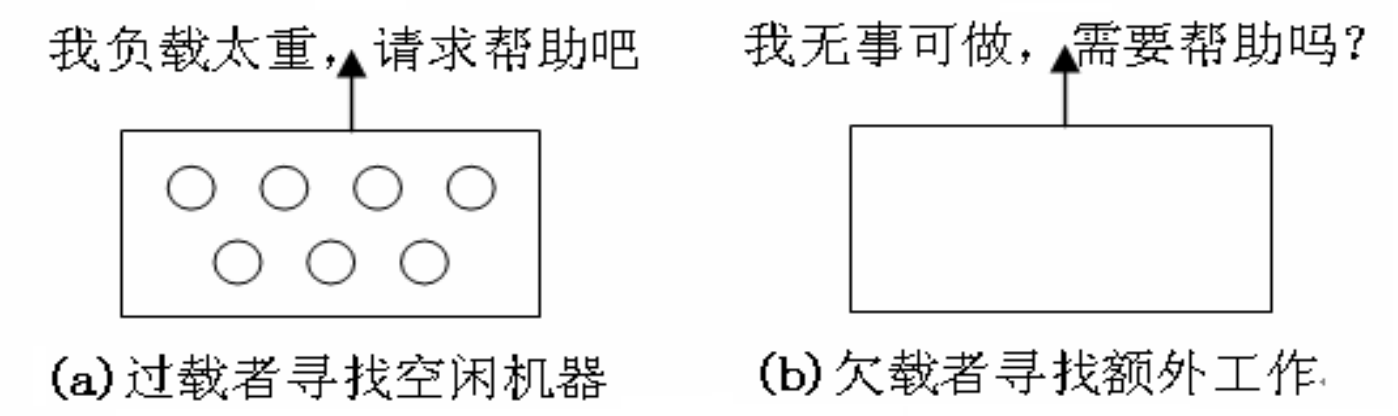
目前存在着两种观点：

- 一种观点主张简单的局部算法：若机器的负载低于某个阀值，那么新进程就在本地机器上运行；否则，就不允许该进程在本地上运行。
- 另一种观点认为局部算法太武断了。最好在决定新进程是否在本地机器上执行之前，先收集其它一些机器上的负载信息。

发送者启动算法和接收者启动算法

- 设计问题与定位策略有关。
  - 一旦决定不允许一个进程在本地机器上运行，那么，迁移算法就必须决定将该进程应该迁移到哪台目的机器上。
  - 显然，迁移算法不能是本地的。它需要通过获得其它机器上的负载信息来决定迁移的目的机器。这些负载信息可以通过两种途径来获得。
    - 一种是过载者启动的
    - 另一种是欠载者启动的

- 过载者启动：由过载者来寻找迁移的目的机器
  - 如图：一个机器超载时，它向其它机器发送求助请求，希望将自己的一些新进程迁移到其它机器上运行。
- 欠载者启动：
  - 当一个机器处于空闲状态即欠载状态时，由这台欠载机器来宣布自己可以接收外来的工作。其目的就是寻找一台可以给自己增加一些额外工作的机器



分配算法的实现问题

- 负载的度量
- 额外消耗
- 复杂性
- 稳定性

实现问题1：负载的度量

- 基本上，所有的算法都假定每一台机器都知道它自己的负载，也就是说，它可以判断自己是超载还是欠载，并且能够告诉其它机器自己的负载。
- 然而，度量一台机器是否超载并不象它看上去那样简单。

负载的度量：方法1

- 度量方法1：以机器上的进程数量作为机器的负载
  - 优点：简单
    原因：只需要计算机器上的进程数量
  - 缺点：用进程数量的多少来表示机器的负载是不确切的。
    原因：即使在一台空闲机器上，仍然会有一些后台监视进程在运行，例如，邮件、新闻、守护进程、窗口管理程序以及其它一些进程。

负载的度量：方法2

- 对度量方法1进行如下改进：
  只计算正在运行或已经就绪进程的数量。
  原因：
  - 每一个正在运行或处于就绪状态的进程都会给系统增加一定的负载，即便它是一个后台进程。

  存在的问题：
  - 许多后台守护进程只是定时被唤醒，检查所感兴趣的事件是否发生，如果没有，则重新进入睡眠状态。因此，这类进程只给系统带来很小的负载。

负载的度量：方法3

- 直接使用处理机利用率：
  就是处理机繁忙时间在全部时间中（繁忙时间+空闲时间）所占的比例。

$$\text{处理机利用率} = \frac{\text{处理机繁忙时间}}{\text{处理机繁忙时间} + \text{处理机空闲时间}}$$

  - 一个利用率为20%的处理机负载要比利用率为10%的处理机大
  - 优点：比较合理
    原因：兼顾了用户进程和守护进程

实现问题2：额外开销

- 许多理论上的处理机分配算法都忽略了收集负载信息以及传送进程的额外开销。
- 若一个算法将一个新创建的进程传送到远程机器上运行仅使系统性能提高10%左右，那它最好不要这样做，原因是传送进程的开销足以抵消所提高的性能。
- 一个好的算法应该考虑算法本身所消耗的处理机时间、内存使用、以及网络带宽等。但很少有算法能做到这一点，因为它太难了。

实现问题3：复杂性

- 在处理机分配算法实现中还必须考虑复杂性。
- 事实上，所有的研究者只分析、模拟或计算处理机的利用率、网络的使用情况以及响应时间，以此来衡量他们所提出算法的好坏。
- 很少有人考虑软件的复杂性对系统的性能、正确性和健壮性所产生的影响。算法性能有可能只是比现有的算法稍好一点，但在实现上却复杂得多。

处理机分配算法的实现问题

- Eager 等人在1986年所做的研究使追求复杂和最优算法的人们看到了希望。
- 他们研究了三个算法，在这三个算法中，所有的机器都测量自己的负载以判断它是否超载。
- 当一个新进程创建时，创建该进程的机器就会检查自己是否超载，如果是，则它就寻找一台欠载的远程机器去运行该进程。这三个算法的不同之处在于寻找远程机器的方法。

处理机分配算法的实现问题

算法1

- 随机地选择一台机器，并把新创建的进程传送到该机器上。
- 如果该接收机器本身也超载，它也同样随机地选择一台机器并把该进程传送过去。
- 这个过程一直持续到有一台欠载的机器接收它为止，或者指定计数器溢出停止该进程的传送

处理机分配算法的实现问题

算法2

- 随机地选择一台机器，然后发送一个信息给该机器询问该机器是超载还是欠载。
- 如果该机器欠载，它就接收新创建的进程；否则，新进程的创建机器继续随机地选择一台机器并向其发送一个询问消息。
- 这个过程一直持续到找到一台欠载机器为止，或超过了一定的询问次数，如果找不到欠载机器，该新创建的进程就只好留在本地机器上运行。

处理机分配算法的实现问题

算法3

- 给k台机器发送询问消息，接收这k台回送的负载消息。这个新进程将发送给负载最小的机器，并在它上面运行。
- 显然，如果我们不考虑所有发送询问负载消息和传送进程的额外开销，那么，人们会认为算法3的性能最好，事实上也确实如此。
- 尽管算法3的性能只比算法2的性能稍好一点，但其复杂性以及额外开销却比算法2要大的多。
- Eager等人认为，如果一个简单算法只比复杂算法在性能上略低一点的话，那么，最好使用简单算法。

实现问题4：稳定性

- 最后一个实现问题就是稳定性。由于不同的机器都在异步地运行各自的计算，所以，整个系统的负载很少能够达到平衡。
- 因此，有时候会发生这样一种情况：在某个时刻，机器A得到的信息是机器B的负载较轻，因而，它就将新创建的进程传送给机器B。机器B在收到该进程之前负载又增加了，所以，收到该进程后，它发现机器A的负载较轻，于是，它就将该进程又传送给机器A。这样造成了某个可怜的进程被来回传送的情况。
- 原因是:每一个机器上的负载每时每刻都在变化。

分配算法实例

- 1. 图论确定算法
- 2. 集中式算法
- 3. 层次性算法
- 4. 超载者启动的分布式启发算法
- 5. 欠载者启动的分布式启发算法
- 6. 拍卖算法

1. 图论确定算法

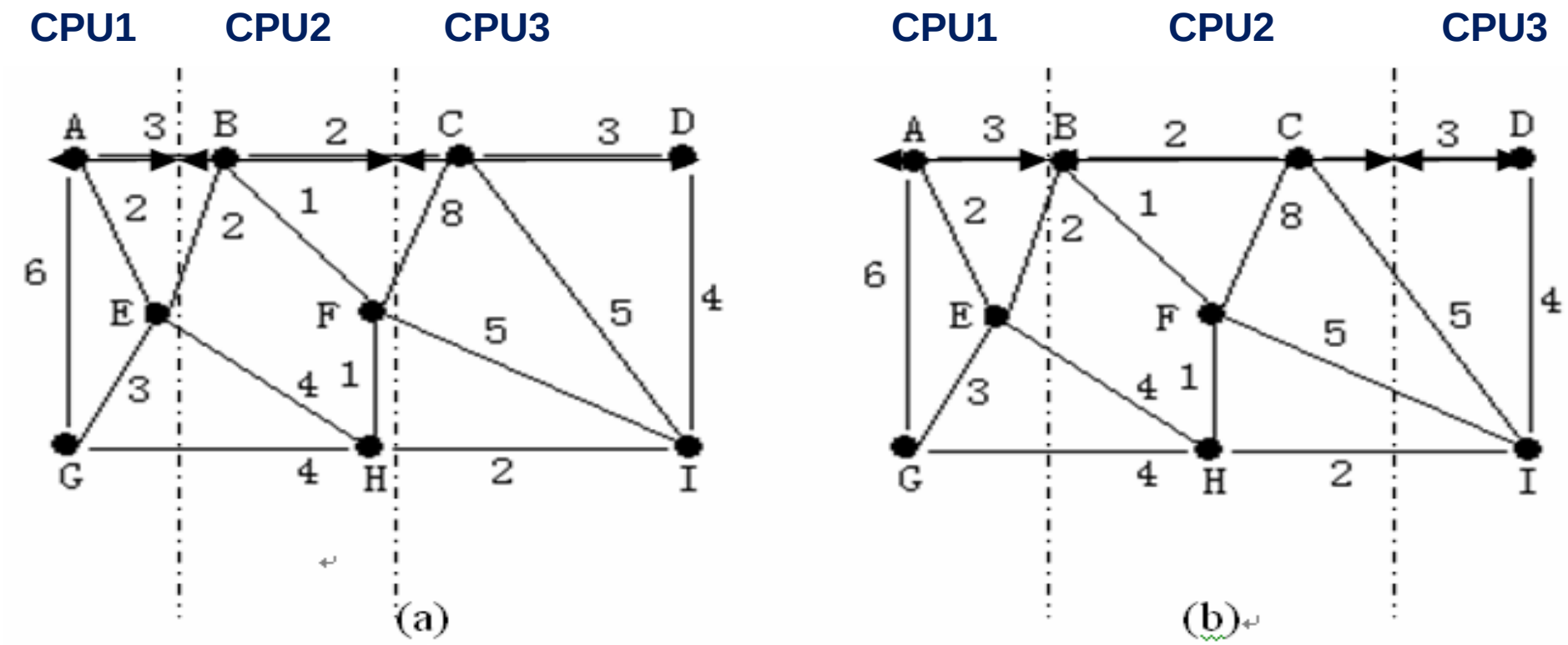
- 假定：每个进程都知道
  1）所需的处理机
  2）所要求的内存
  3）系统中任意一对进程间的平均通信量
- 若系统中处理机的数目k比进程数少，那么系统中的一些处理机就必须被分配多个进程
- 基于图论的确定性算法
  保证在系统网络通信量最小的条件下对处理机进行分配。

系统的带权图表示

- 系统可以被表示图G(V,E)，
  V中的每个节点表示一个进程
  E中的每条边表示两个进程需要通信，边上面的数字表示两个进程之间的通信量。
- 从数学角度看，处理机分配问题已经被简化为：
  在一定的约束条件下（例如，每一个子图总的处理机和内存需求量不超过某一个阀值）将图分割成k个不相连的子图。
- 算法的目标就是在满足所有限制条件下，找到一个分割方法，使得分割后各子图之间的通信量之和最小。

分割举例

- 下图表示了一个图的两种不同的分割方法，并得到了两个不同的通信量。



给 3 个处理机分配 9 个进程的方法

分割举例

a中，系统图被分割为：

A, E, G在处理机1上

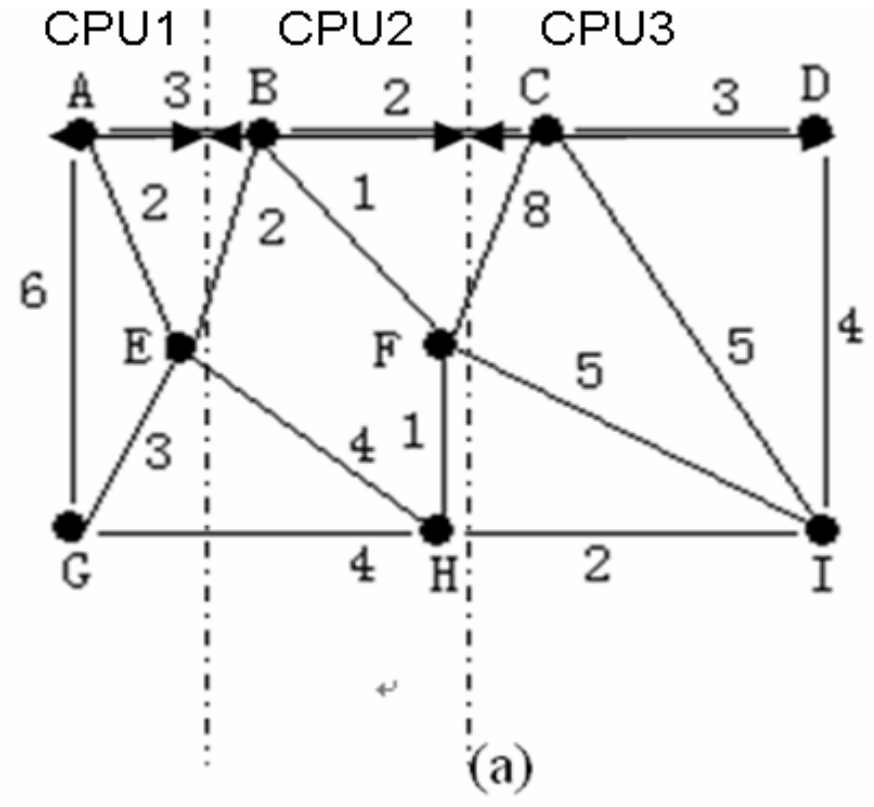
B, F, H在处理机2上

C, D, I在处理机3上

整个网络通信量

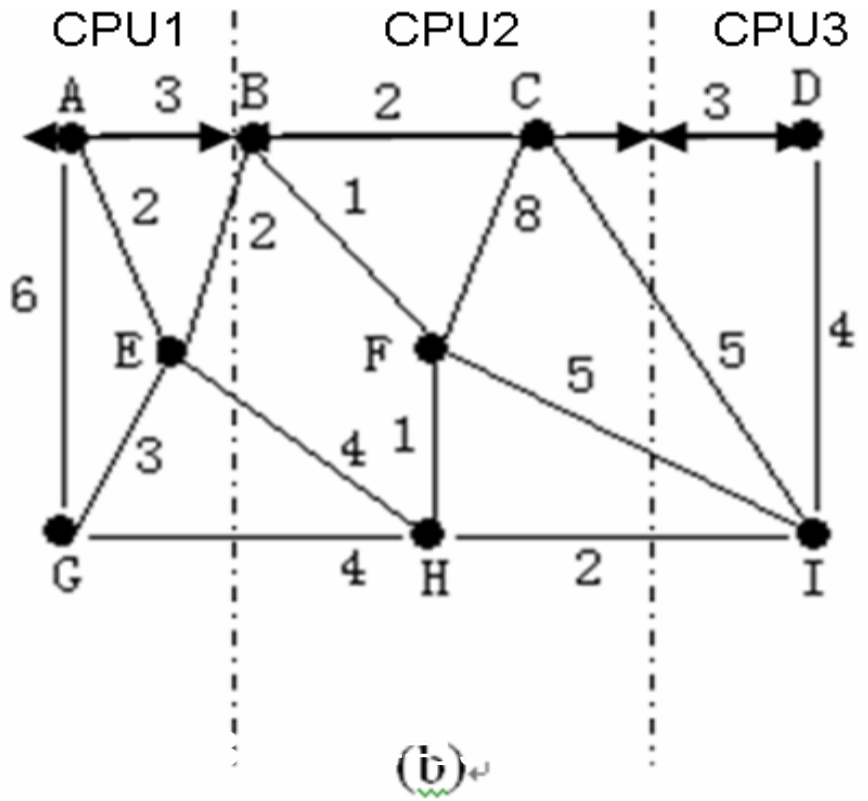
=被虚线分割开的边上的权值之和

=30



分割举例

- b中显示的分割得到的通信量之和为28
- 如果它满足所有对内存和处理机的限制，那它就是一个比较好的分割，因为它要求的网络通信量之和较小。



2. 集中式算法

- 图论算法的局限性在于：
  需要预先知道所有信息，这在一般情况下是办不到的
- 一个不需要预先知道所有信息的集中式启发式算法：“上升-下降”(up-down)算法

集中式算法

- 上升-下降算法的基本思想是

  1）由一个协调器来维护一张使用情况表
  - 每个工作站在表中都对应着一项（初始值为零）
  - 当发生一个重要事件时，就给协调器发送一个消息来更新使用情况表

  2）协调器根据使用情况表来分配处理机
  - 分配时机：调度事件发生时
  - 典型的调度事件：
    - 申请处理机
    - 处理机进入空闲状态
    - 发生时钟中断

集中式算法

- 当创建一个进程时，如果创建该进程的机器认为该进程应该在其它机器上运行，它就向协调器申请分配处理机。
- 如果有可分配的处理机时，协调器就分配一个处理机，否则，协调器就暂时拒绝该处理机的申请，并记录这个请求。

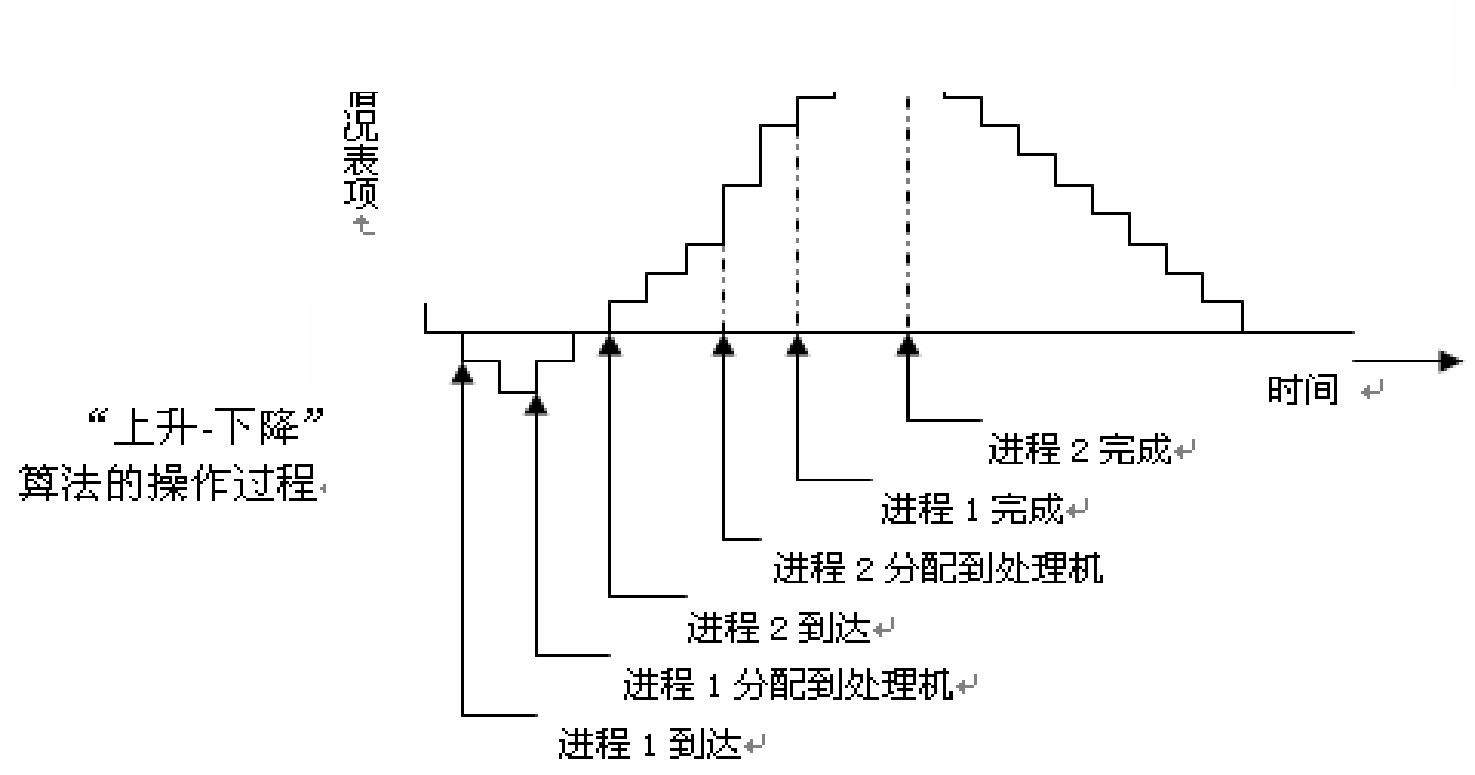
集中式算法

- 增加罚分：
  当一个工作站上的进程正在其它机器上运行时，它的罚分每秒钟增加一个固定值。这个罚分将加在使用情况表中该工作站所对应的项上。
- 减少情况1：每当工作站上的进程需要在其它机器上运行的请求被拒绝时，该工作站在使用情况表中所对应项上的罚分就会减少一个固定值。
- 减少情况2：当没有等待的处理机分配请求，并且处理机也未被使用时，使用情况表中该处理机所对应项上的罚分就会每秒钟减去一个值，直到为0。

集中式算法

- 如图，由于罚分一会儿上升，一会儿下降，算法由此得名。
- 使用情况表中的罚分可以为正数、零和负数。

- 正数表示对应工作站上的用户是在使用系统资源
- 负数表示该工作站需要系统资源。
- 零表示中性。



“上升-下降”算法的操作过程

集中式算法

- 集中式分配算法的启发性在于：
  - 当一个处理机变成空闲状态时，首先分配给罚分最低正在等待处理机的申请。因此，等待时间最长，没有使用处理机的请求将优先得到响应。
  - 实际上，若一个用户已使用了一段时间的系统资源，另一个用户刚开始申请一个进程的运行，负载较轻的后者要比负载较重的前者要优先得到资源。
- 集中式启发式算法公平地分配系统处理机。
- 模拟研究表明，在各种情况下，该算法都具有较好的性能。

3. 层次性算法

- “上升-下降”作为一个集中式算法无法适用于大型分布式系统。

  原因：

  协调器将成为整个系统的瓶颈口，此外，协调器的崩溃将造成整个系统无法进行处理机的分配。

- 上述问题可以通过使用层次分配算法来解决。
  - 层次分配算法既保持了集中式分配算法的简单性，又能更好地适应于大型分布式系统。

层次性算法

- 层次分配算法将所有处理机以一种与物理拓扑结构无关的方式组织成一个逻辑分层结构。
- 这种逻辑分层结构与公司、军队、大学等现实世界中人的层次组成结构一样。例如，可以将一些机器看作为工人，而将另一些机器看作为工头。

层次性算法

- 例如：
  - 对于每一组k个教师来说，分配给一个系主任的任务是检查观察谁正忙碌，谁正空闲。
  - 如果系统很大，那就需要更多的管理者。于是，有些机器将作为院长。每一个院长管理若干个系主任。
  - 如果院长较多，则设置一个校长来管理院长。
  - 这种层次关系可以进一步扩展下去，并且所需要的层次随着教师的数目成对数级增长。
- 由于每一个处理机只需要与一个上级和若干个下属进行通信，所以就可以对系统的信息流进行管理。

层次性算法　崩溃的解决方法1

- 当一个系主任，或者更严重地，一个院长停止了工作（即崩溃了），系统将怎么办？
- 一种方法就是
  - 由崩溃院长所管辖的一个系主任来接替该院长职位，
  - 这个院长职位

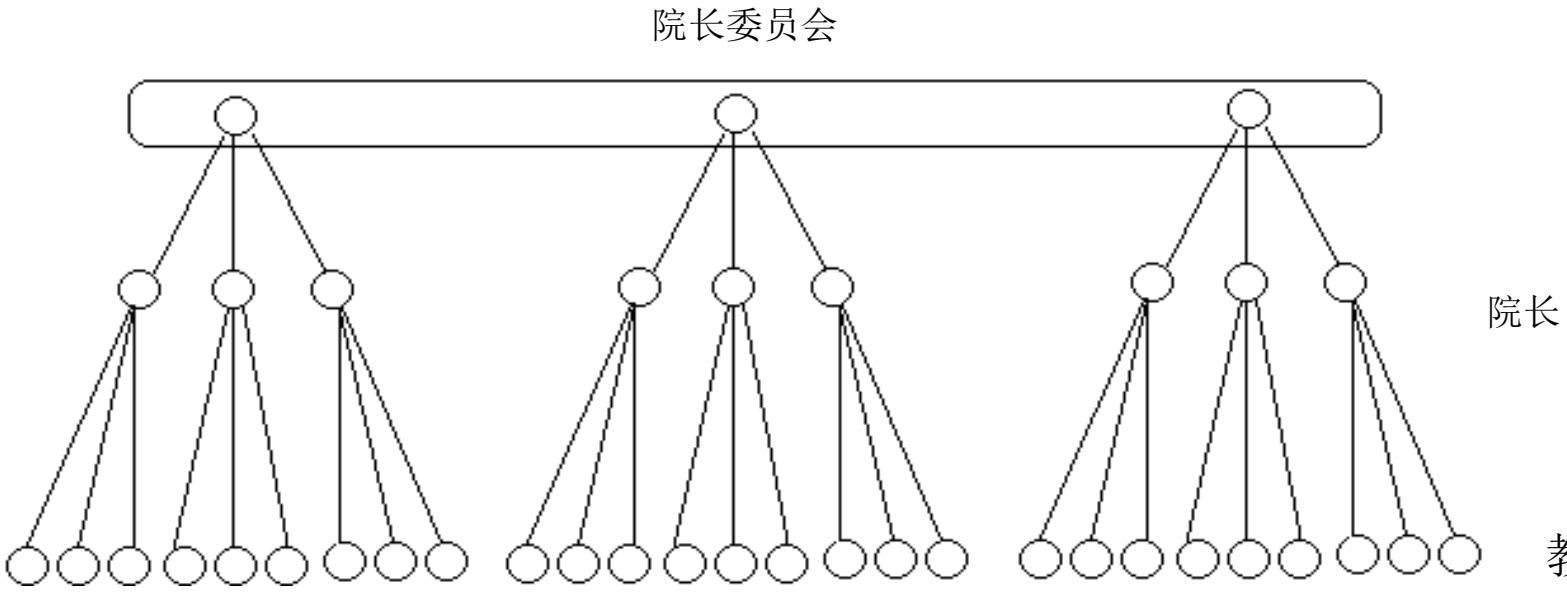
    1）可以由它下级选举产生
    2）也可以由同级院长们选举产生
    3）还可以由它的上级来任命。

层次性算法 最高委员会

- 为了避免单个管理者在层次树的最顶层（造成系统不稳定），可以像下图那样

  去掉树的根节点，最上层组成一个委员会来作为最高决策机构。

  当委员会中的一个成员不工作了，其他人员将在下一层中选出某一个成员来代替。



层次性算法 结构分析

- 结构分析：
  - 可行性：
    尽管这种层次结构并不是真正分布式的，但它却是可行的，并且实践证明它是一个较好的结构。
  - 自重构性：
    特别的是，这种系统可以自重构，并能够容忍被管理者或管理者的突发性崩溃，而不会产生任何长期的影响。

层次性算法 处理器预定

- 算法中，一个处理机只能分配一个进程。
  - 若一个作业产生S个进程，系统必须为它分配S个处理机。作业可以在层次树上的任何一层次上创建。每一个管理者跟踪并记录它辖区内有多少个处理机可用。
    - 如果有足够的处理机可供使用，那它将预定R个处理机，但R≥S必须成立，因为这种估计不一定准确，有些机器可能已经关机。

层次性算法 处理器预定

- 如果没有足够的处理机可供分配，那就把这个申请请求（逐级）向上传递，直到到达某个能够满足该请求的层次。
- 在这一层次上，管理者把这个请求分解成多个申请并向下传递给下级的管理者，一直传递到树的底层。
- 在最低层，被分配的处理机被标为“繁忙”，并把实际分配到的处理机数沿着树向上逐级报告。

层次性算法：R的取值

1）R必须足够的大以便确保有足够数量的处理机可供分配。否则，请求将沿着树向上传递。这样将会浪费了大量的时间。

2）另一方面，如果R太大，那么将有过多的处理机被标为“繁忙”，这将浪费一些计算能力，直到分配消息返回顶层，这些处理机才会被释放。

4. 超载者启动的分布式启发算法

- 一个典型的分布式启发式算法
- 算法描述：

  当一个进程创建时，若创建该进程的机器发现自己超载，就将询问消息发送给一个随机选择的机器，询问该机器的负载是否低于一个阀值。

  1）如果是，那么该进程就被传送到该机器上去运行。

  2）否则，就再随机地选择一台机器进行询问。

  这个过程最多执行N次，若仍然找不到一台合适的机器，那么算法将终止，新创建的进程就在创建它的机器上运行。

算法分析

- 当整个系统负载很重的时候，
  - 每一个机器都不断地向其他机器发送询问消息以便找到一台机器愿意接收外来的工作。
  - 在这种情况下，所有机器的负载都很重，没有一台机器能够接收其它机器的工作，所以，大量的询问消息不仅毫无意义，而且还给系统增添了巨大的额外开销。

5. 欠载者启动的分布式启发算法

- 在这个算法中，当一个进程结束时，系统就检查自己是否欠载。
  - 如果是，它就随机地向一台机器发送询问消息。
  - 如果被询问的机器也欠载，则再随机地向第二台、第三台机器发送询问消息。
  - 如果连续N个询问之后仍然没有找到超载的机器，就暂时停止询问的发送，开始处理本地进程就绪队列中的一个等待进程，处理完毕后，再开始新一轮的询问。
  - 如果既没有本地工作也没有外来的工作，这台机器就进入空闲状态。
  - 在一定的时间间隔以后，它又开始随机地询问远程机器。

算法分析

- 在欠载者启动的分布式启发式算法中，
  - 当系统繁忙时，一台机器欠载的可能性很小。即使有机器欠载，它也能很快地找到外来的工作。
  - 在系统几乎无事可做时，算法会让每一台空闲机器都不间断地发送询问消息去寻找其它超载机器上的工作，造成大量的系统额外开销。
  - 但是，在系统欠载时产生大量额外开销要比在系统过载时产生大量额外开销好得多。

算法比较

- 与超载者启动的分布式启发式算法相比：
  - 欠载者启动的算法不会在系统非常繁忙时给系统增加额外的负载。
  - 而超载者启动的算法中，一台机器却在系统非常繁忙时发送大量的毫无意义的询问。

超/欠载者启动的结合

- 可以将上述两种算法结合起来，让超载机器清除一些工作，而让欠载机器去寻找一些工作。

6. 拍卖算法

- 拍卖算法把分布式系统看作为一个小经济社会，由买卖双方和供求关系来决定服务的价格。
- 进程为了完成自己的任务必须购买处理机时间，而处理机将它的处理机时间拍卖给出价最高的进程。
  - 每一个处理机将自己估计的价格写入一个公共可读的文件中以此来进行拍卖。

6. 拍卖算法

- 价格并不是一直不变的，初始的价格只是表示所提供服务的近似价格（一般，它是以前最后一个买主出的价格）
- 根据处理机的运算速度、内存大小、浮点运算能力以及其它一些特性来确定每一个处理机的价格。
- 处理器提供的服务（例如，预计的响应时间）也要公布出来

拍卖算法

当一个进程要启动一个子进程时，

1. 查询公共可读文件看有谁能够提供它所需要的服务。
2. 确定一个它可以付得起钱的处理机集合。通过计算从这个集合中选出一个最好的处理机。最好的标准是最便宜、速度最快或者性能价格比最高。
3. 给第一个选中的处理机发送一个出价信息，这个出价有可能高于或低于处理机公布的价格。

拍卖算法

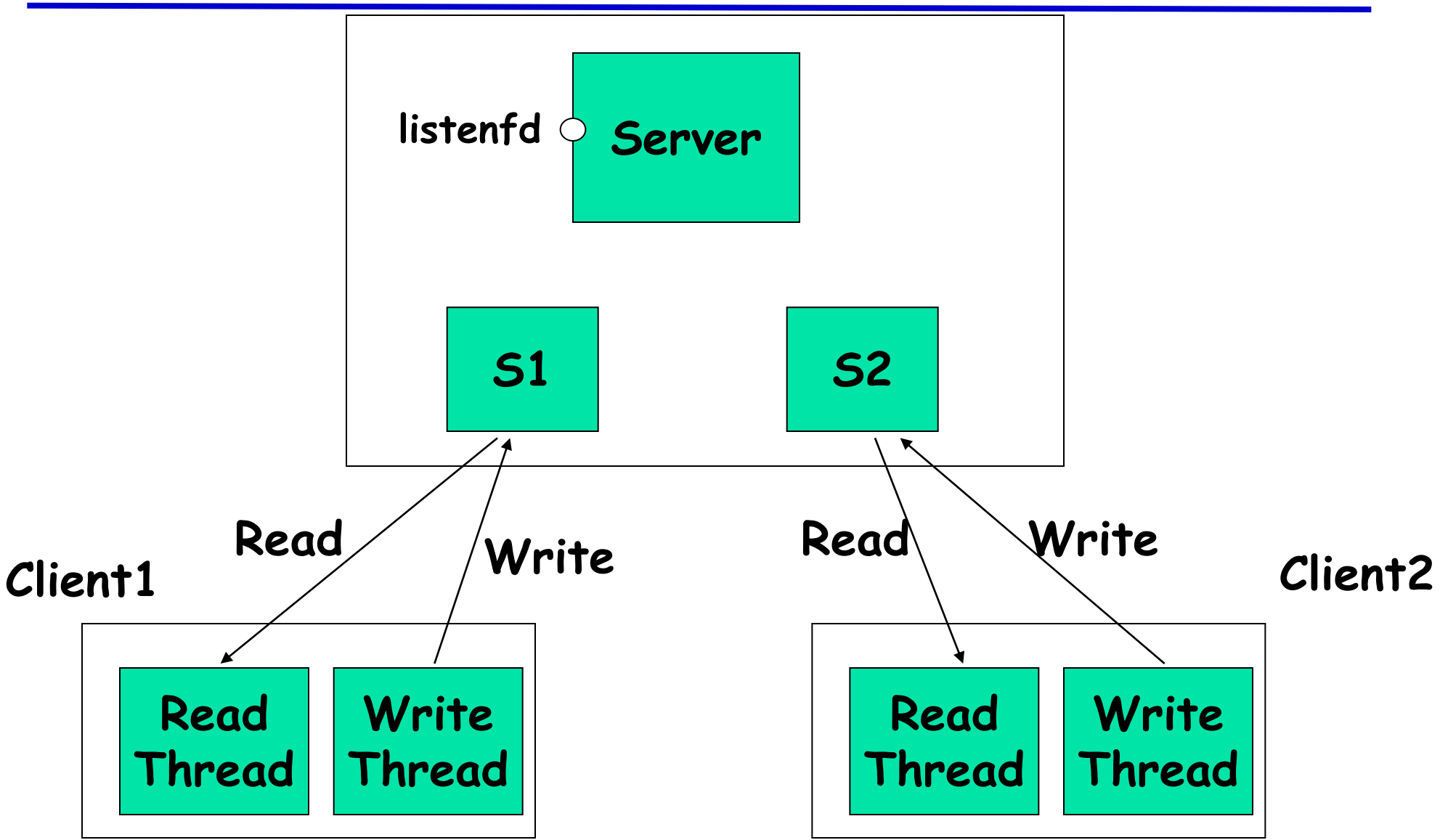
- 处理机
  1. 收集所有发送给它的出价信息
  2. 选择一个出价最高的进程并将通知发送给选中的进程和未选中的进程。
  3. 开始执行被选中的进程。
     此时，公共可读文件中该处理机的价格将被更新以便反映处理机当前最新的价格。

拍卖算法：问题

- 该算法所引起的问题是：
  - 进程从哪里获得钱来购买处理机？
  - 它们有稳定的工资收入吗？
  - 每个进程的月薪都相同吗？还是有些进程的工资高有些进程的工资低呢？
  - 如果用户数量增加而系统未增加相应的一些资源，那么，会不会造成系统通货膨胀？
  - 处理机之间会不会形成同盟来漫天要价敲进程的竹杠？
  - 进程联合工会是否允许这样做呢？
  - 使用磁盘是否也要收费？激光打印机是否收费更高？
  - 等等。

附：Thread-based Echo client-server 程序样例

Thread-based Echo client-server



Thread-based Echo Server

```
main()
{
    int listenfd, connfd;
    int len;
    pthread_t tid;

    /* Start the usual way */
    listenfd = Socket(...);
    Bind(listenfd, ...);
    Listen(listenfd, ...)

    for ( ; ; ) {
        len = addrlen;
        cptr = Malloc(sizeof(int));
        *cptr = Accept(listenfd, ...);

        /* Create a thread in service_func routine */
        Pthread_create(&tid, NULL, service_func, (void *) cptr);
    }
}
```

```
void * service_func(void *arg)
{
    int local_connfd;

    /* release parent from waiting */
    Pthread_detach(pthread_self());

    /* extract connfd from argument */
    local_connfd = *((int *) arg);
    free(arg);

    /* receive and echo client's message      */
    str_echo(local_connfd);

    /* Terminate the connection */
    Close(local_connfd);

    return(NULL);
}
```

Thread-based Echo Client

Single threaded client is not good enough.

Multi-threaded Client ➔

```
int sockfd;
FILE *fp;

main()
{
    pthread_t tid;
    fp = fopen(…);

    /* Start the usual way */
    sockfd = Socket(…);
    …
    Connect(…);

    /* Create a thread to send data */
    Pthread_create(&tid, NULL, write_func, NULL);

    /* read data from sockfd */
    read_func();

    /* wait for child thread */
    Pthread_join(tid, NULL);
}
```

```
void * write_func(void *arg)
{
   char sendline[MAXLINE];
   while( more data in fp)
        Read from fp into sendline[];
        Write sendline[] into sockfd;
   Shutdown(sockfd, SHUT_WR);
   return(NULL);
}
void read_func()
{
   char recvline[MAXLINE];
   while ( more data from sockfd)
     Read from sockfd into recvline[];
     Write from recvline[] to stdout;
}
```

Locking in Threads

Mutex – for mutual exclusion

```
int counter = 0;

void *thread_func(void *arg)
{
    int val;

    /* unprotected code */
    val = counter;
    counter = val + 1;

    return NULL;
}
```

```
int counter = 0;
ptread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *thread_func(void *arg)
{
    int val;

    /* protected by mutex */
    Pthread_mutex_lock( &mutex );
    val = counter;
    counter = val + 1;
    Pthread_mutex_unlock( &mutex );

    return NULL;
}
```

Condition Variable – for signaling

- Think of Producer – consumer problem
- Producers and consumers run in separate threads.
- Producer produces data and consumer consumes data.
- Producer has to inform the consumer when data is available
- Consumer has to inform producer when buffer space is available

Without Condition Variables

```
/* Globals */
int data_avail = 0;
pthread_mutex_t data_mutex = PTHREAD_MUTEX_INITIALIZER;

void *producer(void *)
{
   Pthread_mutex_lock(&data_mutex);

   Produce data

   Insert data into queue;
   data_avail=1;

   Pthread_mutex_unlock(&data_mutex);

}
```

```
void *consumer(void *)
{
    while( !data_avail );
      /* do nothing – keep looping!!*/

    Pthread_mutex_lock(&data_mutex);

    Extract data from queue;
    if (queue is empty)
                    data_avail = 0;

    Pthread_mutex_unlock(&data_mutex);

    consume_data();
}
```

With Condition Variables

```
int data_avail = 0;
pthread_mutex_t data_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cont_t data_cond = PTHREAD_COND_INITIALIZER;

void *producer(void *)
{
    Pthread_mutex_lock(&data_mutex);

    Produce data

    Insert data into queue;
    data_avail = 1;

    Pthread_cond_signal(&data_cond);

    Pthread_mutex_unlock(&data_mutex);

}
```

```
void *consumer(void *)
{
   Pthread_mutex_lock(&data_mutex);

   while( !data_avail ) {
     /* sleep on condition variable*/
     Pthread_cond_wait(&data_cond, &data_mutex);
     }

   /* woken up */
   Extract data from queue;
   if (queue is empty)
                data_avail = 0;

   Pthread_mutex_unlock(&data_mutex);

   consume_data();
}
```