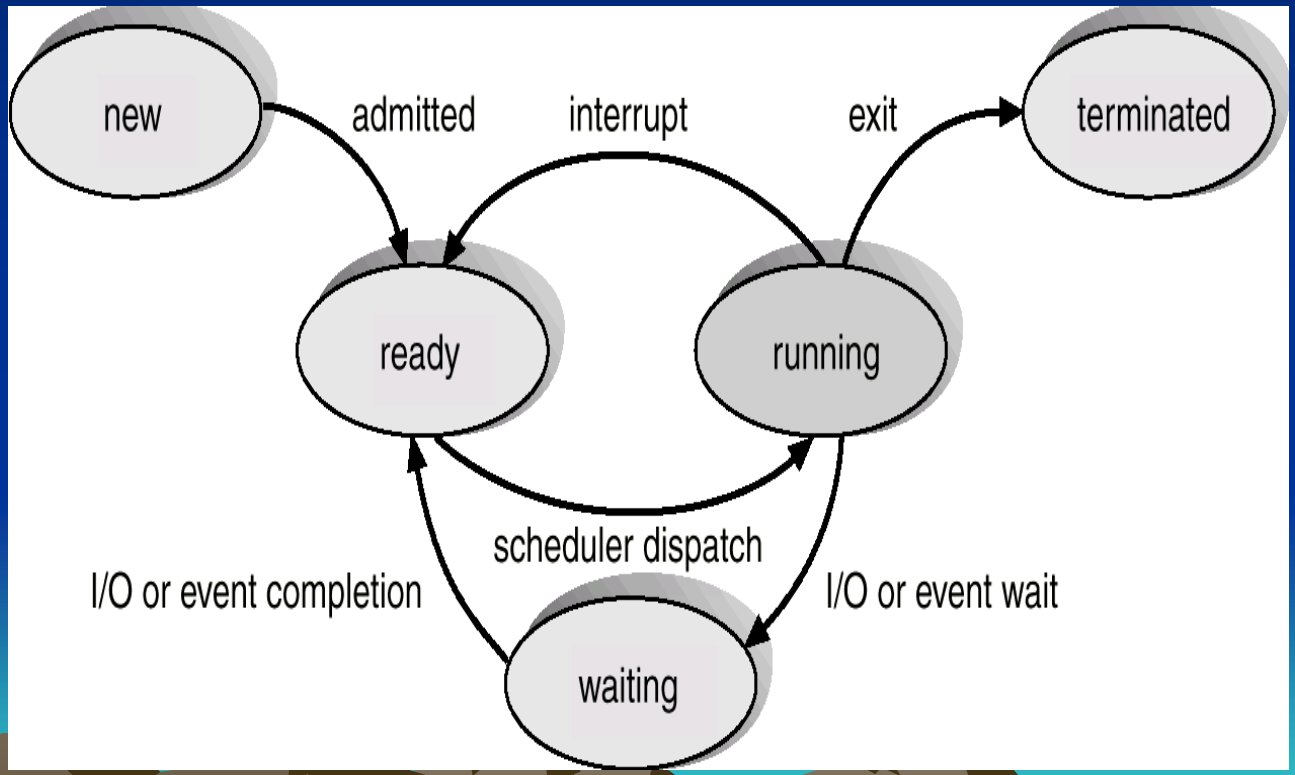
第三章 分布式进程管理

- 线程
- 代码迁移
- 处理器任务分配

进 程

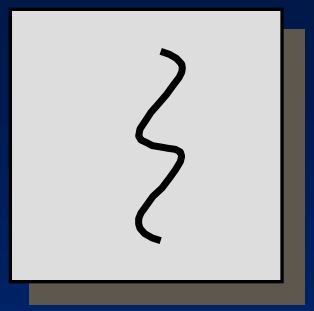
定义：执行中的程序
进程控制块（PCB）



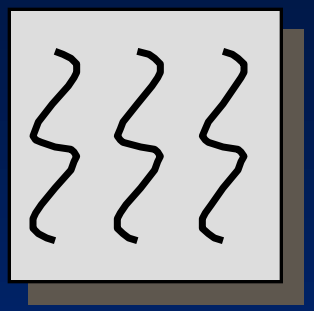
进程的状态

线 程

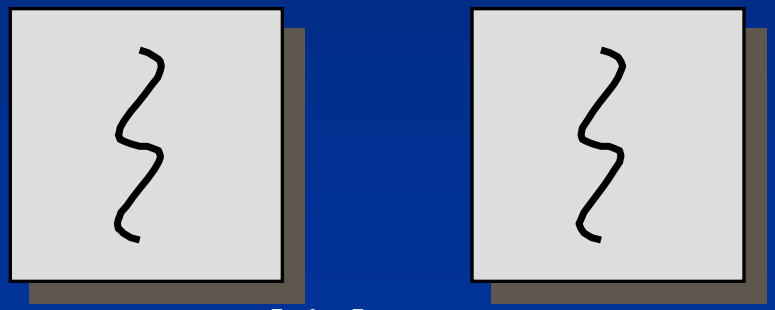
- 未引入线程前的进程：资源分配单位（存储器、文件）和CPU调度（分配）单位。
- 线程：成为CPU调度单位，而进程只作为其他资源分配单位。
  - 线程只拥有必不可少的资源，如：线程状态、寄存器上下文和栈
  - 同样具有就绪、阻塞和执行三种基本状态
  - 进程的终止导致它包含的所有线程的终止
- 线程的优点：减小并发执行的时间和空间开销（线程的创建、退出和调度），因此容许在系统中建立更多的线程来提高并发程度。
  - 线程的创建时间比进程短；
  - 线程的终止时间比进程短；
  - 同进程内的线程切换时间比进程短；
  - 由于同进程内线程间共享内存和文件资源，可直接进行不通过内核的通信；



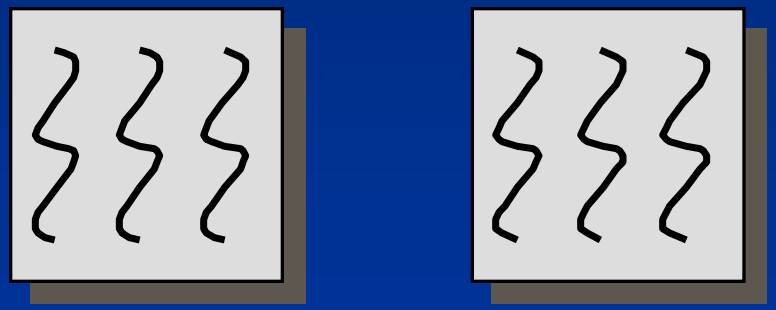
one process
one thread



one process
multiple threads



multiple processes
one thread per process



multiple processes
multiple threads per process

进程与线程的关系

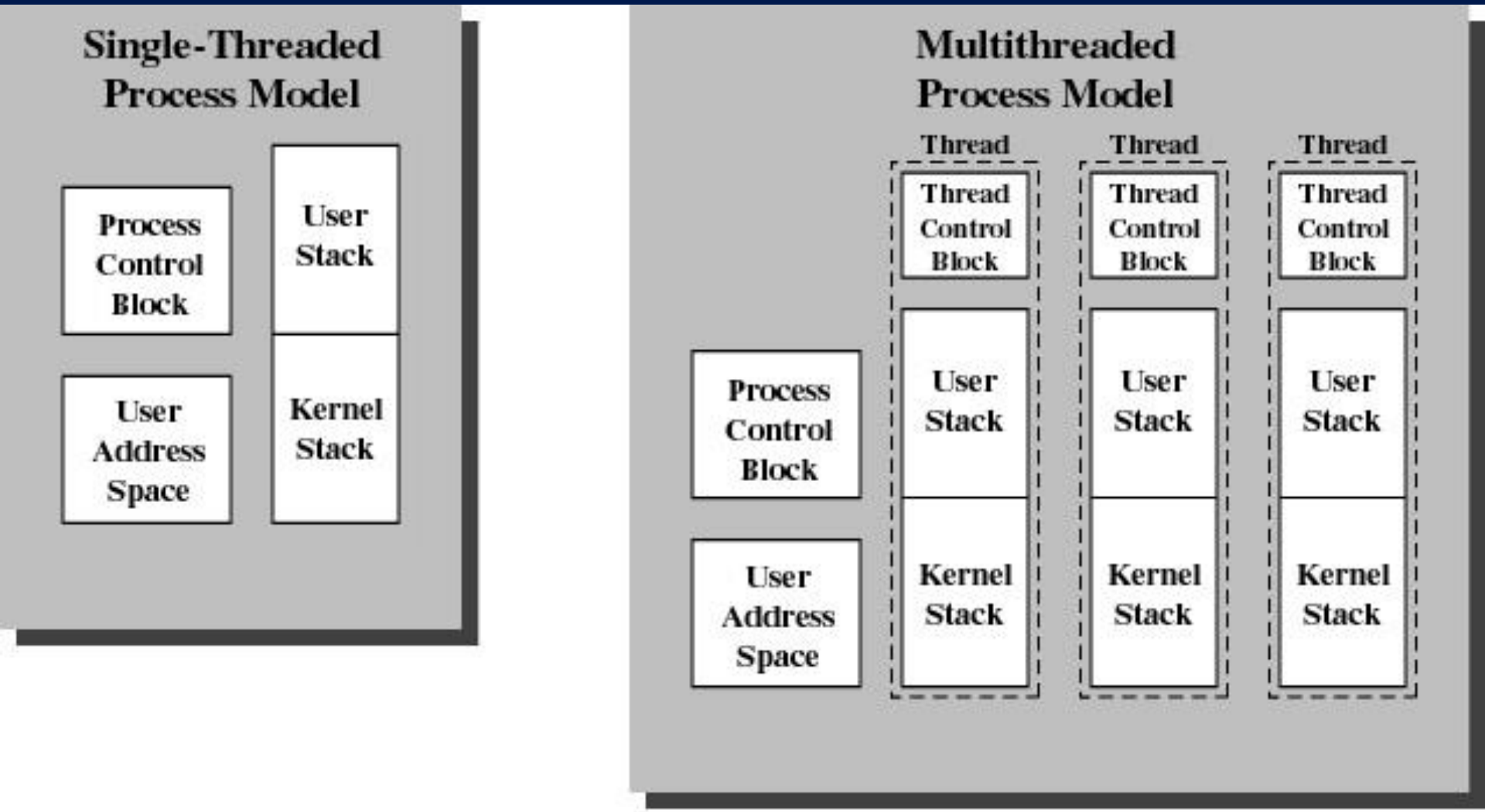
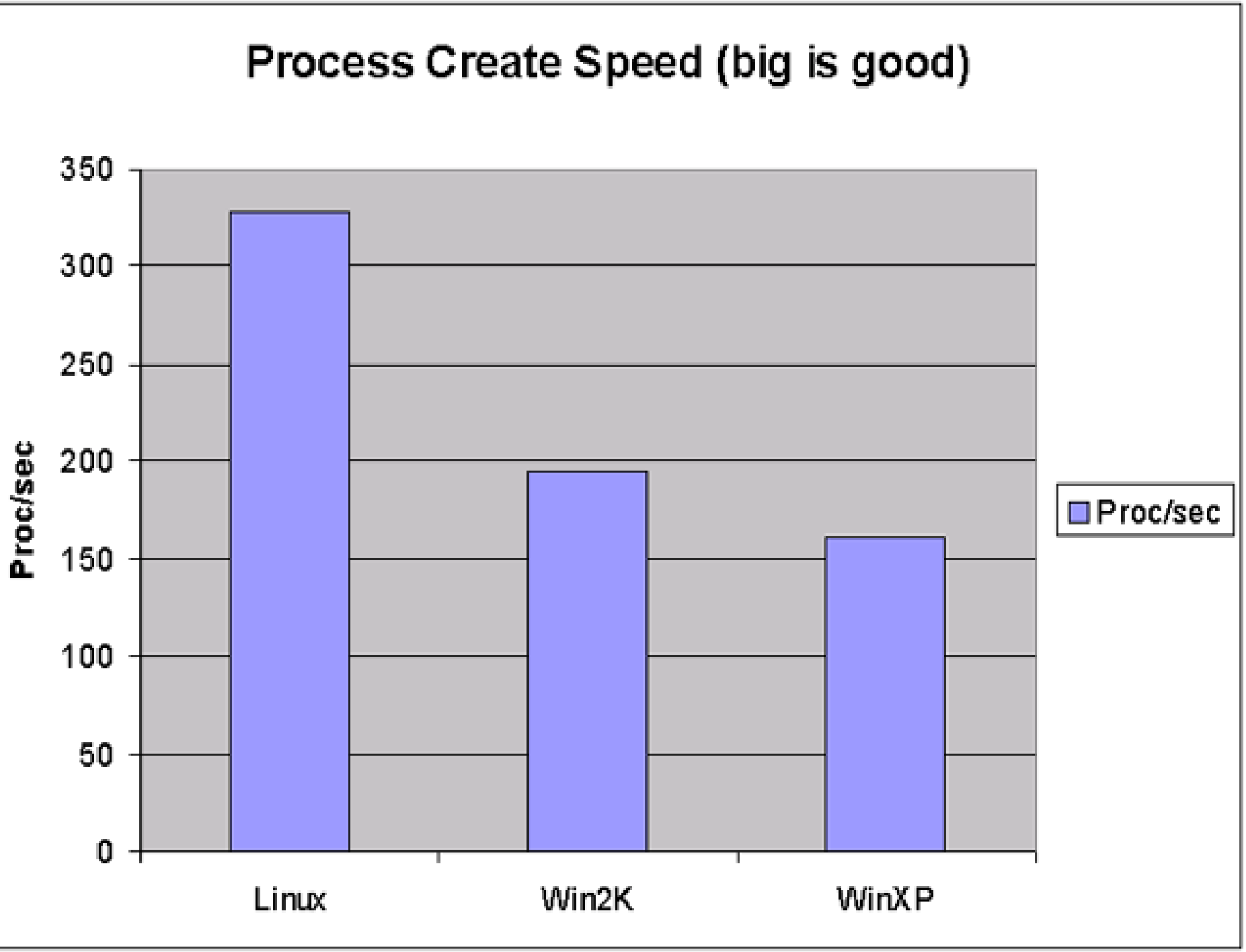
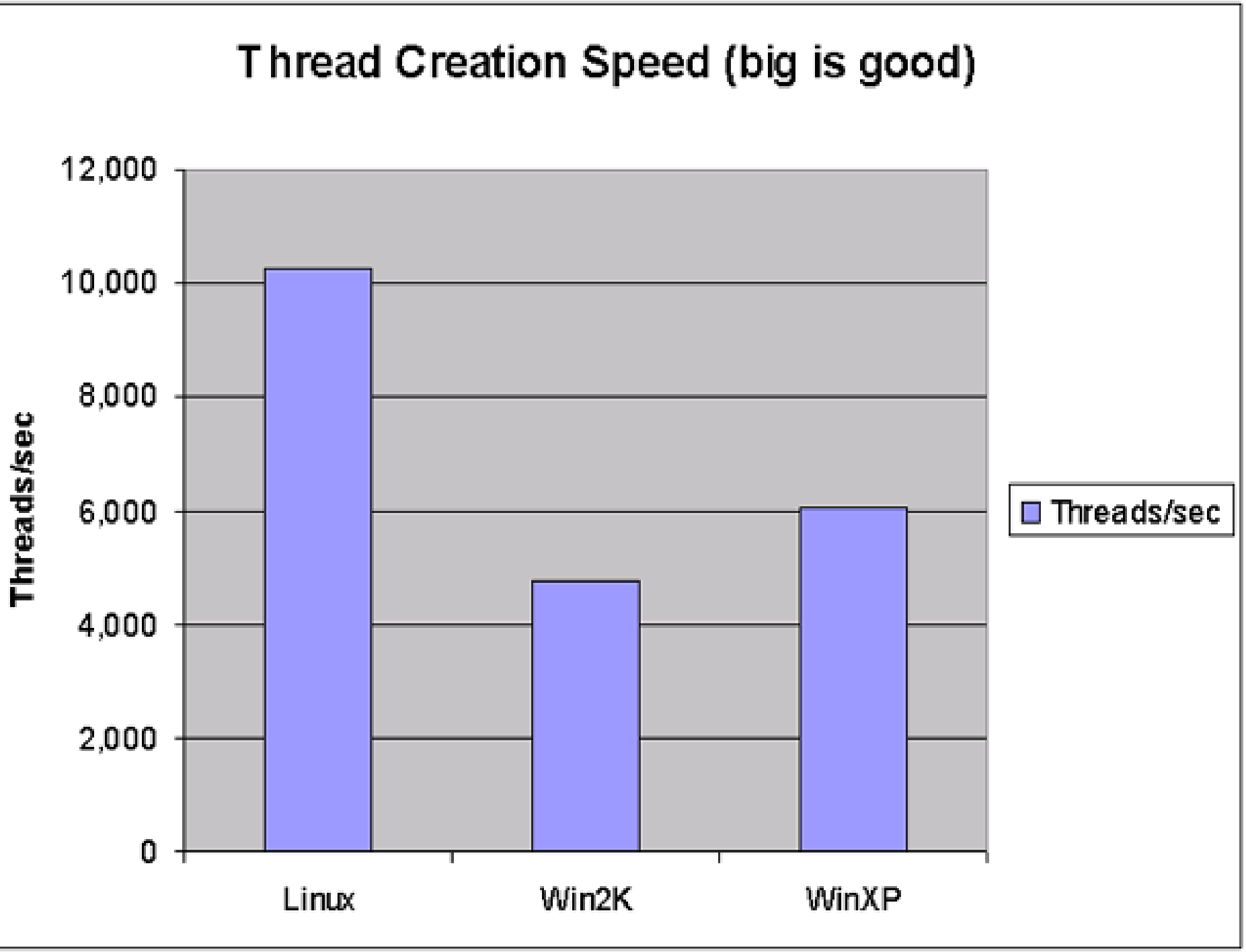


Figure 4.2 Single Threaded and Multithreaded Process Models


Process Create Speed (big is good)
Proc/sec
350
300
250
200
150
100
50
0
Linux
Win2K
WinXP
Proc/sec


Thread Creation Speed (big is good)
12,000
10,000
8,000
6,000
4,000
2,000
0
Threads/sec
Linux
Win2K
WinXP
Threads/sec

进程和线程的比较

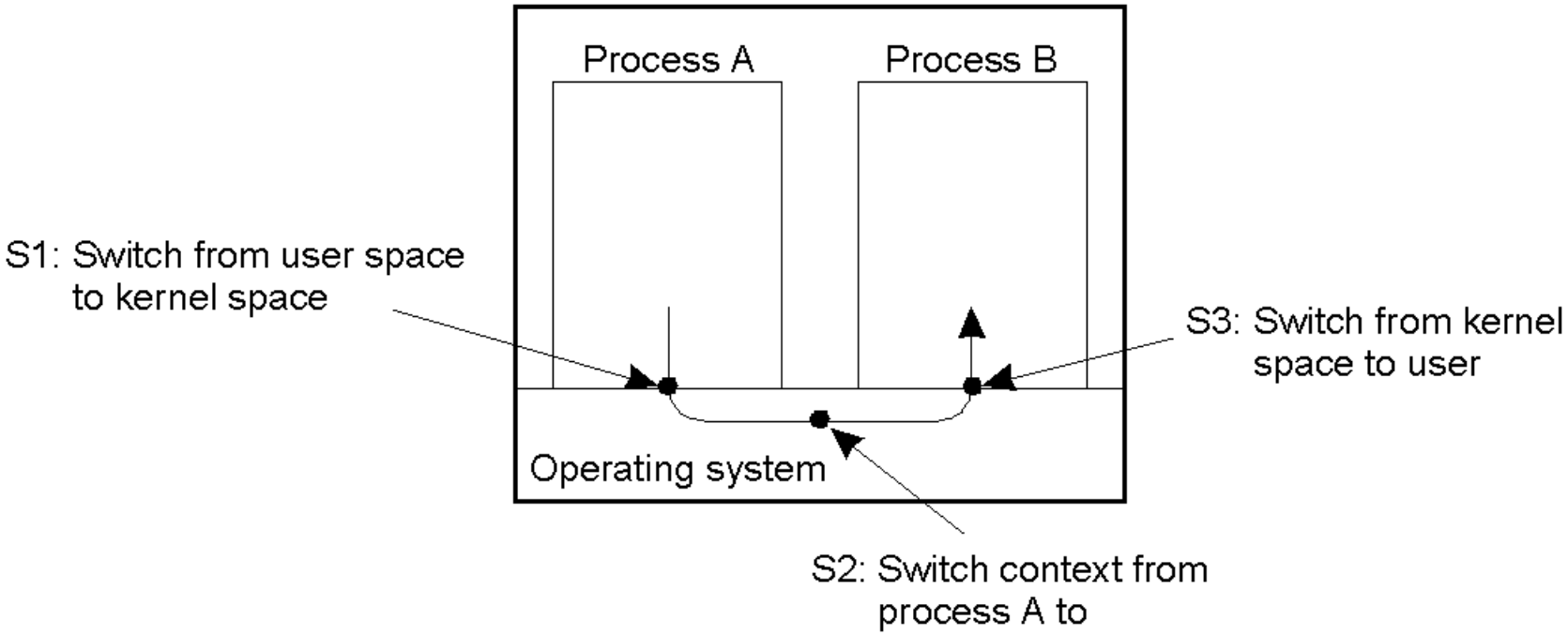
- 地址空间和其他资源（如打开文件）：进程间相互独立，同一进程的各线程间共享该进程地址空间和其他资源——某进程内的线程在其他进程不可见
- 通信：进程间通信通过IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性
- 调度：线程上下文切换比进程上下文切换要快得多；
- 结论：
  - 多线程能提高性能
  - 线程不像进程那样彼此隔离，并受到系统自动提供的保护，因此多线程应用程序开发需要付出更多努力

非分布式系统中线程的使用

- 使用多线程的优点：
  - 在某线程阻塞时，其他线程可以继续工作
  - 利用多处理器，并行工作
  - 缩短IPC通信的时间
  - 出于软件工程的考虑:字处理程序(用户输入、拼写检查、语法检查、文档布局)

非分布式系统中线程的使用

- IPC导致的进程上下文切换

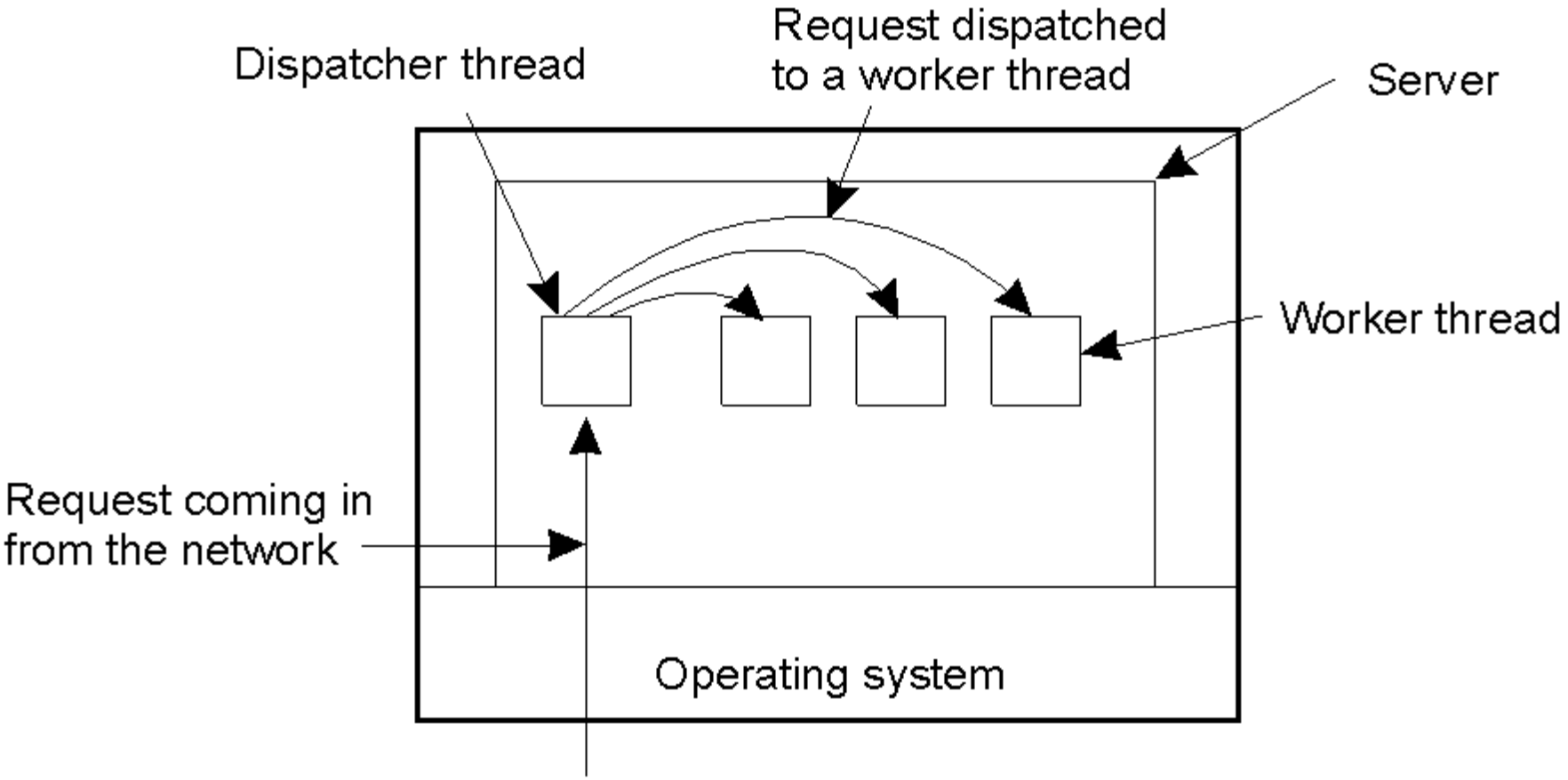


分布式系统中线程的使用

- 多线程客户：Web浏览器
- 多线程服务器

多线程服务器

- 以分发器/工作者组织的多线程服务器



代码迁移

定义：将程序（或执行中的程序）传递到其它计算机

迁移动机：

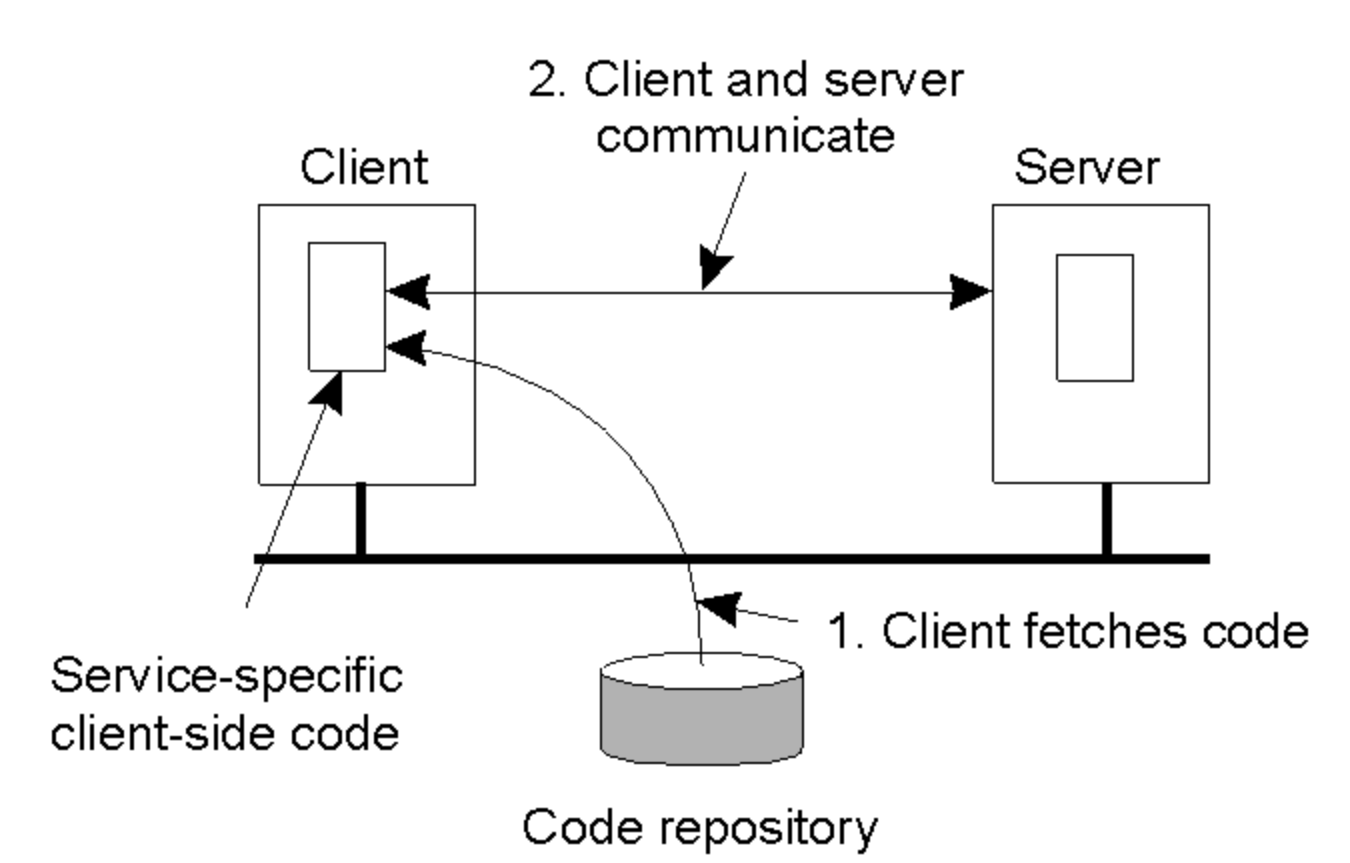
- 实现负载均衡
  - 将进程从负载重的系统迁移到负载轻的系统，从而改善整体性能
- 改善通信性能
  - 交互密集的进程可迁移到同一个节点执行以减少通信开销
  - 当进程要处理的数据量较大时，最好将进程迁移到数据所在的节点

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

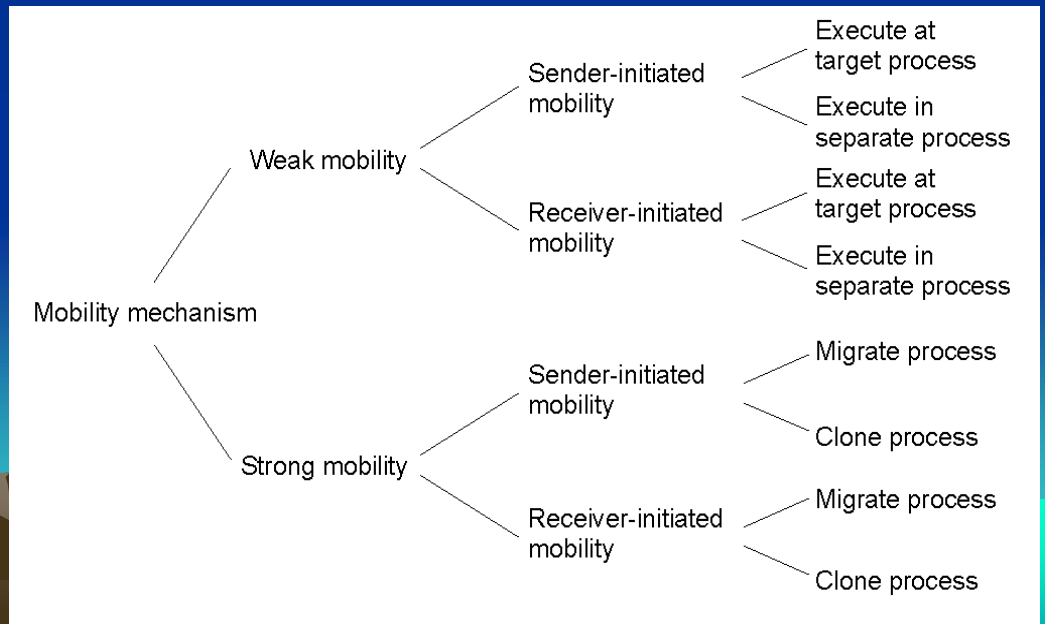
代码迁移-灵活性

- 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，客户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



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

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：

- 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
- 发送者启动算法和接收者启动算法

局部性算法和全局性算法

第四个设计问题与迁移策略有关。

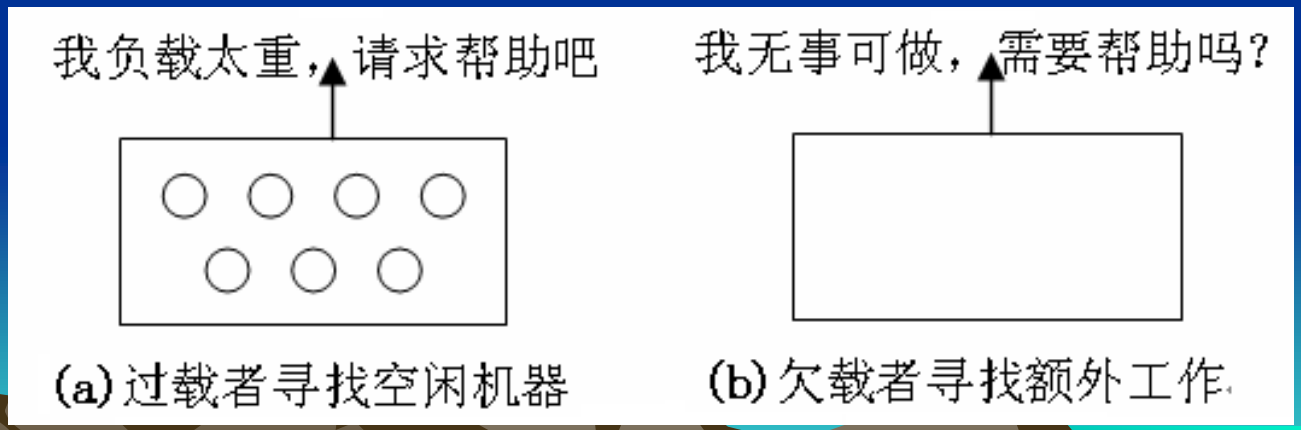
- 当一个新进程被创建时，系统需要决定它是否在创建它的机器上运行。若该机器繁忙，那这个新进程就必须迁移到其它机器上去运行。
- 对于是根据本机局部信息还是全局信息来决定新进程是否迁移，目前存在着两种学派。

  1）一种学派主张简单的局部算法：若机器的负载低于某个阈值，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。
  2）另一种学派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。
- 比较：
  局部算法简单，但远远达不到最优；
  而全局算法需要可能付出巨大的代价来换取一个性能稍微好一点的结果。

发送者启动算法和接收者启动算法

- 最后一个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，却在实现上却复杂得多。

处理机分配算法的实现问题

- 然而，**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

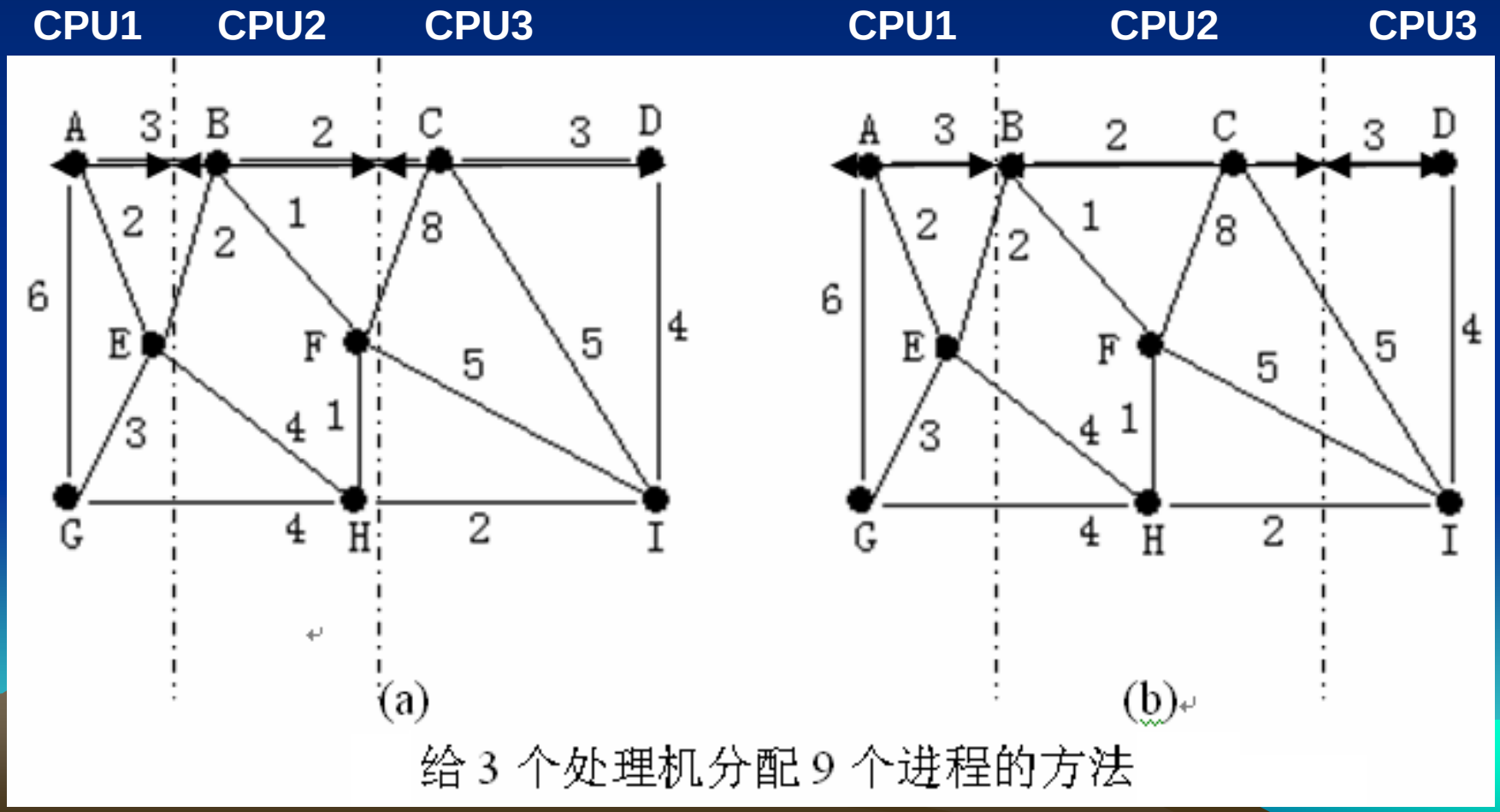
- 假定：每个进程都知道
  1）所需的处理机
  2）所要求的内存
  3）系统中任意一对进程间的平均通信量
- 若系统中处理机的数目k比进程数少，那么系统中的一些处理机就必须被分配多个进程
- 基于图论的确定性算法
  保证在系统网络通信量最小的条件下对处理机进行分配。

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阈值）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

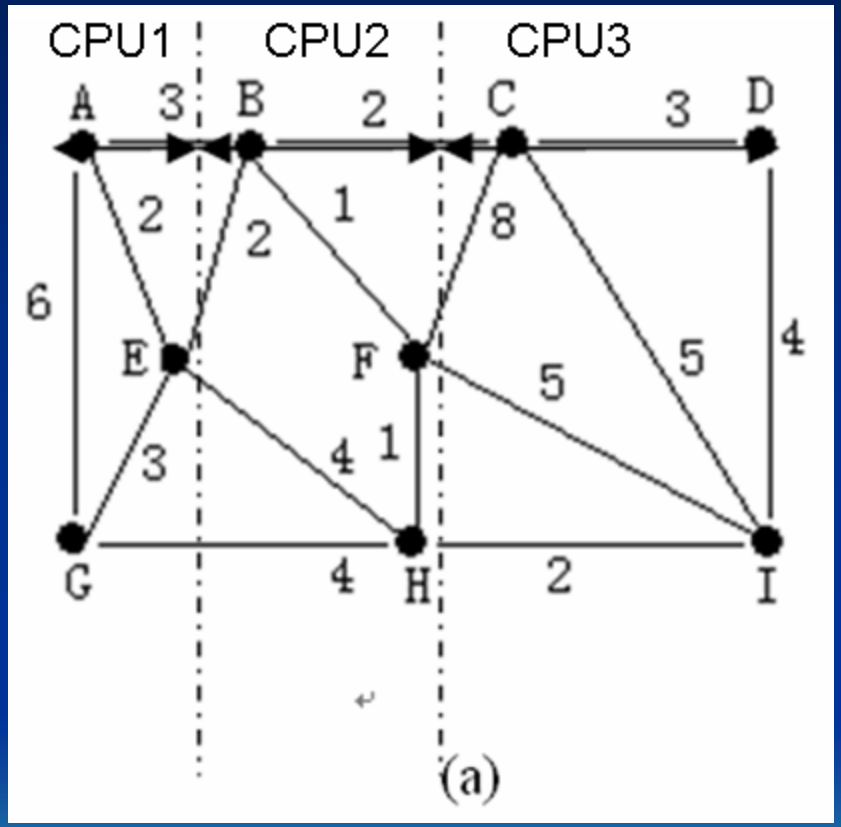
a中，系统图被分割为：

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

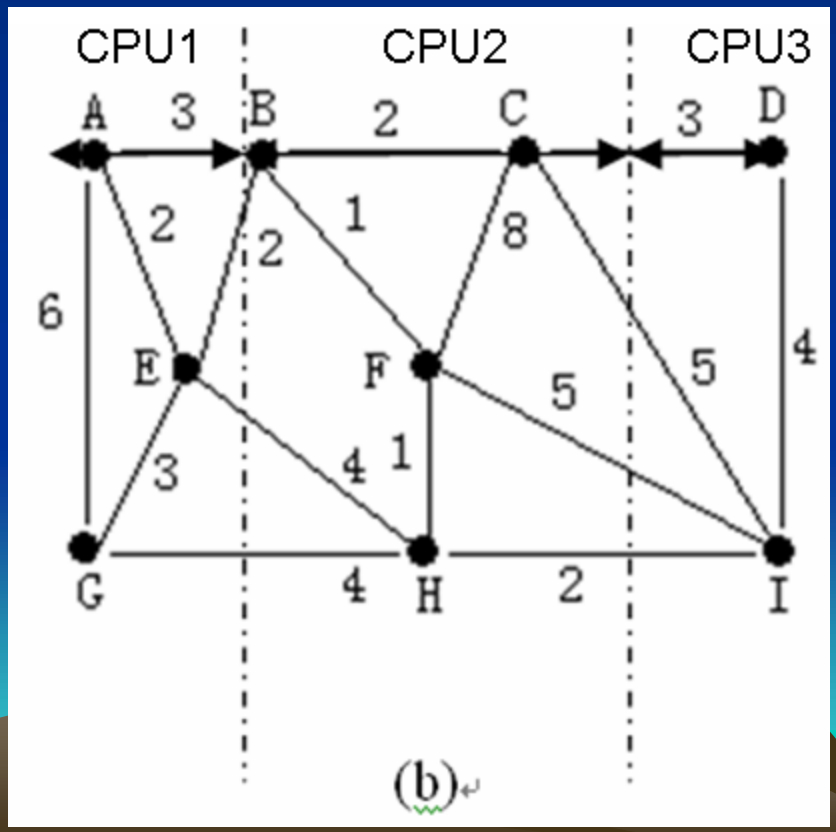
整个网络通信量
= 被虚线分割开的边上的权值之和 = 30



3+2+4+4=13　　2+8+5+2=17

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。



3+2+4+4=13

3+5+5+2=15

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。

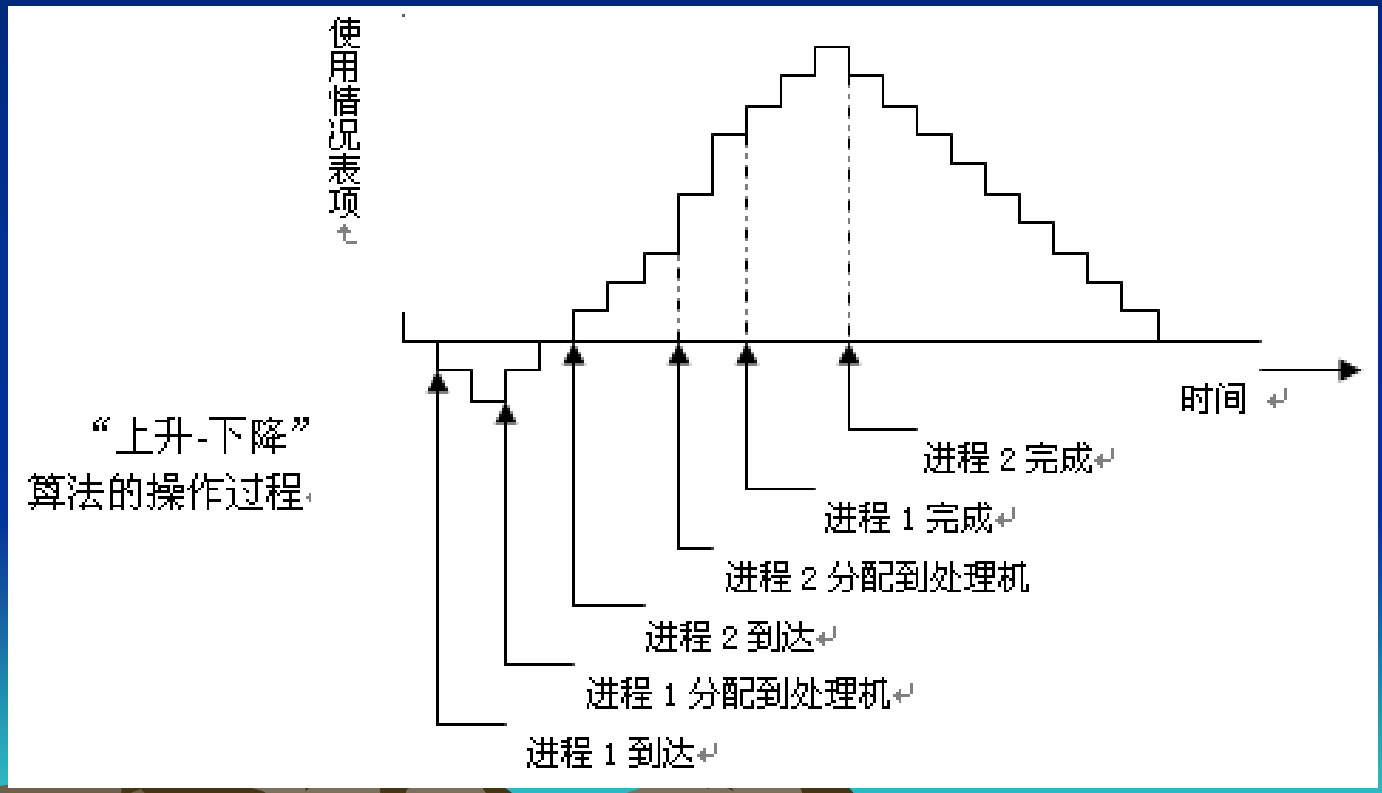
集中式算法

- 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：当工作站没有等待的处理机分配请求，并且也未使用处理机时，使用情况表中该工作站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

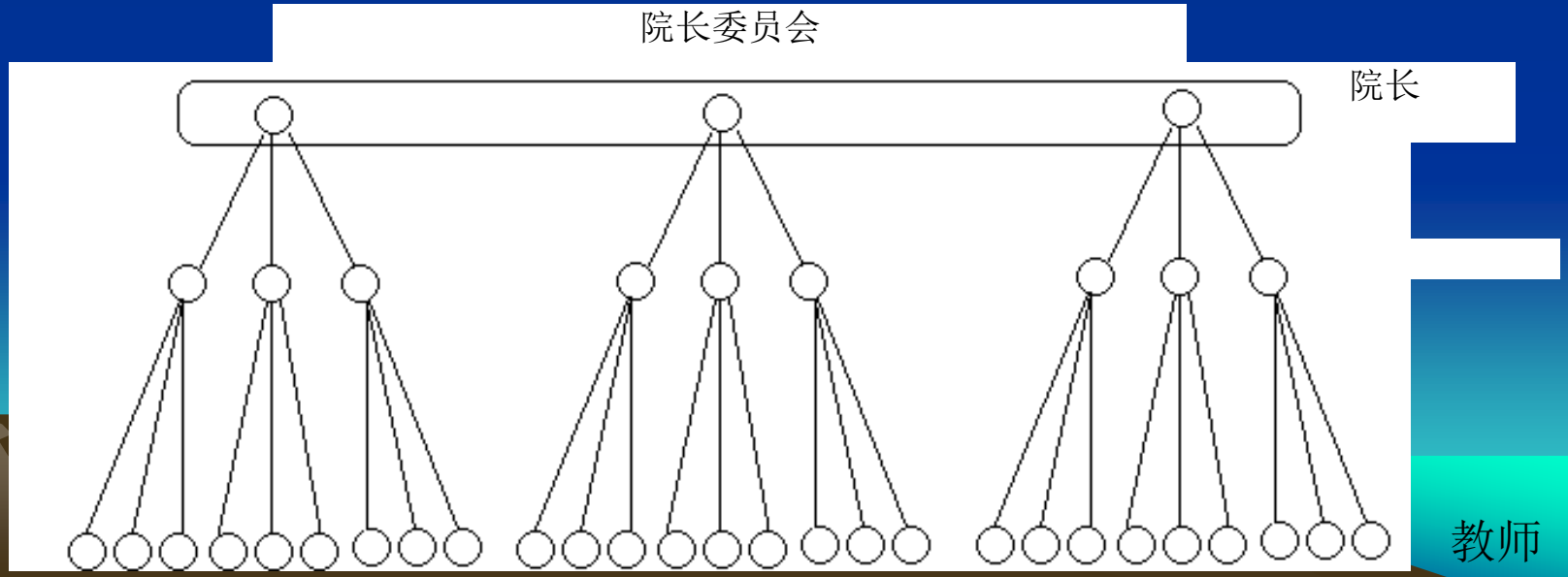
- 当一个系主任，或者更严重地，一个院长停止了工作（即崩溃了），系统将怎么办？
- 一种方法就是
  - 由崩溃院长所管辖的一个系主任来接替该院长职位，
  - 这个院长职位
    1）可以由它下级选举产生
    2）也可以由同级院长们选举产生
    3）还可以由它的上级来任命。

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：
    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式算法

- 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。
- 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。

小 结

- 线程
- 代码迁移
- 处理器分配

习 题

1. 比较在单处理器系统中，使用单线程文件服务器和使用多线程文件服务器读取文件有什么区别。假定需要的数据存放在内存的缓存中（概率为2/3），将花费30ms来接收请求、调度该请求并且完成其他必需的处理工作。否则需要磁盘操作，就需要额外多花90ms，在磁盘操作的过程中线程处于等待状态。

- 如果服务器采用单线程，每秒能处理多少个请求？
- 如果服务器采用多线程呢？

2. 对服务器进程中的线程数目进行限制是否有意义？
3. 列举通过生成进程来构建并发服务器与使用多线程服务器的优点和缺点。