Exam OS Learning Note

考研 OS 学习

ElegantLaTeX Program

Victory won't come to us unless we go to it.

作者：Miao

时间：September 1, 2023

邮箱: chenmiao.ku@gmail.com

版本: 0.10

目　　录

第 1 章 计算机系统概述

1.1 操作系统的基本概念

1.1.1 操作系统的概念

操作系统 *(Operator System)* 是指控制和管理整个计算机系统的硬件和软件资源，合理地组织、调度计算机的工作与资源的分配，进而为用户和其他软件提供方便接口与环境的程序集合。

操作系统是计算机系统中最基本的系统软件。

1.1.2 操作系统的特征

操作系统有如下四大特征：

1. 并发 (Concurrence)

   并发是指两个或多个事件在同一时间间隔内发生。值得注意的是：并发和并行是两个不同的概念，并行一定并发，并发不一定并行。对于单处理器来说，只能并发执行。

   并发是 *OS* 最为基础且必要的特征。

2. 共享 (Sharing)

   共享即资源共享，是建立在并发之上的。

   (互斥式共享) 系统中的某些资源在一个时间段内有且仅能一个程序进行访问和操作，且把该资源称为临界资源。

   (同时式共享) 该类资源可以由多个程序同时段进行访问 (多出现在读操作上)

   并发和共享是操作系统最基本的特征，两者之间互为存在的条件：资源共享必定是由并发产生的，而共享影响了并发则会导致并发崩溃。

3. 虚拟 (Virtual)

   虚拟是指把一个物理上的实体变为若干逻辑上的对应物。物理实体是实际存在的，而虚拟逻辑是用户感觉上的事物。

   虚拟处理器的存在：时分复用技术

   虚拟内存的存在：空分复用技术

4. 异步 (Asynchronism)

   多道程序环境允许多个程序执行，但由于资源有限 (竞争的情况)，进程并非一贯到底的执行。而且走走停停，不停的切换，因此以不可预知的速度向前推

进。

1.1.3 操作系统的目标与功能

为了给多道程序提供环境，*OS* 应该具备：处理机 *(进程)* 管理、存储器 *(内存)* 管理、设备管理和文件管理 *(现代操作系统中，不仅仅这几个模块)*。

除了上述的模块外，还需要提供各种接口。

操作系统作为计算机系统资源的管理者

1) 处理机 (进程) 管理

在多道程序环境下，处理机的分配和运行都以进程 (或线程) 为基本单位，因此对处理机管理可归结为进程管理。

并发指的是计算机内同时运行多个进程，因此如何管理进程则是最主要的任务。进程管理的主要功能包括进程控制、进程同步、进程通信、死锁处理、处理机调度等。

2) 存储器 (内存) 管理

内存管理是为了给多道程序的运行提供良好的环境，方便用户使用以及提高利用率。其主要包括内存分配与回收、地址映射、内存保护与共享和内存扩充等。

3) 文件管理

计算机中的信息都是以文件的形式存在的，文件管理包括文件存储空间的管理、目录管理以及文件读写管理与保护等。

4) 设备管理

设备管理的主要任务是完成用户的 I/O 请求，方便用户使用各种设备，提高设备利用率。主要包括缓冲管理、设备分配、设备处理和虚拟设备等。

操作系统作为用户与计算机硬件系统之间的接口

1) 命令接口

(联机命令接口 (交互式)) 适用于分时或实时操作系统。由一组键盘命令组成，用户通过控制台或 *terminal* 输入命令与 *OS* 进行交互。类似于 Python 交互器。

(脱机命令接口 (批处理)) 适用于批处理系统。由一组作业控制命令组成，不能直接干预作业的运行，事先使用对应的作业控制命令写成一份操作流程，然后递交给 *OS*。类似于 Bash 脚本。

2) 程序接口

程序接口由系统调用 (又称广义指令) 组成，例如 printf、malloc 等 C 语言调用接口。

GUI 就是图形接口，其最终是通过调用程序接口实现的。严格的说，GUI 不是操作系统的一部分，但 GUI 所使用的系统调用是 OS 的一部分。

操作系统实现了对计算机资源的扩展

没有任何软件支持的计算机被称为裸机，其仅构成计算机系统的物质基础。

也就是说，操作系统的内部是各种物理结构所组成的逻辑环境，操作系统所提供的资源管理功能和方便用户的各种服务功能，将裸机改造成功能更强、更方便的机器。通常，把覆盖了软件的机器称为扩充机器或虚拟机。

1.2 操作系统发展历程

1.2.1 手工操作系统

此阶段并未产生严格意义上的 *OS*。

手工操作系统的突出缺点：用户独占全机，资源利用效率极低；*CPU* 等待手工操作，*CPU* 利用效率极低。

1.2.2 批处理阶段

为了解决人机矛盾以及 *I/O* 设备之间速度不匹配的问题，出现了批处理系统。

单道批处理系统

系统对作业的处理是成批进行的，但内存中始终只保存一道作业。其主要特征为：

1) 自动性。磁带上的一批作业自动逐个进行，无需人工干预
2) 顺序性。磁带上的作业按顺序进入内存
3) 单道性。内存中仅有一道作业运行，即监督程序每次从磁带上只调入一道程序。

单道批处理系统的主要问题在于：内存中仅有一道作业，*CPU* 有大量的时间是在等待 *I/O* 的完成。

多道批处理系统

为了解决资源利用率和系统吞吐量，引入了多道程序技术。多道程序技术允许多个程序同时进入内存并允许在 *CPU* 中交替运行，共享系统中的各种软硬件资源。

其设计的特点为多道、宏观上并行，微观上串行。但是，多道程序设计需要解决：

1) 如何分配处理器
2) 如何分配内存
3) 如何分配 I/O 设备

4) 如何组织和存放大量的程序和数据，且保证数据安全和一致性

其优点在于：资源利用率高，且共享计算机资源从而使各种资源得到充分利用；系统吞吐量大，*CPU* 和其他资源保持“忙碌”状态。

缺点在于：用户响应时间较长；不提供人机交互能力，用户既不能了解运行情况也不能控制计算机。

分时操作系统

分时技术，也就是将处理器的运行时间分成很短的时间片，按照时间片轮流把处理器分配给各联机作业使用。这使得每个用户 *(程序)* 感觉起来就像自己独占一台计算机。

分时操作系统，指的是多个用户通过终端同时共享一台主机，用户可以同时与主机进行交互操作而互不干扰。因此，分时系统最关键的问题在于：如何使用户与自己的作业交互。分时系统支持多道程序设计，但不同于多道批处理，分时系统是人机交互的系统：

1) 同时性。同时性又称多路性，允许多个终端用户同时使用一台计算机。
2) 交互性。用户能够与系统进行人机对话。
3) 独立性。系统中的各个用户独立操作，互不干扰。
4) 及时性。用户请求能在很短时间内响应，采用时间片轮转算法。

实时操作系统

为了满足某个时间限制内完成某些紧急任务而不需要时间片排队，诞生了实时操作系统。

硬实时系统：某个动作必须绝对地在规定的时刻内完成或发生。

软实时系统：能够接收偶尔违反时间规定且不会引起任何永久性的损害。

在实时操作系统的控制下，计算机系统接收到外部信号后及时处理，并在严格时间内处理完毕，其主要特点就是及时性和可靠性。

1.3 操作系统运行环境

1.3.1 处理器运行环境

在 OS 中，CPU 通常执行两种不同性质[1]的程序：一种是 OS Kernel 程序；一种是 User program。对于 OS 来说，前者是后者的管理者，管理程序需要执行一些特权指令，而用户程序出于安全考虑不允执行。

[1]就我所知，现代 *OS* 中的等级应该大多数是三级，*M*、*S*、*U*，分别表示机器级、监管者、用户级。

同时，OS 中将程序的环境分成了两态，分别对应了特权级的内核态以及非特权级的用户态[1]，其用于用户程序和内核程序不同运行的环境。

1) 特权指令，指的是不允许用户直接使用的指令，比如 *I/O* 指令，置中断指令等一系列需要内核态环境下完成的指令。
2) 非特权指令，指的是允许用户直接使用的指令，不能直接访问系统中的软硬件资源，仅限于访问用户的地址空间。

需要注意的是：*OS 中的内核态和用户态是不断切换的，以适应不同的需求*。如果想要从内核态切换到用户态，那么直接使用对应的特权指令进行“降级操作”(例如在 rv 架构上的 sret 指令)。如果需要从用户态切换到内核态，那么必须通过中断操作 (此处的中断是广义的，包括了狭义的中断和异常)，触发中断信号，硬件自动完成变态操作。

内核是计算机上配置的底层软件，管理了系统的各种资源：

1. 时钟管理

   在计算机的各部件中，时钟是最为关键的设备。时钟的第一功能是计时，OS 通过时钟管理来获取标准的系统时间，且通过时钟中断实现进程的切换。

2. 中断机制

   引入中断的初衷是为了提高多道程序运行环境中 CPU 的利用率。就目前来说，现代操作系统就是靠中断驱动的软件。

   中断机制中，仅有一小部分属于内核，负责保护和恢复终端现场信息，转移控制权到相关的处理程序。

3. 原语

   操作系统提供的基本操作接口或函数，用于实现高级操作系统功能和管理底层资源。原语通常是操作系统内核的一部分，提供了对硬件和系统资源的底层访问和控制。

   1) 处于 OS 的最底层，是最接近硬件的部分
   2) 具有原子性，其操作只能一气呵成
   3) 运行时间短，调用频繁

   定义原语的直接方法是关闭中断，让其所有动作不可分割地完成后再打开中断。

4. 数据结构及处理

   在 OS 中，定义了很多登记状态的数据结构，如作业控制块、进程控制块 (PCB)、设备控制块、各种链表、队列等。常见的为三种：

   1) 进程管理。进程状态、调度与分派、创建、撤销等
   2) 内存管理。内存分配回收、保护、对换等
   3) 设备管理。缓冲区、设备分配、回收等

[1]在 OS 中，一般会存在一个特殊寄存器 PSW(程序状态字寄存器)，主要用于判断当前运行环境是内核态还是用户态。

1.3.2 中断和异常的概念

OS 引入内核态和用户态后，必须考虑的事情就是如何切换，如何留下“门”。

中断和异常的定义

中断 (Interruption) 也叫做外中断，指的是来自 *CPU* 执行指令外部的事件，通常用于信息输入/输出，例如 *I/O* 中断，时钟中断等。

异常 (Exception) 也叫做内中断，指的是 *CPU* 执行指令内部的事件，例如非法操作、越界、缺页或 *trap* 指令。值得注意的是，异常不能被屏蔽，一旦出现，需要立刻处理。

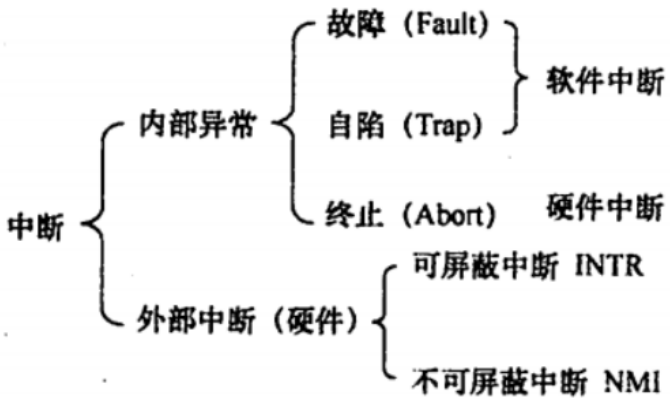


图 1.1: 中断和异常的联系与区别

中断和异常的分类

外中断可分为可屏蔽中断和不可屏蔽中断。可屏蔽中断通过 *INTR* 线发出中断请求，通过改变屏蔽字可以实现多重中断；不可屏蔽中断通过 *NMI* 线发出中断请求，一般比较紧急，因此不可屏蔽。

异常分为故障、自陷 (trap) 和终止。故障 (Fault) 通常由指令引起的异常，如非法操作码，缺页异常，除零等；自陷 (Trap) 是一种实现安排的“异常”行为，用户通过自陷操作从而进入内核态执行特权代码。终止 (Abort) 指出现了使 CPU 无法继续执行的硬件故障。

故障异常和自陷异常属于软中断，外部中断和终止异常属于硬中断。

中断和异常的处理

1. 当 CPU 运行时检测到异常事件 (或检测到一个中断请求信号)
2. 保存现场状态

3. CPU 打断当前执行，通过中断 (异常) 信号，查询中断向量表，跳转到对应的中断或异常处理程序
4. 若能处理，则返回执行，恢复现场; 若不能处理，终止程序

1.3.3 系统调用

用户在程序中调用 *OS* 所提供的一些子功能，系统调用可被视为特殊的公共子程序。

系统中的各种共享资源都是由 *OS* 统一管理，因此，*OS* 为了安全以及稳定起见，统一的为用户提供了接口，使得用户只要涉及资源相关的操作，就必须通过系统调用来提出服务请求，由 *OS* 来代理完成。

- 设备管理。完成设备的请求和释放，以及设备的启动等
- 文件管理。完成文件的读写、创建、删除等
- 进程管理。完成进程的创建、删除、阻塞等
- 进程通信。完成进程之间信息的传递等
- 内存管理。完成内存的分配、回收等

那么，对于上述的操作，其内部肯定是需要在内核态中运行一些特权指令的，因此，系统调用就需要用用户态切换到内核态去执行。因此，OS 提供了特殊的 trap 指令来发起系统调用请求。

也就是说，OS 的运行环境就可以为：用户通过 OS 运行上层应用，其依赖于 OS 的底层管理程序提供服务支持，当需要管理程序服务时，则通过中断 (异常) 机制进入内核态，运行管理程序。同时，在进入内核态时，需要保存现场。

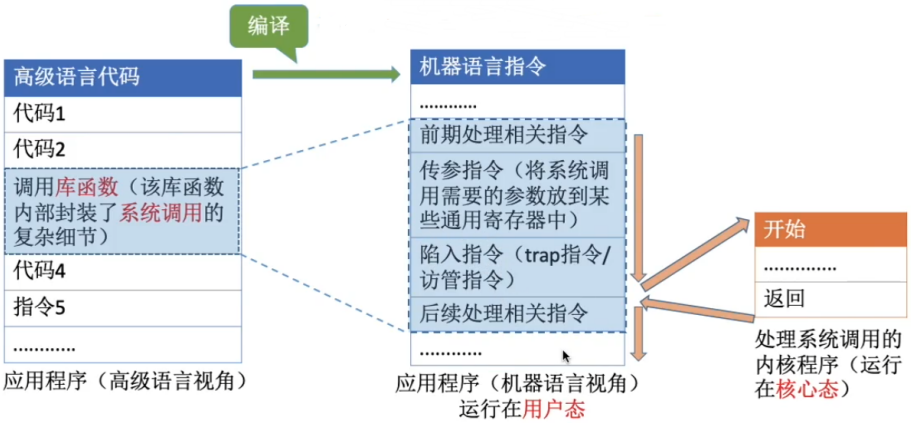


图 1.2: 系统调用的过程

值得注意的是：

1. 陷入指令是在用户态执行的，会立即产生一个软中断，使 *CPU* 进入内核态。

2. 发出的系统调用请求在用户态，执行在内核态 *(*通过传参指令，告诉系统调用入口程序要调用哪一个系统调用*)*。

1.4 操作系统结构

1.4.1 分层法

分层法是将操作系统分为若干层，最底层是硬件，最高层是用户接口，每层只能调用紧邻的低层的功能和服务 (单项依赖)。

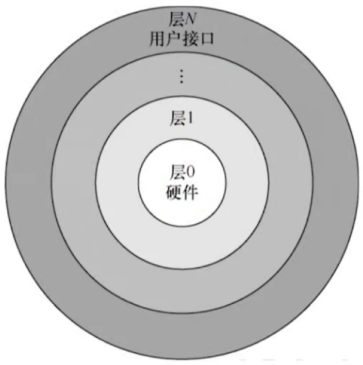


图 1.3: 分层法系统

分层法的优点：*1.* 便于系统的调试和验证，简化了系统的设计和实现；*2.* 易扩充和已维护，只需要逐层验证，且保证上下层接口即可轻易的扩充和修改。

分层法的缺点：*1.* 合理定义各层困难，需要考虑固定的依赖关系；*2.* 效率差，由于只能一层层的传递，因此每层都需要通信机制，而通信机制又会带来额外的开销，在分层法中尤为突出。

1.4.2 模块化

就目前来说，模块化是主流 *OS* 必不可少的机制。

模块化是将 *OS* 按功能划分为若干具有一定独立性的模块，每个模块具有某方面的功能，并规定好接口使之能互相通信。

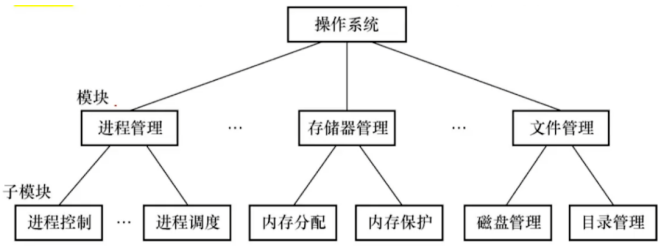


图 1.4: 模块化系统结构

模块化的划分是一件困难的事情，若划分太小，则各模块之间的耦合度过高；若太大，模块内部的复杂性过高。同时，还需要考虑到模块的独立性，越高，各模块之间的交互就越少，结构就越清晰。衡量模块的独立性主要为：

- 内聚性。模块内部各部分间联系的紧密程度，内聚性越高，独立性越好
- 耦合度。模块间相互联系和相互影响的程度，耦合度越低，独立性越好

模块化的优点：*1.* 提高了 *OS* 设计的正确性、可理解性和可维护性；*2.* 增强了 *OS* 的可适应性；*3.* 加速了 *OS* 的开发过程；*4.* 支持动态的增加模块功能

模块化的缺点：*1.* 模块间的接口规定难以满足对接口的实际需求；*2.* 各模块设计者共同开发，水平层次不齐，无法建立在上一个正确决定的基础上

1.4.3 宏内核

主流 *OS* 的选择。

宏内核，也称单内核或大内核，指的是将 *OS* 的主要功能模块都作为一个紧密联系的整体运行在内核态，从而为用户程序提供高性能的系统服务。

因为各模块间共享信息，能有效利用互相之间的有效特性，所以具有无可比拟的性能优势。

不过，就目前来说，例如 *Linux*，目前的瓶颈就在于宏内核使得代码量冗余，且无法有效的拆分 *(*哪怕是运用了模块化的机制*)*，导致内核功能复杂，难以维护。且内核中若某个重要模块出错，则会导致整个系统崩溃。

1.4.4 微内核

微内核架构，指的是将内核中最基本的功能保留在内核，其余功能移交到用户态执行，从而降低内核的设计复杂性。且同时本身具有模块化的机制，也就是将移出内核的模块按照分层的原则划分为若干服务程序，执行互相独立，交互借助于微内核通信。

微内核结构将 OS 分为了两块：微内核和多个服务器。微内核是精心设计的、实现 OS 最基本核心功能的小型内核，其只包括了：*1.* 与硬件相关的部分；*2.* 原语操作；*3.* 通信功能。其他类似于，进程管理、文件管理等则作为一个个服务模块与微内核进行通信。

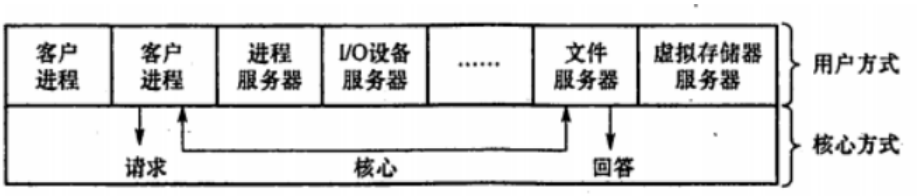


图 1.5: 单机模式下的微内核模式

为了提高可靠性，微内核架构中只有微内核运行在内核态，其他模块均在用户态，需要通信时，必须借助于微内核；同时，模块的错误只会导致模块崩溃，不会影响到微内核的运行。

微内核所提供的基本功能：

1) 进程 (线程) 管理。进程 (线程) 之间的通信是微内核最基本的功能，同时还有切换、调度等。对进程的分类，优先级则是属于策略，应当放入进程管理服务器中。
2) 低级存储器管理。微内核中只配置了类似于逻辑地址转换物理地址的页表机制和变换机制这种依赖硬件的功能。至于页表置换等策略，部署在存储器管理服务器中。
3) 中断和陷入处理。

微内核的特点：*1.* 扩展性和灵活性；*2.* 可靠性和安全性；*3.* 可移植性；*4.* 分布式计算

微内核的缺点：最致命的，也就是主流 *OS* 不选择微内核的问题就是，效率过低，由于模块间会不断的通信，因此会在用户态和内核态频繁切换，执行开销过大。

1.4.5 外核

OS 外核 (Exokernel) 是一种操作系统设计模式，它的目标是提供最小化的内核功能，将更多的控制权和灵活性交给应用程序。在外核模式下，内核的功能被精简为最基本的硬件抽象和资源分配，而应用程序可以直接访问和控制底层硬件资源。

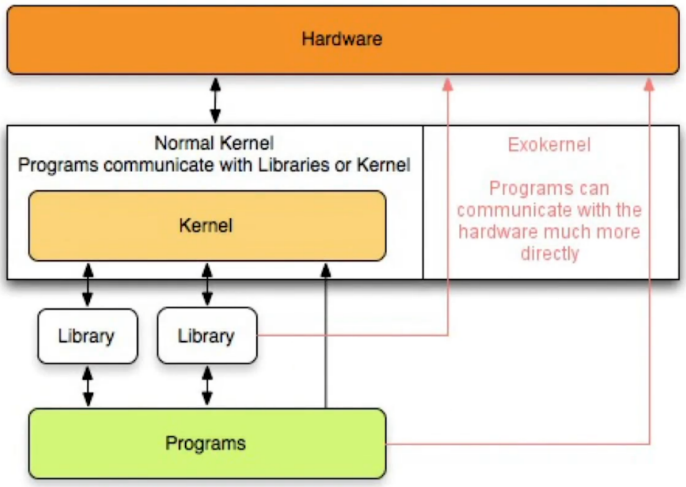


图 1.6: 外核结构

外核只提供了硬件资源的抽象核保护机制，直接将底层硬件资源暴露给应用程序，应用程序根据需求和策略进行资源分配、调度和优化。

外核模式的优势在于提供了更高的性能和灵活性。但是增加了开发和维护的复杂性。

1.5 操作系统引导

OS 是一种程序，程序以数据的形式存放在磁盘中，而硬盘分为多个区。OS 引导是指计算机利用 *CPU* 运行特定程序，通过程序识别硬盘，识别硬盘分区，识别 *OS*，最

后启动。

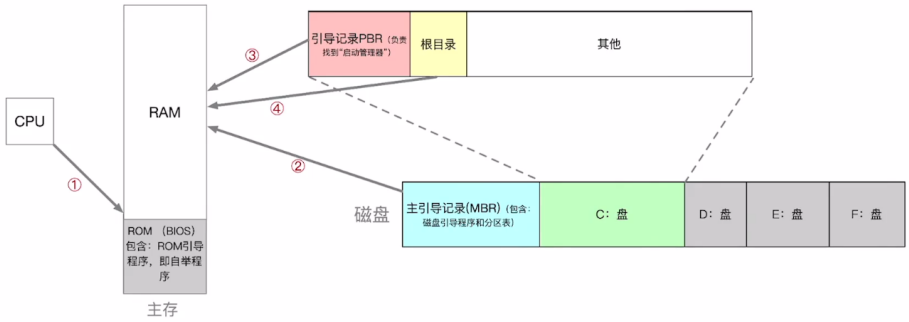


图 1.7: 操作系统引导过程

1) 激活 CPU。CPU 读取 ROM 中的 Boot 程序，将指令寄存器置为 BIOS 的第一条指令
2) 硬件自检。BIOS 首先会对硬件进行检查。
3) 加载带有 OS 的硬盘。BIOS 读取 BootSequence(CMOS 中的启动顺序，或用户交互)，把控制权交给启动顺序排第一位的存储设备，然后加载该扇区内容。
4) 加载主引导程序 MBR。硬件以特定的标识符区分引导硬盘和非引导硬盘。主引导 MBR 的作用是告诉 CPU 去硬盘的哪个分区找到 OS。
5) 扫描硬盘分区表，并加载硬盘活动分区 (含有 OS 的分区)。硬盘以特定的标识区分活动分区和非活动分区。若找到，移交控制权。
6) 加载分区引导记录 PBR。读取活动分区的第一个扇区，其作用是寻找并激活分区根目录下用于引导 OS 的程序 (启动管理器)
7) 加载启动管理器。
8) 加载 OS

1.6 虚拟机

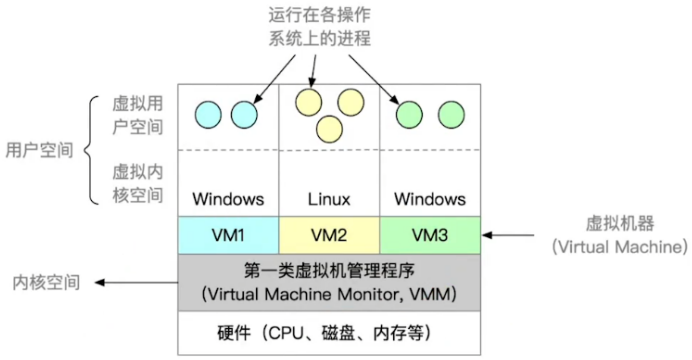
虚拟机是一台逻辑计算机，指利用特殊的虚拟化技术，通过隐藏特定计算平台的实际物理特性，为空户提供抽象的、统一的、模拟的计算环境。

1.6.1 第一类虚拟机管理程序

注意，第一类 *VMM* 和双系统是两个截然不同的东西。

从技术上来说，第一类 *VMM* 就像一个 *OS*，因为它是唯一一个运行在最高特权级的程序。*VMM* 向上提供若干虚拟机，这些虚拟机时逻辑硬件的精确复制品。

虚拟机作为用户态的一个进程运行，不允许执行特权指令 (因为虚拟机运行在用户态)。但是，由于虚拟机是一个完整的 *OS*，因此它认为自身运行在内核态，称为虚拟内核态。因此，虚拟机内核态所执行的特权指令，会被转化为对 *VMM* 的调用请求。



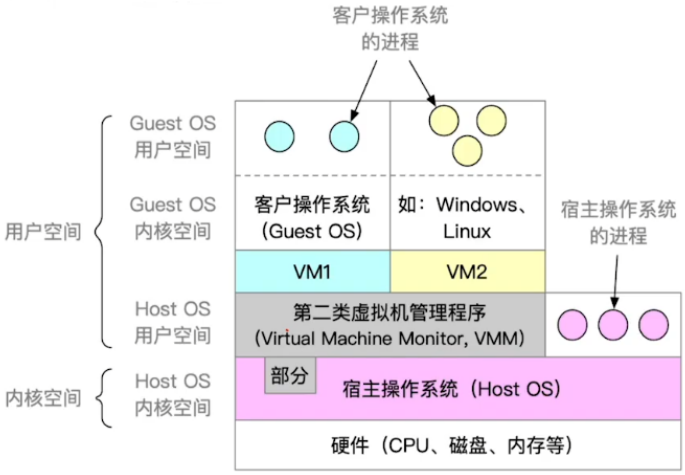
第一类VMM，直接运行在硬件上

图 1.8: 第一类 VMM

1.6.2 第二类虚拟机管理程序

第二类 VMM 就是我们所熟知的 VMware、VMBox 软件所虚拟的程序。

第二类 VMM 运行在 OS 上，就像一个普通的进程，同时，OS 为在 VMM 上运行的虚拟 OS 提供各种虚拟资源。



第二类VMM，运行在宿主操作系统上

图 1.9: 第二类 VMM

1.6.3 区别与联系

- 对资源的控制权

  第一类直接运行在硬件上，直接控制和分配；第二类运行在 Host OS 上，依赖于 Host OS 分配和管理

- 资源分配方式

  第一类安装 Guest OS 时，VMM 需要在原本的硬盘上自行分配存储空间；第二类拥有自己的虚拟磁盘，实际是 Host OS 文件系统上的一个大文件

- 性能

  第一类性能优于第二类

- 可支持的虚拟机数量

  第一类不需要和 Host OS 竞争资源，因此能够分配更多；第二类需要竞争，因此更少

- 可迁移性

  第二类的迁移性优于第一类

- 运行模式

  第一类 VMM 在最高优先级 (Ring0)；第二类 VMM 运行在用户态、部分运行在内核态 (VM 驱动程序)。Guest 发出的系统调用会被 VMM 截获，转化为 VMM 对 Host 的调用请求

第 2 章 进程与线程

2.1 进程与线程

2.1.1 进程的概念和特征

通常而言，在不同的认知角度上，对进程的定义是不同的，比较典型的定义为：

1) 进程是程序的一次执行过程
2) 进程是一个程序及其数据在处理及上顺序执行时所发生的活动
3) 进程是具有独立功能的程序在一个数据集合上运行的过程，是系统进行资源分配和调度的一个独立单位。

当然的，在 OS 中，进程经过了抽象从而定义了一个重要的数据结构：进程控制块 (Process Control Block，PCB)[1]。系统利用 PCB 来描述进程的基本情况和运行状态，进而控制和管理进程。相应地，由程序段、相关数据段和 *PCB* 三部分构成了进程实体 (进程映像)。

注意：*PCB* 是进程存在的唯一标识。

进程映像可以被视为进程在内存中的静态表示，描述了进程的初始状态和资源分配情况。虽然进程映像在进程运行期间保持不变，但进程本身可能会发生动态的变化。

注意：进程映像是静态的，进程则是动态的。

引入进程映像的概念后，又有了新的定义：进程是进程映像的运行过程，是系统进行资源分配和调度的一个独立单位。

进程是由多道程序的并发执行而引出的，其基本特征是对比单个程序的顺序执行提出的，也是对进程管理的基本要求：

1) 动态性。进程是程序的一次执行，动态性是进程最基本的特征。
2) 并发性。多个进程映像同存于内存中，能在一段时间内同时运行。并发性是进程的重要特征，也是 OS 的重要特征。
3) 独立性。进程映像是一个能独立运行、独立获得资源和独立接受调度的基本单位 (此处暂时不考虑线程)。
4) 异步性。由于进程的互相制约，使得进程各自独立、不可预知的向前推进。异步性会导致执行的不可再现性，因此必须配置同步机制。

[1] *Linux* 中为 *task_struct*，定义在 *sche.h* 文件中

2.1.2 进程的状态与转换

进程在其生命周期内，会不断地发生状态变化。通常进程有五种状态，前三种是基本状态：

1) 运行态 (Running)。进程正在 CPU 上运行。
2) 就绪态 (Runable)。进程获得了除处理机外的一切所需资源，通常处于就绪态的进程有多个，因此 *OS* 用就绪队列组织。
3) 阻塞态 (Blocking/Waiting)。进程正在等待某一事件而暂停，通常处于阻塞态的进程有多个，因此 *OS* 用阻塞 *(*等待*)* 队列来组织，甚至可以根据阻塞原因设置多个阻塞队列。
4) 创建态 (Creating)。进程正在被创建，未转到就绪态。
5) 终止态 (Exiting)。进程正在被释放或结束。

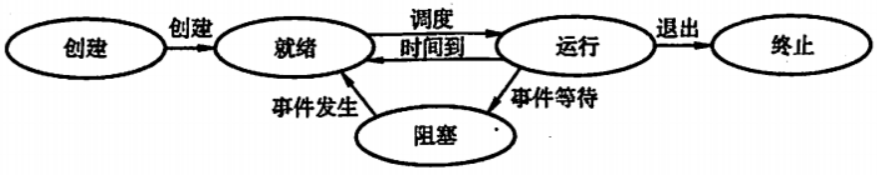


图 2.1: 五种状态的转换

值得注意的是：就绪态是指仅仅缺少处理机资源，也就是在就绪队列中还未到其处理的事件。而阻塞态是指，既缺乏处理机资源，又缺乏运行所需的必要资源 *(*或事件发生*)*。

一个进程从运行态变为阻塞态是主动的行为，从阻塞态变成就绪态是被动的行为。同时，就绪态不可能直接变为阻塞状态，因为变成阻塞必定是申请某种资源或事件发生，而这种情况之可能发生在运行时。

2.1.3 进程的组成

进程是一个独立的运行单位，也是 OS 进行资源分配和调度的基本单位。其由三个部分组成

进程控制块

进程创建时，需要新建一个 PCB 常驻于内存中，PCB 是进程存在的唯一标识，也是进程映射的一部分。

在整个生命周期中，系统总是通过 *PCB* 对进程进行控制的，即系统唯有通过进程的 *PCB* 才能感知到该进程的存在。

1) 进程描述信息。*PID* 是进程的唯一标识符。UID 标识了进程所属的用户
2) 进程控制和管理信息。

表 2.1: PCB 通常包含的内容

进程描述信息	进程控制和管理信息	资源分配清单	处理机相关信息
进程标识符 (PID)	进程当前状态	代码段指针	通用寄存器值
用户标识符 (UID)	进程优先级	数据段指针	地址寄存器值
	代码运行入口地址	堆栈指针	控制寄存器值
	程序外存地址	文件描述符	标志寄存器值
	进入内存时间	键盘	状态字
	处理机占用时间	鼠标	
	信号量使用		

进程当前状态：描述进程的状态信息，作为调度的凭据

进程优先级：用于抢占机制

3) 资源分配清单。用于说明内存地址空间或虚拟地址空间的状态，以及文件和 I/O 情况
4) 处理机相关信息。用于保存进程的信息，当进程被切换时，必须将上下文信息保存在此处 (线程就是为此的)。

在 OS 中，为了满足各种情况，通常有：就绪队列，等待队列以及优先级队列，这三种队列都会保存在 PCB 中以供使用。

特别的，一般而言还有另外一种组织方式：索引方式，将同一状态的进程组织在一张索引表内，然后通过索引表项指向 PCB。

程序段

程序段就是能够被进程调度程序调度到 *CPU* 执行的程序段代码。程序可能被多个进程共享。

代码段通常是只读的，意味着程序在运行时无法修改代码段中的指令。这是为了确保程序的逻辑和一致性。如果程序需要修改自身的指令，通常会使用特殊的技术，如自修改代码或动态代码生成。

在计算机的内存中，代码段通常位于程序的虚拟地址空间的一个固定位置。操作系统负责将代码段加载到内存中，并为程序提供执行的环境和资源。

数据段

一个进程的数据段，可以是进程对应的程序加工处理的原始数据，也可以是程序执行时产生的中间或最终结果。

进程的可变性，就主要体现在数据段中的变化。

2.1.4 进程控制

进程控制的主要功能是对系统中的所有进程实施有效的控制，具有创建、删除、转换等原语。

进程的创建

允许一个进程创建另一个进程。子进程可以继承父进程的所有资源。在 OS 中，系统调用的上层接口是 fork()，而底层的原语接口是 clone()。

在 OS 中，终端用户登录系统、作业调度、系统提供服务、用户程序的应用请求等都会引起进程的创建：

1) 为新进程分配唯一的一个 PID，并申请一个空白 PCB。
2) 为进程分配运行所需的各种资源。
3) 初始化 PCB，主要包括初始化标志信息、初始化 CPU 状态信息和初始化 CPU 控制信息，以及设置优先级
4) 若进程就绪队列能够容纳，则并入就绪队列，等待被调度

进程的终止

引起进程终止的事件主要有：正常结束；异常结束；外界干预这三种：

1) 根据被终止进程的标识符，检索出进程的 PCB，从中读取进程状态
2) 若被终止进程处于运行态，立即终止执行，并将处理机资源分配给其他任务
3) 若该进程有子进程，则还应将子进程终止
4) 将该进程的全部资源归还给父进程或 OS
5) 将该 PCB 从队列中移除

进程的阻塞和唤醒

正在运行的任务，由于期待的事件尚未发生，进程便通过调用阻塞原语，使自身从运行态转变为阻塞状态。可见，阻塞态是一种进程的自身主动行为，因此只能处于在运行态的进程才可能转换为阻塞态：

1) 找到将要被阻塞进程的标识号对应的 PCB
2) 若进程为运行态，则保护现场，将其转换为阻塞态，停止运行
3) 将该 PCB 插入到对应事件的等待队列，将处理机资源调度给其他就绪任务

当被阻塞进程所期待的事件发生时，由有关进程调用唤醒原语 (Wakeup):

1) 在该事件的等待队列中找到响应的 PCB
2) 将其从等待队列中移除，并置为就绪态
3) 把该 PCB 插入就绪队列，等待调度程序调度

注意：阻塞和唤醒原语是一对作用相反的原语，因此需要成对使用。

进程的切换

进程的切换一般由于：当前进程时间片到；更高优先级任务到达；当前进程主动阻塞；当前进程终止等，这时就需要切换原语：

1) 将运行环境信息存入 PCB
2) PCB 移入对应队列
3) 选择另一个任务执行，更新 PCB
4) 根据 PCB 恢复新进程所需要的环境

对于切换的原语，可以在内核中轻易的找到：

```
#define switch_to(prev,next,last) \
asm volatile(SAVE_CONTEXT                                    \
    "movq %%rsp,%P[threadrsp](%[prev])\n\t" /* save RSP */     \
    "movq %P[threadrsp](%[next]),%%rsp\n\t" /* restore RSP */  \
    "call __switch_to\n\t"                                   \
    ".globl thread_return\n"                                 \
    "thread_return:\n\t"                                     \
    "movq %%gs:%P[pda_pcurrent],%%rsi\n\t"                   \
    "movq %P[thread_info](%%rsi),%%r8\n\t"                   \
    "btr  %[tif_fork],%P[ti_flags](%%r8)\n\t"                \
    "movq %%rax,%%rdi\n\t"                                   \
    "jc   ret_from_fork\n\t"                                 \
    RESTORE_CONTEXT                                          \
    : "=a" (last)                                            \
    : [next] "S" (next), [prev] "D" (prev),                  \
    [threadrsp] "i" (offsetof(struct task_struct, thread.rsp)), \
    [ti_flags] "i" (offsetof(struct thread_info, flags)),\
    [tif_fork] "i" (TIF_FORK),                               \
    [thread_info] "i" (offsetof(struct task_struct, thread_info)), \
    [pda_pcurrent] "i" (offsetof(struct x8664_pda, pcurrent))   \
    : "memory", "cc" __EXTRA_CLOBBER)

struct task_struct *
__switch_to(struct task_struct *prev_p, struct task_struct *next_p)
```

2.1.5 进程的通信

进程通信是指进程间的信息交换。PV 操作是低级通信方式[1]，高级通信方式是指以较高的效率传输大量数据的通信方式。

共享存储

进程空间一般都是独立的，进程运行期间不能访问其他进程。这是共享存储的前提条件，在通信的进程之间存在一块可以直接访问的共享空间，通过对该空间进行读写操作，实现进程间的消息互换。

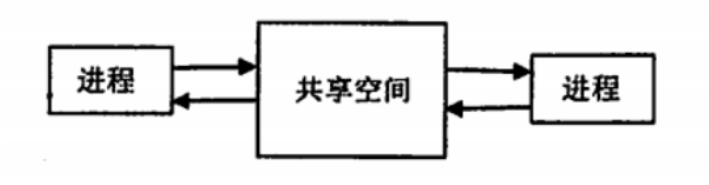


图 2.2: 共享存储

为了实现共享存储，一般 OS 会提供上层接口以供使用。例如 Linux 中，提供了 shm_* 等操作实现申请、撤销共享空间等操作，然后通过 mmap 将共享空间映射到进程自身的空间中。

此处给出简单的示例：

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    const char* shm_name = "/my_shared_memory";
    int shm_fd = shm_open(shm_name, O_RDWR | O_CREAT, 0666);
    if (shm_fd == -1) {
        perror("shm_open");
        exit(1);
    }

```

[1] *PV 操作是一种用于进程同步的经典算法，用于解决进程之间的互斥和同步问题。PV 操作通常与信号量 (Semaphore) 相关联。当进程需要访问共享资源时，执行 P(wait) 操作；当进程结束访问时，需要执行 V(signal) 操作。*

```
    off_t size = 4096; // 共享内存的大小
    if (ftruncate(shm_fd, size) == -1) {
        perror("ftruncate");
        exit(1);
    }

    void* addr = mmap(NULL, size, PROT_READ | PROT_WRITE,
MAP_SHARED, shm_fd, 0);
    if (addr == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }

    // 现在可以通过addr指针来访问共享内存中的数据

    // 访问完成后，记得使用munmap函数解除映射
    if (munmap(addr, size) == -1) {
        perror("munmap");
        exit(1);
    }

    // 关闭共享内存文件描述符
    if (close(shm_fd) == -1) {
        perror("close");
        exit(1);
    }

    // 删除共享内存对象
    if (shm_unlink(shm_name) == -1) {
        perror("shm_unlink");
        exit(1);
    }

    return 0;
}
```

对于共享存储而言，拥有两种方式：低级的共享是通过语言特性，例如全局的数据结构的共享，但是这样效率低、局限大；高级的共享就是通过这样实现一块存储区，

速度快，局限小。

消息传递

在消息传递系统中，进程间的数据交换以格式化的信息为单位。进程通过系统提供的发送/接收原语进行数据交换，这种方式隐藏了通信实现细节，使通信过程对用户透明，简化了通信程序的设计，是目前最广泛应用的机制。

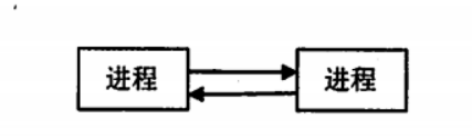


图 2.3: 消息传递

1) 直接通信方式。发送进程直接把消息发送给接收进程，并将其挂在接收进程的消息缓冲队列上，接收进程从中获取消息。
2) 间接通信方式。发送进程把消息发送到某个中间实体 (类似于电子邮件的机制，是有一个邮件服务器的缓冲区)，接收端从中间实体获取消息，该中间实体一般称为信箱。

管道通信

*在 Linux 中，管道是一种使用非常频繁的通信机制，本质上也是一种文件。*管道通信允许两个进程按生产-消费者模型进行通信，数据在管道中是 FIFO 的。管道的大小也是有规定的，一般为 4KB(也就是说，是一个固定的缓冲区)。

1. 管道只能采用半双工通信，某段时间内只能单向通信；但可以使用两个管道实现全双工通信。

2. 各进程需要互斥的读写管道。当管道写满时，写进程阻塞；当管道读空时，读进程堵塞。

3. 数据一旦被读取，一般情况下会消失。因此会出现争议：a). 一个管道允许多个写进程，一个读进程[1]*。b). 允许多个写进程，多个读进程，系统会让进程各自轮流读取*[2]*。*

一个简单的示例，实现进程间互相通信：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

```

[1]如果是答题，按照参考答案上的结果就是这样。如果从理解上来看，这种和后面一种的方式都是正确的。

[2]*Linux* 中的策略

```
int main() {
    int pipe1[2]; // 管道1，用于父进程向子进程发送数据
    int pipe2[2]; // 管道2，用于子进程向父进程发送数据

    if (pipe(pipe1) == -1 || pipe(pipe2) == -1) {
        perror("pipe");
        exit(1);
    }

    pid_t pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(1);
    }

    if (pid == 0) {
        // 子进程
        close(pipe1[1]); // 关闭管道1的写端
        close(pipe2[0]); // 关闭管道2的读端

        char message[100];
        read(pipe1[0], message, sizeof(message)); // 从管道1读取数据
        printf("子进程收到消息：%s\n", message);

        const char* reply = "Hello，父进程!";
        write(pipe2[1], reply, strlen(reply) + 1); // 向管道2写入数据

        close(pipe1[0]); // 关闭管道1的读端
        close(pipe2[1]); // 关闭管道2的写端
    } else {
        // 父进程
        close(pipe1[0]); // 关闭管道1的读端
        close(pipe2[1]); // 关闭管道2的写端

        const char* message = "Hello，子进程!";
        write(pipe1[1], message, strlen(message) + 1); // 向管道1写入数据
```

```
42          char reply[100];
            read(pipe2[0], reply, sizeof(reply)); // 从管道2读取数据
44          printf("父进程收到消息：%s\n", reply);

46          close(pipe1[1]); // 关闭管道1的写端
            close(pipe2[0]); // 关闭管道2的读端
48      }

50      return 0;
}
```

2.1.6 线程和多线程模型

线程的基本概念

引入线程的目的是减少程序在并发执行时所付出的时空开销，提高 *OS* 的并发性能。

线程最为直接的理解就是“轻量级进程”，是一个基本的 *CPU* 执行单元，也是程序执行流的最小单元。线程是进程的一个实体，是被 *OS* 独立调度和分派的基本单位*(*引入线程后，进程就不再是调度的基本单位了*)*，线程不拥有系统资源，但可以与同属进程的其他线程共享所拥有的全部资源。

那么，现在重新定义：进程是只作为除 *CPU* 外的系统资源的分配单元，线程作为处理机的分配单元。

线程与进程的比较

1) 调度

引入线程后，线程是独立调度的基本单位，线程切换的代价远小于进程，同时，在同一进程中的不同线程切换，不需要切换进程。

2) 并发性

不仅进程能够并发，线程也是能够并发的，准确来说，线程就是为此而生的

3) 拥有资源

进程是系统中拥有资源的基本单位，线程不拥有系统资源，但能够访问所属进程的资源 *(*主要表现在，同一进程中的所有线程具有相同的地址空间*)*

4) 独立性

每个进程都拥有独立的地址空间和资源，除了共享的全局变量；而同一进程的所属线程间，共享进程的地址空间和资源。

5) 系统开销

显然，线程的各种开销明显小于进程

6) 支持多处理机系统

线程的状态和转换

与进程类似，各线程间因为并发的原因也存在共享资源和相互合作的制约关系：

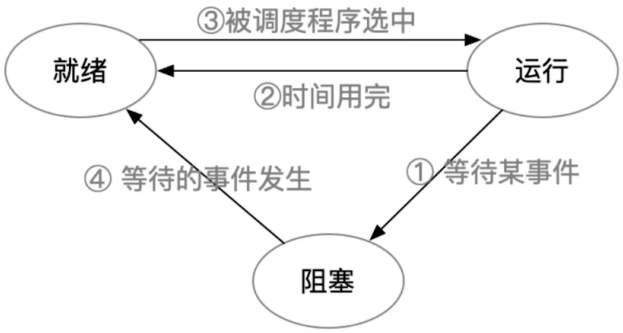


图 2.4: 线程的状态与转换

线程的组织与控制

与进程类似，OS 也为线程配置了一个线程控制块 TCB，用于记录控制和管理线程的信息。

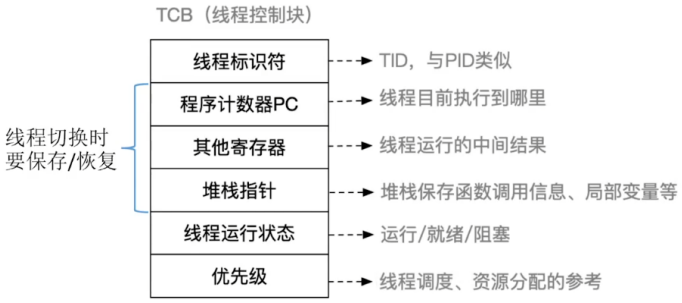


图 2.5: TCB

同一进程中所有线程都完全共享进程的空间与资源，一个线程能够读写甚至删除另一个线程的堆栈。同时，线程的创建和终止也和进程类似，不过在删除时，必须考虑进程的因素，从而选择是否释放资源。

线程的实现方式

线程的实现可以分为两类：用户级线程 (User-Level Thread，ULT) 和内核级线程 (Kernel-Level Thread)

用户级线程　在用户级线程中，有关线程管理的所有工作都是由应用程序在用户态完成的，内核根本意识不到线程的存在。因此，对于设置了 ULT 的系统，其调度实际上还是以进程为单位的。

这种方式的优点在于：

1. 线程切换不需要进入到内核态，节省了切换的开销。

2. 调度算法是进程专用的，不同进程可根据自身需求，为自己的线程选择不同的调度算法。

3. 用户级线程的实现与 *OS* 无关，对线程管理的代码属于用户程序的一部分

而缺点在于：

1. 系统调用的阻塞不仅会导致该进程被阻塞，且进程内的所有线程都会被阻塞

2. 严重降低了处理效率，不能发挥多处理机的优势

内核级线程　内核级线程是由内核支持的，线程管理的工作在内核空间中实现。因此内核为每个线程设置了 TCB，内核能够对内核级线程进行感知。

这种方式的优点在于：

1. 发挥了多处理机的优势，能够同时调度同一进程中的多个线程

2. 如果进程中的一个线程阻塞，那么可以对另外一个线程进行调度

3. 内核支持线程具有小的数据结构和堆栈，切换较快，开销较小

4. 内核本身也提供多线程技术，提高系统的执行效率

而缺点在于：

同一进程中的线程切换，需要转换到内核态，系统开销较大。

ULT & KLT　ULT 和 KLT 的组合能够使得结合各自的优点并克服各自的不足。支持多个内核级线程的建立、调度和管理，同时允许用户级线程的使用，能够使得线程阻塞但不会导致其他线程阻塞。

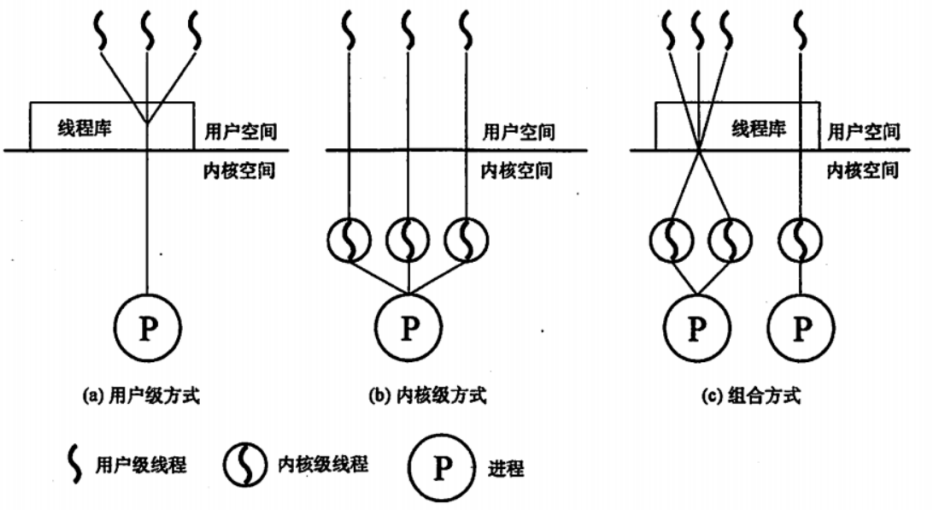


图 2.6: 用户级线程和内核级线程

多线程模型

基于有些系统同时支持用户线程和内核线程，因此就会产生不同的多线程模型：

多对一模型 将多个用户级线程映射到一个内核级线程 (一般来说，多对一就是特指一个内核级线程)，这些用户级线程通常属于一个进程。

优点：线程管理在用户空间进行，效率较高

缺点：一个线程发生阻塞，那么整个进程都会阻塞；且任何时刻只能由一个线程访问内核

一对一模型 每个用户级线程映射到一个内核级线程

优点：当一个线程被阻塞后，允许调度另一个线程，并发能力较强

缺点：每创建一个用户线程，都需要一个内核线程，开销过大

多对多模型 将 n 个用户级线程映射到 m 个内核级线程，要求 $n \geq m$

特点：既克服了多对一模型并发度不高的缺点，又克服了一对一模型开销过大的问题。

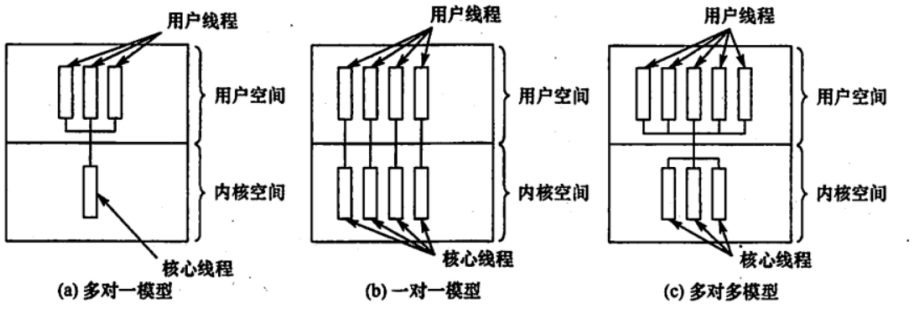


图 2.7: 多线程模型

2.2 处理机调度

2.2.1 调度的概念

调度的基本概念

处理机调度是对处理机进行分配，即从就绪队列中按照一定的算法选择一个进程并将处理机分配给它允许，以实现进程的并发执行

调度的层次

一个作业 (也就是一个程序) 从提交到完成，往往经历以下三级调度：

1) 高级调度 (作业调度)

按照一定的原则从外存中处于后备队列中的作业中挑选一个 (或多个)，给他们分配内存、输入/输出等必要资源，并建立对应的 *PCB*。

作业调度是内存与辅存之间的调度，每个作业只调入一次、调出一次。

2) 中级调度 (内存调度)

中级调度的目的是为了提高内存利用率和系统吞吐量。可以将暂时不能运行的进程调至外存等待，此时进程的状态为挂起态[1]

当其具备运行条件且内存有足够的空闲时，修改其状态为就绪态，挂在就绪队列中等待。中级调度实际上是存储器管理中的对换功能 *(swap)*。

3) 低级调度 (进程调度)

按照某种算法从就绪队列中选取一个进程，将处理机分配给它。调度算法是最基本的一种调度，在各种 *OS* 中必须配置该调度。

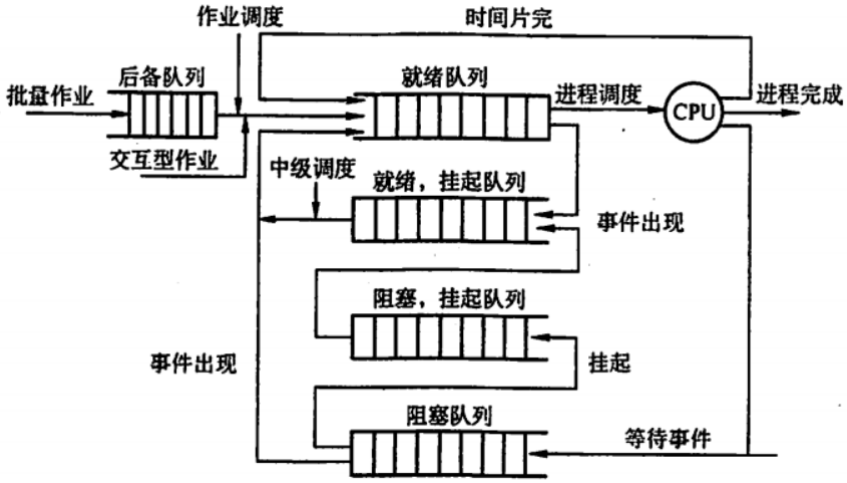


图 2.8: 处理机的三级调度

三级调度的联系

作业调度从外存中的后备队列选择一批作业进入内存，且创建对应的 PCB，然后送入就绪队列中，进程调度从就绪队列选择一个进程，将其状态改为运行态，并分配处理及资源。中级调度为了提高内存利用率。

1) 作业调度为进程活动做准备，进程调度使进程正常活动
2) 作业调度次数少，中级调度略多，进程调度频率最高
3) 进程调度是最基本的，不可或缺

2.2.2 调度的目标

1) CPU 利用率

[1]挂起态可以分为就绪挂起和阻塞挂起，这就是七状态模型。

CPU 是 OS 中最重要和昂贵的资源之一，因此应该尽可能使 CPU 保持“忙”状态：

$$CPU_{Utilization} = \frac{CPU_{efficient}}{CPU_{efficient} + CPU_{free}}$$

2) 系统吞吐量

表示单位时间内 CPU 完成作业的数量。

3) 周转时间

指从作业提交到作业完成所经历的时间，是作业等待、在就绪队列中排队、运行和输入/输出操作所花费时间的总和：

$$Turnaround = Work_{done} - Work_{submit}$$

平均周转时间是指多个作业周转时间的平均值

$$Avr_{Turnaround} = \frac{Turnaround_1 + Turnaround_2 + \cdots + Turnaround_n}{n}$$

带权周转时间是指作业周转时间与作业实际运行时间的比值

$$P_{Turnaround} = \frac{Turnaround}{Real_{time}}$$

平均带权周转时间

$$Avr_{PT} = \frac{P_{Turnaround}1 + \cdots + P_{Turnaround}n}{n}$$

4) 等待时间

指进程处于等待处理机的时间之和，等待时间越长，用户满意度越低。处理机调度算法实际上不影响作业执行或输入/输出的时间，只影响作业在就绪队列中等待所花的时间。[1]

5) 响应时间

指从用户提交到系统首次执行或输入/输出操作的时间。在交互式系统中，周转时间往往不是最好的评价准则，一般采用响应时间作为衡量调度算法的重要准则之一。

2.2.3 调度的实现

调度程序 (调度器)

用于调度和分派 CPU 的组件称为调度程序，通常由三部分组成：

1) 排队器。将系统中的所有就绪进程按照一定的策略排成一个或多个队列。

[1]值得注意的是，等待 *IO* 并不算在等待时间，因为此时是正在被服务的。

2) 分派器。依据调度程序中所选的进程，将其从就绪队列中取出，将 CPU 分配给新进程。
3) 上下文切换器。在调度的时候，会发生切换，因此需要保存各自进程中的上下文数据。

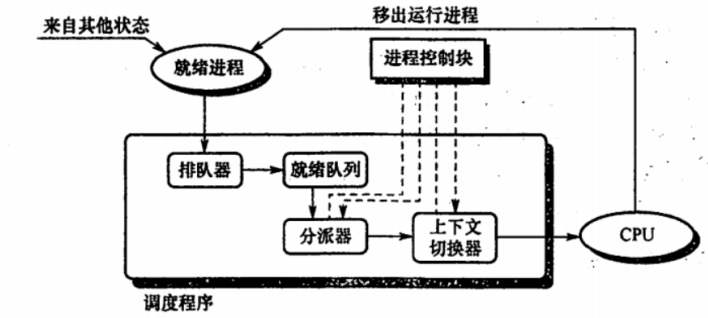


图 2.9: 调度程序的结构

闲逛进程

在进程切换时，如果系统中没有就绪进程，就会调度闲逛进程 (idle) 运行。闲逛进程的优先级最低，只有没有任何进程时，才会运行闲逛进程。

闲逛进程不需要 *CPU* 之外的资源，其不会被堵塞。

2.2.4 调度算法

先来先服务 (FCFS) 调度算法

FCFS 算法是一种最简单的调度算法，即可用于作业调度，也可用于进程调度。其主要从“公平”的角度考虑，按照作业/进程到达的先后顺序进行服务。

同时，FCFS 算法是非抢占式算法，其优点在于简单、公平；缺点在于对短作业及其不友好。不会导致饥饿的情况。

表 2.2: FCFS 调度算法的性能

作业号	提交时间	运行时间	开始时间	等待时间	完成时间	周转时间	带权周转时间
1	8	2	8	0	10	2	1
2	8.4	1	10	1.6	11	2.6	2.6
3	8.8	0.5	11	2.5	11.5	2.7	5.4
4	8.9	0.2	11.5	2.5	11.7	2.7	13.5

短作业优先 (SJF) 调度算法

短作业优先算法是指对短作业优先调度的算法，是非抢占式的，一般而言，非抢占式的短作业调度算法是指短进程优先调度算法 *(SPF)*，不过通常不会特别区分。

SJF 算法的优点在于，能够使得平均等待/周转/带权周转时间均比 FCFS 要好，但是 *SJF* 对长作业不友好，容易导致长作业发生饥饿甚至饿死的情况。

同时注意：*SJF* 算法并非是平均等待时间、平均周转时间最少的算法 *(*除非特别指明是 *SRNT* 算法*)*。

最短剩余时间优先 (SRTN) 调度算法

SRTN 算法是基于 SJF 的调度算法，不过 SRTN 是可抢占的，每当加入一个新的进程，就绪队列就需要判断是否发生调度，如果新的进程运行时间比当前进程的剩余运行时间短，那么就发生调度。

因此，*SRTN* 算法是平均等待时间、平均周转时间最少的算法 *(*相较于这几种算法*)*。

同时，可以给出另一种定义：在所有进程几乎同时到达时，采用 *SJF* 算法的平均等待时间、平均周转时间最少。

并且：由于 *SJF* 算法的作业的时间长短由用户提供的估计执行时间决定，因此可能导致用户无意间缩短其运行时间，因此*SJF* 不一定能够实现真正意义上的短作业优先。

高响应比优先 (HRRN) 调度算法

HRRN 算法主要用于作业调度，是对 FCFS 和 SJF 算法的一种综合平衡，同时考虑了作业的等待时间和运行时间。

在每次调度时，先计算每个作业的响应比，选择响应比高的作业进行调度：

$$Response_{R_p} = \frac{Time_{wait} + Time_{Service}}{Time_{Service}}$$

是一种非抢占式调度算法，*1.* 作业的等待时间相同时，要求服务时间越短，响应比越高；*2.* 要求服务时间相同时，响应比取决于等待时间；*3.* 对于长作业，响应比可以随等待时间增加而提高，因此能够克服饥饿现象。

上面的几种调度算法，主要用于早期的批处理 *OS*，因此几乎都不关心响应时间，而更在乎整体性能。

时间片轮转 (RR) 调度算法

时间片轮转调度算法主要用于分时系统，更注重响应时间，忽略周转时间。轮流让就绪队列中的进程依次执行一个时间片。

注意：如果一个时间片用完时，刚好另一个任务被提交上来，那么新加入的任务优先进入队列。

时间片轮转的好处是显而易见的，但是，如果时间片设置过大，那么就会退化为 *FCFS* 调度算法，并且增大进程响应时间；如果时间片过小，又会导致进程切换过于频繁，使得资源开销过大。

一般来说，设计时间片时要让切换进程的开销占比不超过 *1%*。

优先级调度算法

优先级调度算法既可用于作业调度，又可用于进程调度。主要描述一个作业的紧急程度。

优先级算法每次从后备队列中选取优先级最高的任务进行调度，可以分为抢占式和非抢占式：

1) 非抢占式优先级调度。每次选择已到达任务中优先级最高的任务进行调度
2) 抢占式优先级调度。根据每次到达任务中优先级最高的任务进行调度

对于优先级来说，进程可以设置两种不同的优先级：

1) 静态优先级。创建进程时就已经确定，且保持运行期间不会发生改变。
2) 动态优先级。根据进程情况的变化动态调整优先级。
    a) 系统进程 > 用户进程。系统的优先级理应高于用户
    b) 交互型进程 > 非交互型进程。需要交互的进程需要更快的响应速度。
    c) I/O 型进程 > 计算型进程。I/O 设备一般比 CPU 运行的更慢，能够更快进行 I/O 设备的调度就能够使得等待时间和响应更快，提升 OS 的整体效率

多级队列调度算法

在多处理机中，上述都是单一的调度策略，多级队列设置多个就绪队列，将不同类型或性质的进程固定分配到不同的就绪队列。

多级反馈队列调度算法

多级反馈队列调度算法是时间片轮转调度算法和优先级算法的综合发展，通过动态调整进程优先级和时间片大小，能够兼顾多方面的系统目标。

各级队列优先级从高到低、时间片从小到大；新进程进入按照 *FCFS* 算法进入最高优先级队列，若在一个时间片内无法结束，那么降级到第二级优先级。只有当上级队列为空时，才能够调度下一级队列中的任务。

2.2.5 进程切换

狭义的进程调度指的是将一个任务从就绪队列中调度上 *CPU* 的动作；而进程切换通常指的是一个任务让出处理机，另一个任务占用，会涉及到上下文切换。对于广

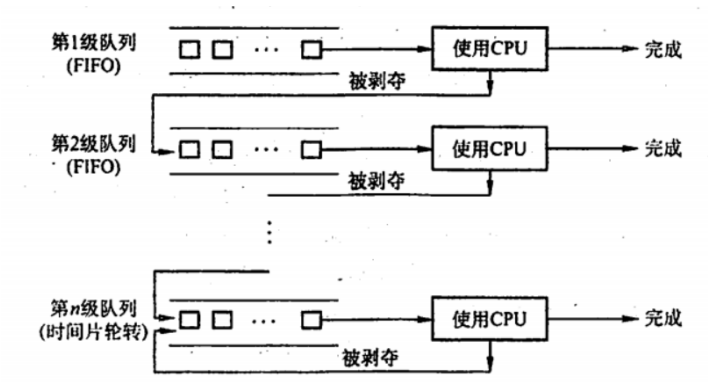


图 2.10: 多级反馈队列调度算法

义的进程调度，则是都包含。

上下文切换

从 CPU 上切换一个任务需要保存当前任务状态，并恢复另一个任务的状态，同时需要恢复其环境，这个过程叫做上下文切换。上下文指的是某一时刻 *CPU* 寄存器和程序计数器中的内容。

1) 挂起一个进程，保存 CPU 上下文
2) 更新 PCB 信息
3) 把进程的 PCB 移入相应的队列
4) 选择另一个进程执行，更新其 PCB
5) 跳转到新进程 PCB 中的程序计数器所指向的位置执行
6) 恢复处理机上下文

上下文切换与模式切换

通常而言，模式切换指的是用户态与内核态之间的状态切换，其不会改变当前进程。上下文切换只会发生在内核态，是多任务 *OS* 中的一个必需的属性。

2.3 同步与互斥

2.3.1 同步与互斥的基本概念

临界资源

在 *OS* 中，多个进程可以共享各种资源，但其中许多资源一次只能为一个进程使用，因此将一次仅能一个进程使用的资源称为临界资源。

对临界资源的访问必须互斥的进行，每个进程中，访问临界资源的那段代码称为临界区：

1) 进入区。检查进程是否能够使用临界资源，且阻止其他进程同时进入临界区。
2) 临界区。进程中访问临界资源的代码段
3) 退出区。修改临界标志
4) 剩余区。代码中其余部分

同步

同步可以称为直接制约关系，指为完成某种任务而建立的两个或多个进程，因为需要在某些位置上协调工作次序而等待、传递信息所产生的制约关系。

互斥

互斥可以称为间接制约关系，当一个进程进入临界区后，另一个进程必须等待其完成才能访问。

为禁止两个进程同时进入临界区，同步机制应遵循：

1) 空闲让进。临界区空闲时，可以允许一个请求进入临界区的进程立即进入
2) 忙则等待。当已有进程进入临界区时，其他试图进入临界区的进程必须等待
3) 有限等待。对请求访问的进程，应保证能在有限时间内进入临界区
4) 让权等待。当进程不能进入临界区时，应立即释放处理器，防止进程忙等待

2.3.2 实现临界区互斥的基本方法

软件实现方法

单标志法　通过设置一个公用整型变量，用于指示被允许进入临界区的进程编号。该算法良好的保证了一次只允许一个进程进入，但两个进程必须交替进入，若某个进程不再进入，另一个进程则也无法进入：违背空闲让进原则。

```
P0:                                P1:
    while (turn != 0) ;                while (turn != 1) ;
    critical section ;                 critical section ;
    turn = 1 ;                         turn = 0 ;
    remainder section ;                remainder section ;
```

双标志法先检查　通过在进程访问临界区资源之前，先检查临界区资源是否被访问，若正在被访问，则等待。该算法避免了交替进入，但是由于两个进程并发可能“同时”进入临界区导致违背忙则等待原则。

```
P0:                                P1:
```

```
    while (flag[0]) ;                while (flag[1]) ;
3   flag[1] = true ;                 flag[0] = true ;
    critical section ;               critical section ;
5   flag[0] = false ;                flag[1] = false ;
    remainder section ;              remainder section ;
```

双标志法后检查 为了防止同时进入临界区，可以先标识自身，然后检查对方的意愿。避免了同时进入临界区，但导致了更严重的问题，二者都无法进入临界区使得饥饿现象的出现。

```
    P0:                              P1:
2   flag[0] = true ;                 flag[1] = true ;
    while (flag[1]) ;                while (flag[0]) ;
4   critical section ;               critical section ;
    flag[0] = false ;                flag[1] = false ;
6   remainder section ;              remainder section ;
```

Peterson's Algorithm 为了防止进程进入临界区无限期等待，设置一个类似于单标志法的共有变量，主体是双标志法和单标志法的综合。

```
    P0:                              P1:
2   flag[0] = true ;                 flag[1] = true ;
    turn = 1;                        turn = 0;
4   while (flag[1] && turn == 1) ;   while (flag[0] && turn == 0) ;
    critical section ;               critical section ;
6   flag[0] = false ;                flag[1] = false ;
    remainder section ;              remainder section ;
```

硬件实现方法

计算机硬件提供了特殊的指令，允许对一个字中的内容进行检测和修整，或进行交换等，通过硬件支持实现临界段问题的方法称为低级方法，或元方法。

中断屏蔽方法 一个进程正在执行临界区代码时，防止其他进程进入的最简方法是关中断。但是，这种方法限制了处理机交替执行程序的能力，且中断指令是内核态的特权指令。

硬件指令方法

TestAndSet 指令　又称 TS 或 TSL 指令。该指令是原子操作，即执行该代码时不允许被中断。其功能是读出指定标志后把该标志设置为真：

```
bool TestAndSet (bool* lock) {
    bool old = *lock;
    *lock = true;
    return old;
}

while (TestAndSet(&lock)) ;
...
lock = false;
...
```

Swap 指令　又称 Exchange 或 XCHG 指令。

```
void Swap (bool* a, bool* b) {
    bool temp = *a;
    *a = *b;
    *b = temp;
}
```

值得注意的是，上述的 C 代码仅仅是对其功能的描述，真实的设计是在硬件上直接完成的。

硬件方法的优点在于：适用于任意数目的进程，简单、容易检验其正确性，支持多个临界区。

其缺点在于：不能实现让权等待原则，可能导致饥饿现象。

2.3.3 互斥锁

解决临界区最简单的工具就是互斥锁 *(mutex lock)*。值得注意的是，互斥锁的申请和释放都必须是原子操作。互斥锁的主要缺点是忙等待，因为在进入临界区判断时，会一直自旋。

需要连续循环忙等的互斥锁，可以叫做自旋锁 *(spin lock)*。

```
acquire() {
```

```
    while (!available) ;
    available = false;
}

release() {
    available = true;
}
```

2.3.4 信号量

信号量机制时一种功能较强的机制，用于解决互斥同步问题，由 *OS* 提供的一对原语：*wait* 和 *signal* 操作。

一般来说，这对原语也被称为 *PV* 操作，因此 *P(S)* 和 *V(S)* 分别代表了 *wait(S)* 和 *signal(S)*。注意：在做题时，*P(S)*、*V(S)* 默认代表记录型信号量。

整型信号量

整型信号量被定义为一个用于表示资源数目的整型量 S。

```
wait(S) {
    while (S <= 0) ;
    S = S - 1;
}

signal(S) {
    S = S + 1
}
```

可以看见，对于整型信号量未能解决忙等问题。

记录型信号量

记录型信号量是一种不存在忙等现象的进程同步机制。其内部不仅仅由一个代表资源数目的变量组成，还由一个进程链表，用于链接所有等待该资源的进程。

```
typedef struct {
    int value;
    struct task_struct* task;
} semaphore;
```

```
void wait(semaphore S) {
    if (--S.value < 0) {
        add this task to S.task;
        block(S.task);
    }
}
void signal(semaphore S) {
    if (++S.value <= 0) {
        remove a task T from S.task;
        wakeup(T);
    }
}
```

对于记录型信号量来说，wait 操作统计资源数，如果资源耗尽，那么就需要将当前请求资源的进程进行阻塞等待资源，这样就使得不会忙等，让出了处理机资源。当一个拥有资源的进程用完后，进入 signal，对资源进行复原，如果资源仍旧小于等于 0，那么就说明，有进程在申请该资源，则将申请资源的进程从阻塞队列中唤醒即可。

利用信号量实现同步

```
semaphore S = 0;
P1() {                  P2() {
    stmt1...                P(S);
    V(S);                   stmt1...
    ...                     ...
}                       }
```

上述实现的意义为：P2() 中的某一些操作必须在 P1() 中的某些操作执行完之后执行。通过 V(S) 对 P2 进行唤醒。

利用信号量实现互斥

```
semaphore S = 1;
P1() {                  P2() {
    P(S);                   P(S);
    stmt1...                stmt1...
    V(S);                   V(S);
```

```
6       ...                     ...
}                       }
```

利用信号量实现前驱关系

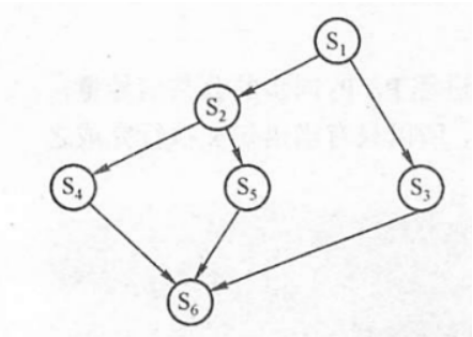


图 2.11: 利用信号量实现前驱关系

```
1  semaphore s1_1 = s1_2 = s2_1 = s2_2 = s3 = s4 = s5 = 0;
   S1() {          |  S2() {          |  S3() {
3      ...         |      P(s1_1);    |      P(s1_2);
       V(s1_1);    |      ...         |      ...
5      V(s1_2);    |      V(s2_1);    |      V(s3);
   }               |      V(s2_2);    |  }
7                  |  }               |
   ------------------------------------------------
9  S4() {          |  S5() {          |  S6() {
       P(s2_1);    |      P(s2_2);    |      P(s4);
11     ...         |      ...         |      P(s5);
       V(s4);      |      V(s_5);     |      P(s6);
13 }               |  }               |      ...
                   |                  |  }
```

可以看见，其实依赖关系通过 P(S) 来阻塞，依赖关系一旦满足，则使用 V(S) 来解除依赖即可。因此，做题时，对于同步或多级同步来说，只需要 *V* 前 *P* 后即可满足。

2.3.5 管程

在信号量机制中，每个要访问临界资源的进程都必须自备同步 PV 操作。因此，一种新的进程同步工具————管程产生了，管程保证了进程互斥，且无需程序员自行实现互斥，降低了死锁的可能性。

管程的定义

系统中的各种硬件资源和软件资源，都可以用数据结构抽象地描述其资源，即用少量信息和对资源所执行的操作来表征该资源，忽略内部结构和实现细节。

因此，如果用 OOP 的思想来看待，管程就是一个用户自描述的一个类，用共享数据结构抽象表示共享资源，然后通过类方法对共享资源进行操作。

管程 *(monitor)*，就是代表共享资源的数据结构，以及对该数据结构和实时操作的一组过程所组成的资源管理程序。

管程的基本特征为：

1) 局部于管程的数据只能被局部于管程的过程所访问

   也就是说，管程中的共享数据，只允许通过管程中定义的方法操作

2) 一个进程只有通过调用管城内的过程才能进入管程访问共享数据

   和第一点所表达的意思是一致的

3) 每次仅允许一个进程在管程中执行某个内部过程

   也就是说，管程是互斥的，管程中的操作，只能由一个进程访问，不允许多个进程同时执行。

如图 2.12 所示，在 C++ 中利用互斥锁和条件变量自定义了一个管程 (monitor) 类，其中，buffer 就是我们的缓冲区资源，而 max_size 就是规定的最大数量。

而 produce 和 consume 函数，就是对管程的操作过程，其中，里面的 wait 函数，用于条件变量的作用。

```cpp
class monitor {
private:
    const int max_size = 5;

    std::mutex mtx;
    std::condition_variable cond;
    std::queue<int> buffer;
public:
    auto produce(int value) -> void {
        std::unique_lock<std::mutex> lock( &: mtx);
        cond.wait( &: lock,  p: [this] -> bool { return buffer.size() < max_size; });
        buffer.push( x: value);
        std::cout << "Produced: " << value << std::endl;
        cond.notify_one();
    }
    auto consume() -> decltype(auto) -> int {
        std::unique_lock<std::mutex> lock( &: mtx);
        cond.wait( &: lock,  p: [this] -> bool { return !buffer.empty(); });
        int value = buffer.front();
        buffer.pop();
        std::cout << "Consumed: " << value << std::endl;
        cond.notify_one();
        return value;
    }
};

int main() {
    monitor moni;

    std::thread producer( f: [](monitor& m) -> void {
        for (int i = 0; i <= 10; i++) {
            m.produce( value: i);
            std::this_thread::sleep_for( rtime: std::chrono::milliseconds( rep: 500));
        }
    }, std::ref( &: moni));
    std::thread consumer( f: [](monitor& m) -> void {
        for (int i = 0; i <= 10; i++) {
            auto value : int = m.consume();
            std::this_thread::sleep_for( rtime: std::chrono::milliseconds( rep: 500));
        }
    }, std::ref( &: moni));

    producer.join();
    consumer.join();
    return 0;
}
```

图 2.12: 利用管程处理生产者消费者模型