

# 操作系统 A 期中

---

## 1. 操作系统概述

---

**KEY WORDS:** 操作系统不同角度的定义，并发性，共享性，虚拟性，随机性，中断，通道，**SPOOLing**，多道程序设计，**CTSS**，**OS/360**，**MULTICS**，操作系统结构

### 什么是操作系统

资源的“有效的”管理者，向用户提供各种“方便使用”的服务，可以对机器进行“扩展”。操作系统是计算机系统中的一个系统软件，是一些程序模块的集合。

从软件的角度：操作系统由各个模块组成（内在特性），向用户提供各种界面和接口（外在特性）。

从资源管理的角度：操作系统是从底向上的资源的管理者，具体包括的硬件资源有CPU，内存，设备等，软件资源有磁盘文件，信息等。对资源进行管理，是为了实现资源共享，提高其利用率，可以通过在时间角度复用和空间角度共享来实现。

管理资源包括：跟踪记录资源使用情况，分配和回收资源（公平竞争），提高资源利用率（共享），保护机制（读写保护等），协调进程对资源请求的冲突

基本功能：进程和线程的管理（包括进程控制，调度，通信，同步互斥等），存储管理（分配回收，地址映射，存储保护，内存扩充等），文件管理（文件系统，磁盘空间，存取控制等），设备管理（设备驱动，缓冲，分配回收等），用户接口（系统命令，编程接口等）

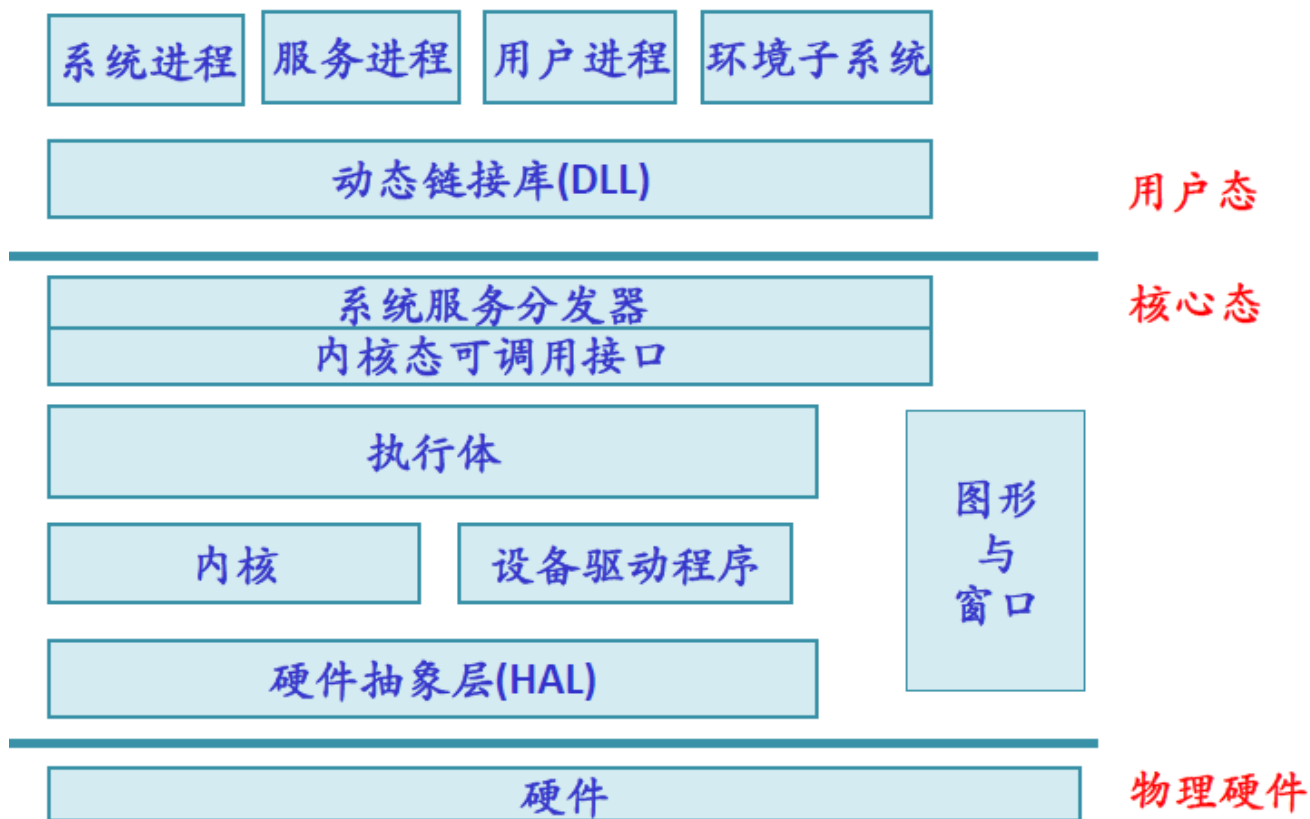
从进程的角度：操作系统是由一些可以同时但又独立运行的进程和一个对这些进程进行协调的核心组成，这个系统是动态的，不断运行的。

进程是程序的一次动态执行的过程，程序是进程的静态体现。进程会产生存在和消亡，是动态的有生命的。

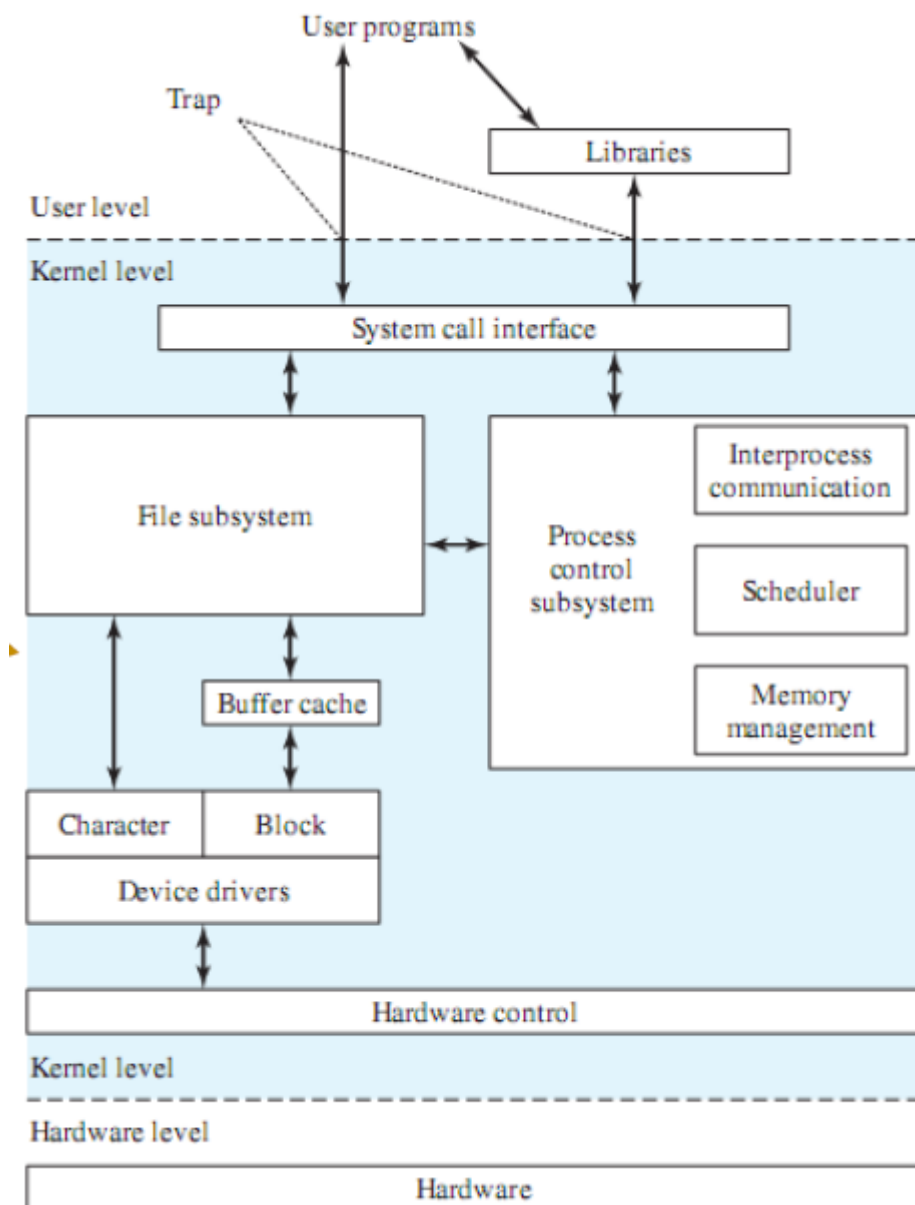
从虚拟机的角度：操作系统是分层的结构，每一层完成某些特定功能从而构成虚拟机并对上一层提供支持（通过界面和接口），通过逐层的功能扩充，最终完成整个操作系统虚拟机。

### 不同操作系统的架构

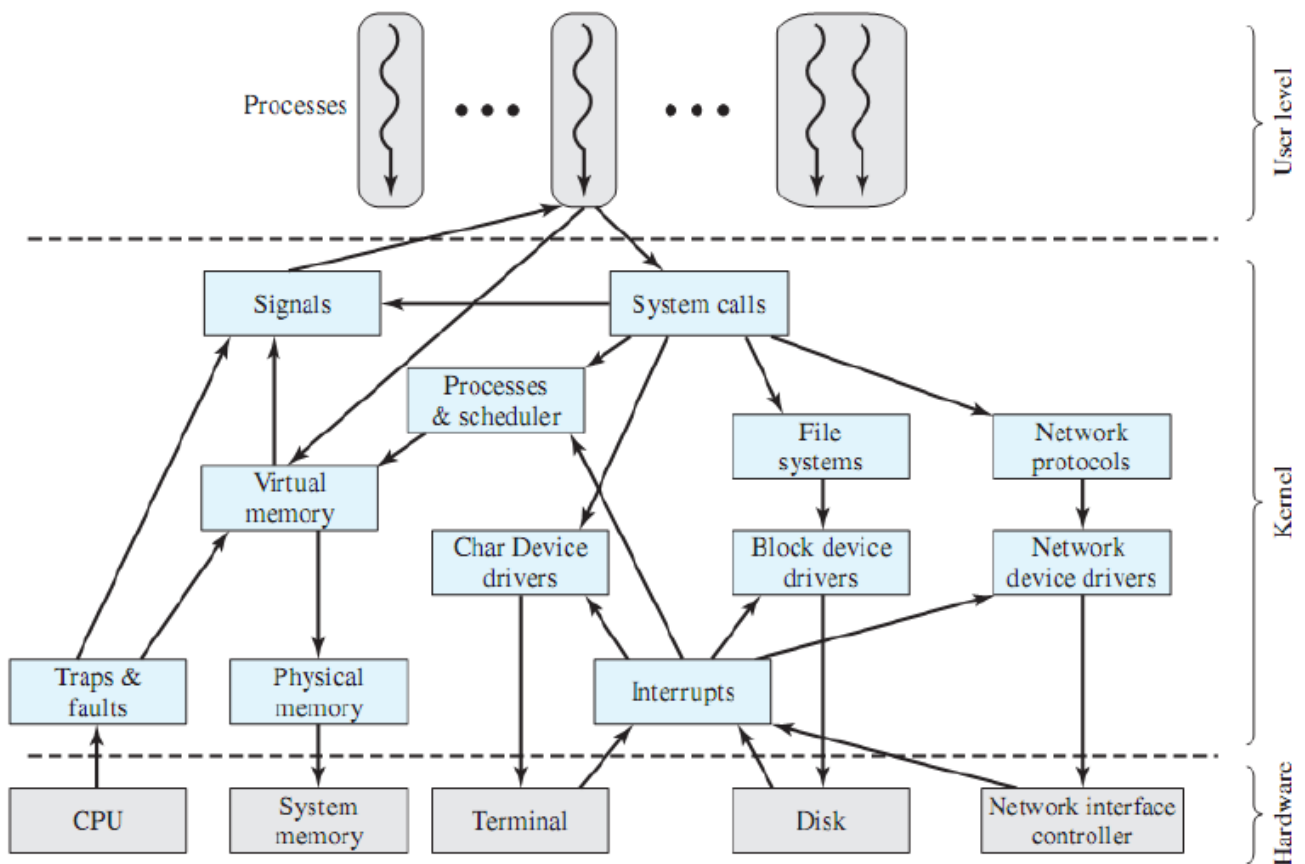
**Windows架构:** Windows在物理硬件上构建**HAL**（硬件抽象层），之后在其基础之上构建操作系统的核心态；在核心态之上构建**DLL**（动态链接库），并在其上构建用户态（系统进程，服务进程，用户进程等）。Windows架构自下而上可以分作计算机硬件层，操作系统核心层，系统功能调用层，应用程序层。



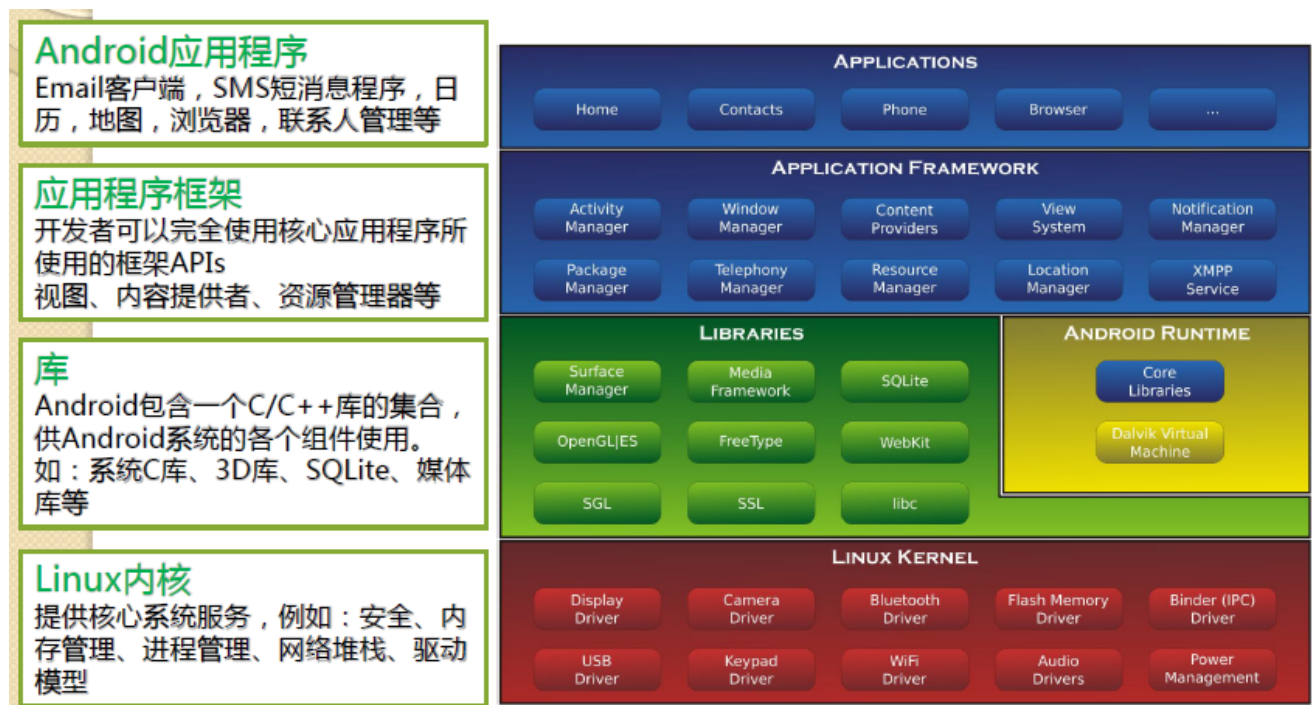
**UNIX架构:** UNIX在物理硬件之上构建硬件控制层，之后在其上构建核心态，在核心态上构建用户态（库属于用户态），用户态可以通过Trap的方式陷入核心态使用系统调用层。UNIX架构自下而上可以分作硬件层，Kernel（核心），系统调用层，UNIX命令和库+用户应用。



**Linux架构：**Linux在CPU，主存，磁盘，网络接口等硬件之上构建核心，Kernel（核心）中包括存储模块，文件系统模块，网络模块，外设模块，进程和调度模块等，在Kernel之上为用户层，用户层中全部为进程。Linux架构自下而上可以分作硬件层，内核（硬件抽象层+内核模块+系统调用接口），命令程序+编译器/解释器+系统库，应用程序。



**Android架构:** Android是建立在Linux内核之上的一个移动端操作系统，其充分体现了虚拟机的概念。Android架构自下而上为Linux内核，库，应用程序框架，Android应用程序。



Windows, UNIX和Linux都具有如下特点：在硬件之上构建硬件控制层，之后在控制层之上逐步构建操作系统内核，在内核之上构建接口层，最后在接口层之上为用户程序层。

Android体现了虚拟机的概念，Linux内核对于Android而言就是一个“虚拟机”，和硬件+硬件控制层相当。

## 操作系统的特征

**Concurrency并发：**有能力处理多个同时性的活动。在宏观上计算机系统中存在的多个程序是同时执行的；但对于单CPU计算机，任何时刻都只有一个程序在执行，这多个程序实在CPU上轮流执行的。

**Parallel并行与并发相似，**但多指的是硬件支持。

并发会在切换、保护、具有依赖关系的活动间的同步等方面存在问题。

**Sharing共享：**操作系统与多个用户程序共同使用计算机系统的资源。由于系统资源有限，因而必须共享有限的系统资源。

互斥共享：在时间上可以共享，但是必须是互斥的，比如多个计算机连接同一台打印机。

同时访问：在同一时刻，多个进程可以共享资源，比如可重入代码。

问题在于资源必须要对系统资源进行合理的分配和使用，而资源分配难以达到最优化。

**Virtual虚拟：**通过将一个物理实体映射为若干个逻辑实体，达到提供资源利用率的目的。虚拟技术是操作系统管理系统资源的重要手段，通过虚拟技术可以达到分时或分空间的目标。

CPU物理上只有一个（单CPU计算机）——每个用户进程都有一个“虚处理器”

存储器物理上只有有限的字节——每个进程都占有独立的地址空间（代码+数据+堆栈）

显示设备只有一个——多窗口或虚拟终端技术

**随机性：**操作系统必须随时对以不可预测次序发生的事件进行高效正确的响应。

引起的问题：进程运行速度不可预测；难以重现系统在某个时刻的状态（或错误）

## 操作系统发展历史

操作系统是随着计算机硬件技术、应用需求的发展、软件新技术的出现而发展的，其目标在于充分利用硬件资源和提供更好的服务。

**第一阶段（1948-1970）：**硬件昂贵，人工便宜。这一阶段特点是更有效地利用硬件资源而缺乏用户与计算机间的交互。控制台（独占资源）一次只能完成一个用户的任务，而批处理技术一次装入程序后可以不断运行处理很多用户的任务（线性处理，不需要保护机制），最后打印输出。为了能更好地利用硬件资源，提出多道程序设计和SPOOLing等技术。

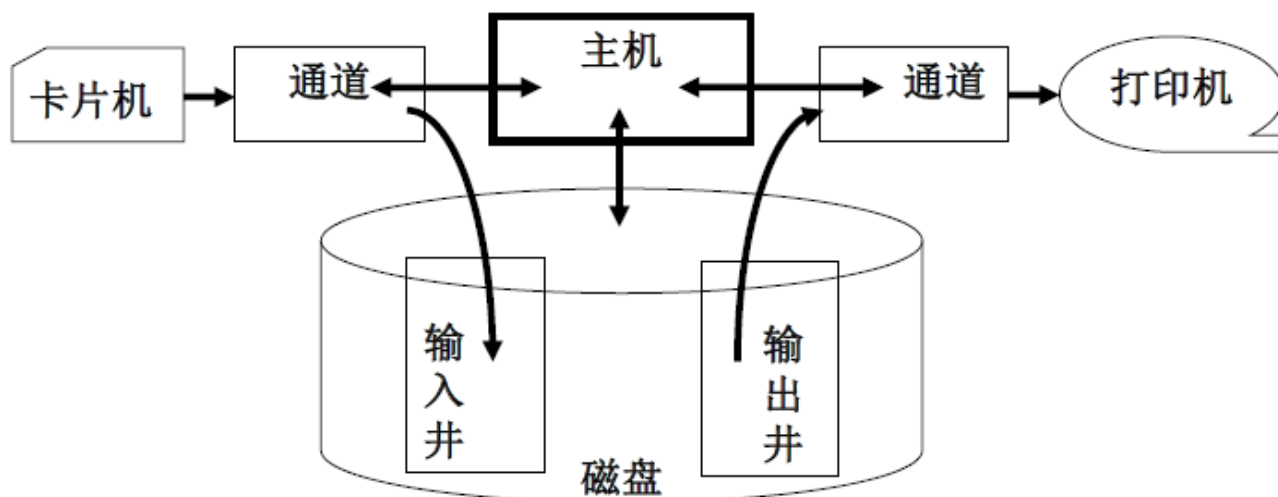
**数据通道和终端：**使得I/O和计算可以并行重叠执行。

**批处理：**用户将作业交给系统操作员，系统操作员将很多用户的作业合成一批输入到计算机，形成一个连续作业流，启动操作系统后系统自动按次序依次执行每个作业，最后由操作员获取解读结果并将其交给用户。

**多道程序设计：**多个程序同时运行，多个用户共享系统，需要存储保护。

**SPOOLing:** Simultaneous Peripheral Operation On-Line，同时的外围设备联机操作，又称作假脱机技术。该技术利用磁盘作为缓冲，将输入，计算，输出分别组织成为独立的任务流，使得I/O和计算并行。通过SPOOLing技术，人工输入→输入机→人工搬运→计算→人工搬运→输出机（打印机）可以合为一个机器，该机器通过输入井不断地读入和存储，通过输出经不断地读取和输出，而计算过程从输入井中获取作业，计算作业，最后将结果输出到输出井。目前的打印机等外设依然以SPOOLing方式进行使用。

- 作业进入到磁盘上的输入井
- 按某种调度策略选择几种搭配得当的作业，并调入内存
- 作业运行的结果输出到磁盘上的输出井
- 结果从磁盘上的输出井送到打印机



第二阶段（1970-1985）：硬件便宜，人工变得昂贵。这一阶段的特点是交互和分时——利用便宜的终端，多个用户可以同时与系统交互；通过牺牲CPU时间使得用户得到更好的响应时间。问题是增加用户时，系统的性能降低。

**CTSS**：第一个分时操作系统，由MIT在1959年提出，在1961年开发成功。32各交互式用户各有一个联机终端，调试程序的用户常常只发出简短的命令而很少会有很长的费时命令，计算机为32个用户提供交互式快速服务，同时还可以在CPU空闲时在后台运行大型作业。

**OS/360**：IBM开发，带着已知的1000个错误发布。

**MULTICS**：1963年开始开发，1969年才正式发布。

**UNIX**：贝尔实验室开发，初衷为在一台无人使用的DEC PDP-7小型机上玩电脑游戏。

第三阶段（1981-）：硬件非常便宜，人工非常昂贵。在硬件资源有限的时代，一次只运行一个程序，OS只是一个历程库，非常简单；这一阶段，个人计算机时代开始，PC资源丰富，大型OS出现——存储保护和多道程序设计等概念再次出现。

挑战在于如何充分发挥人的时间来利用计算机。

第四阶段（1981-）：分布式系统。网络出现，允许不同机器很容易地相互共享资源，比如打印机、Web等。

问题在于如何共享和安全问题。

第五阶段（1995-）：移动计算时代。各种小型、移动、便宜但能力十分有限的移动终端出现。

第六阶段（2006-）：云计算时代。提供可以无限扩展的、可以随时获取的、按需求使用的、按使用计费的资源网络。

**云计算操作系统：**云计算后台数据中心的整体管理运营系统，云平台综合管理系统，通常具备大规模基础软硬件管理、虚拟计算管理、分布式文件系统、业务/资源调度管理、安全管理控制等模块。作用在于：管理海量基础硬件，将海量硬件资源在逻辑上整合为一套硬件资源；为云应用提供统一标准的结构；为海量的云计算任务进行管理和资源调配。

第七阶段（200? -）：泛在计算/普适计算/物联网（IoT）。

## 传统操作系统分类

**批处理操作系统：**系统操作员从用户收集作业输入系统，形成连续作业流；启动操作系统后，系统自动按次序执行各个作业；系统操作员分发结果给用户。

批处理系统的作业：用户程序+数据+作业说明书（作业控制语言）。

批作业处理的流程：批作业中的各个作业的处理相同——读入用户作业和编译连接程序，编译链接用户作业生成可执行程序，启动执行并输出执行结果。

批处理操作系统的问题在于慢速输入输出的处理都由主机完成，在输入输出时，CPU空闲等待造成浪费。该问题的解决方法为使用“卫星机”，卫星机完成输入输出工作，中间结果暂存供主机使用；使用多道批处理系统，采用多道程序设计的思想，将多个批处理系统组合成一个来减少空闲等待浪费；使用SPOOLing技术。

**分时操作系统：**引入时间片的概念，操作系统将连续的CPU的时间划分成若干个片段。操作系统以时间片为单位轮流服务各个用户，每次服务一个时间片。

利用了用户的错觉，使得各个用户以为自己独占系统，因此其目标在于响应时间尽量短。

**响应时间：**从终端发出命令到系统回复所经历的全部时间。

**CTSS操作系统**是第一个分时操作系统。

**通用操作系统：**分时系统和批处理系统的结合。通过“分时优先，批处理在后”的原则，营造出可以频繁交互，同时又进行大量批处理的“假象”。

**前台作业：**需要频繁交互的作业。

**后台作业：**时间性要求不强的作业，一般是批处理作业。

**实时操作系统：**指计算机能够及时相应外部事件的操作系统，要求在严格规定的时间内完成对时间的处理，并控制所有实时设备和实时任务协调一致地工作。

实时系统要求对外部事件的响应时间和响应的可靠性严格控制在允许范围内。按照其关键参数时间可以有如下分类

**硬实时系统：**某个事件响应绝对一定必须在规定时间范围或规定时刻之前完成，比如导弹拦截系统。

**软实时系统：**可以甚至是必须接受偶尔的违例事件，比如工业控制系统（因而需要额外的质检系统减少违例事件的影响）。

## Tanenbaum操作系统分类

大型机操作系统

服务器操作系统

多处理机操作系统

**个人计算机操作系统：**计算机在某个时间内只为一个单一用户服务。其追求目标是使用方便界面友好，并且具有丰富的应用软件。

**网络操作系统：**基于网络计算机，是在各种计算机操作系统上按网络体系结构协议标准开发的软件系统。其追求目标是安全的通信和资源共享

**分布式操作系统：**所有系统任务可以在系统中的任何一个处理机上运行，自动实现全系统范围内各个处理机的任务分配和调度。

**分布式系统：**以计算机网络为基础的，处理和控制分散的计算机系统。

分布式操作系统的特征：操作系统统一，若干处理机可以协同完成一项任务；资源透明共享（对于分布用户来讲，是不知道分布的概念的）；平等自治，各处理机间没有主从关系；处理能力、速度和可靠性相较于各处理机（甚至各处理机之和）有所增强。

**嵌入式操作系统：**运行在嵌入式系统中，对整个嵌入式系统极其操作控制的各个部件进行统一协调调度、指挥控制的操作系统。

**嵌入式系统：**在各种设备、装置或系统中，完成特定功能的软硬件系统，通常工作在对处理时间和事件有比较严格要求的环境之中。

**智能卡操作系统：**只控制管理非常有限的运算、存储单元和设备的非常简易的操作系统。

**智能卡：**一种包含一块CPU芯片的信用卡。有非常严格的运行能耗和存储空间的限制，一般只有单项或少数几项功能；使用非常简单（相比于其他现代操作系统而言）的专用操作系统。

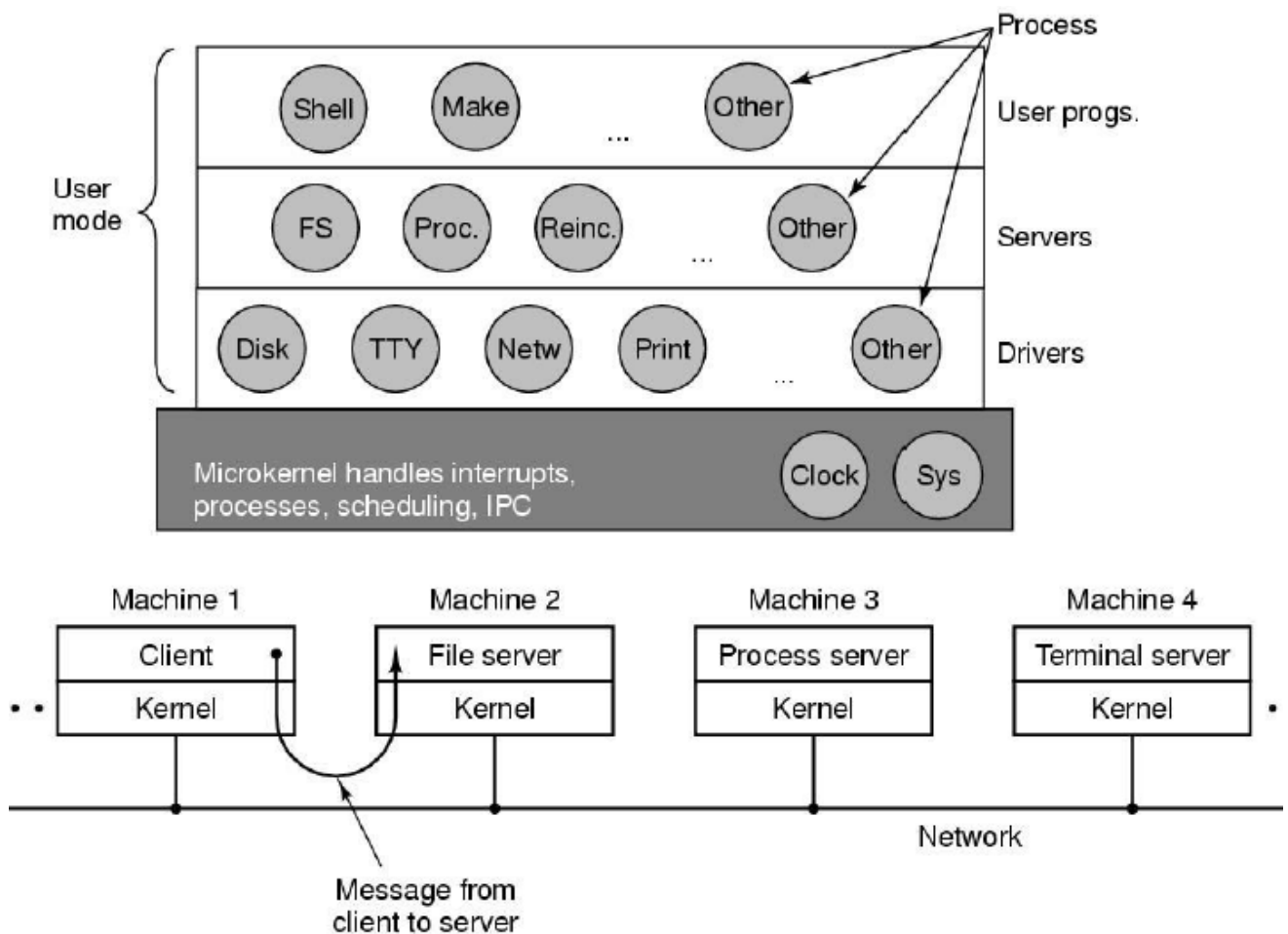
## 操作系统结构

**整体式结构：**按照过程集合的方式编写的操作系统，任何过程都可以自由调用其他过程，可以视作一个链接成的一个巨大的可执行文件。

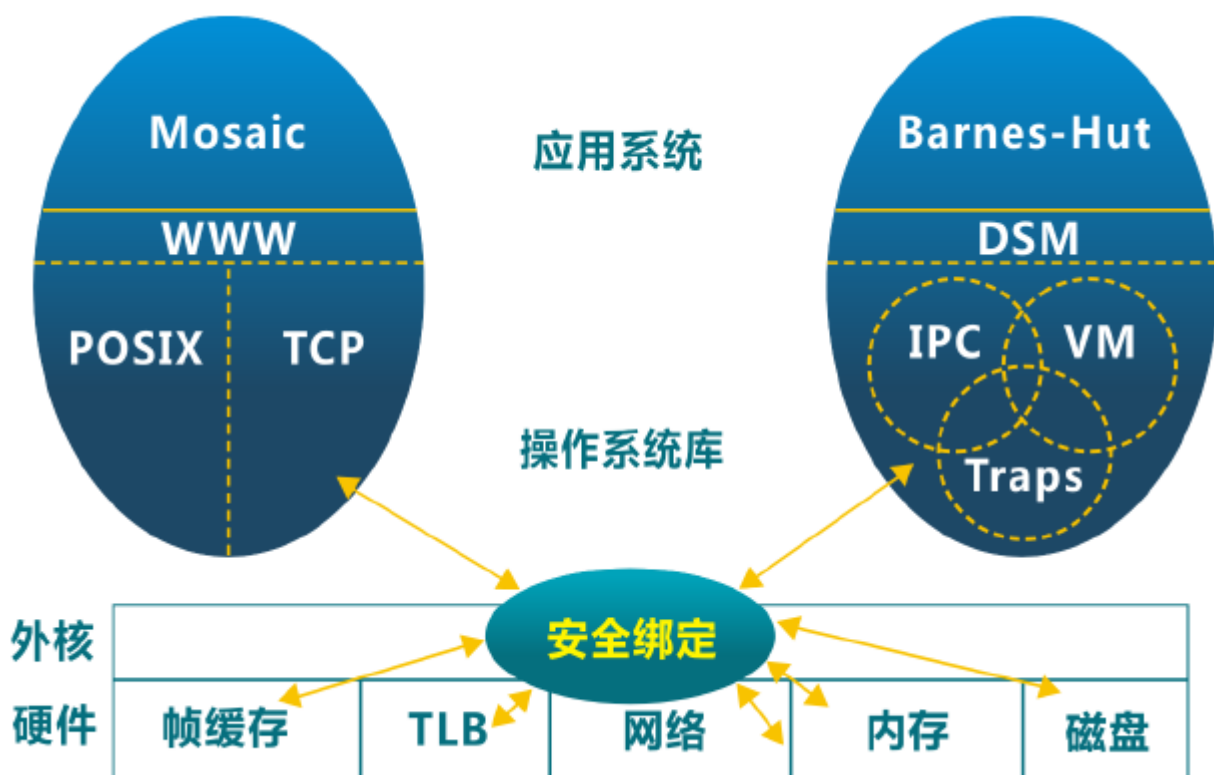
**分层结构：**每一层都建立在低层之上，且每一层都仅使用低层提供的操作和服务，目前主流的Windows和Linux操作系统都为着一种结构。

**微内核结构：**在内核空间只保留少量必须的功能，尽可能将内核功能移至用户空间，各个用户模块间使用消息传递进行通信。





外核结构：将保护和控制分离，内核分配物理资源给多个应用程序，各个程序可以链接到操作系统库（libOS）。



虚拟机结构：VMM（虚拟机管理器）将单独的机器接口转换成很多的虚拟机，每个虚拟机都是一个原始计算机系统的有效副本，并且可以完成所有的处理器指令。目前常见的JVM，Android都是虚拟机结构。

## 2. 运行环境和运行机制

**KEY WORDS:** CPU状态，特权指令，非特权指令，管态/目态，内核态/用户态，R0/R3，中断和异常，中断向量表，中断描述符，系统调用，机制和策略

### 操作系统运行环境

操作系统的运行环境，即为操作系统设计者必须考虑的硬件问题。

**CPU中央处理器：**处理器由运算器、控制器、一系列寄存器及高速缓存构成。

**用户可见寄存器：**高级语言编译器分配并使用，目的是减少程序访存，包括数据寄存器，地址寄存器和条件码寄存器等。**数据寄存器：**用于各种算逻指令和访存指令，又称通用寄存器。**地址寄存器：**用于存储数据及指令的物理地址、线性地址或者有效地址，用于某些特定方式的寻址，比如segment pointer，stack pointer等。**条件码寄存器：**保存CPU操作结果的各种标记位，比如溢出、符号等。

**控制和状态寄存器：**用于控制处理器的操作，一般由操作系统代码使用，只有在某些特权级之上才可以访问或修改。**PC程序计数器：**记录将要取出的指令的地址。**IR指令寄存器：**存放最近取出的指令。**PSW程序状态字：**记录处理器运行的状态，比如条件码、模式、控制位等。

保护是操作系统的基本需求，因为操作系统尽量要并发执行和共享资源，这就必须要实现保护和控制。需要硬件提供的基本运行机制：处理器具有特权级，可以在不同的特权级运行不同的指令集合；硬件机制可以将OS与用户隔离。

**CPU状态（模式）：**在PSW中设置一位，根据运行程序对资源和指令的使用权想设置不同的CPU状态。现代CPU通常将CPU状态设计为2-4种。

**内核态Kernel Mode：**在该状态下运行操作系统程序，又称管态、特权态、核心态、系统态等。

**用户态User Mode：**在该状态下运行用户程序，又称目态、普通态等。

**特权指令：**只能由操作系统使用，用户程序不可以使用的指令。

**非特权指令：**用户程序可以使用的指令。

**x86系列CPU的特权等级：**支持从R0到R3四个特权等级，特权能力从高到低。**R0**相当于内核态，**R3**相当于用户态，**R1**和**R2**介于两者之间不同级别可以运行不同的指令集和。目前大多数基于x86的CPU的操作系统只使用了**R0**和**R3**两个特权等级。

**用户态与内核态间的切换：**从用户态到内核态的唯一途径是**中断/异常/陷入**；从内核态返回用户态的途径是设置PSW。

**陷入指令（访管指令）：**操作系统提供给用户程序的接口，可以调用操作系统的服务，比如int，trap，syscall等。

**中断机制：**操作系统是“事件驱动”或“中断驱动”的，因而中断机制对于操作系统十分重要。中断机制可以使得操作系统及时处理设备发送的请求，及时捕获用户程序提出的服务请求，防止用户程序执行过程中的破坏性活动等。

**中断（外中断）：**外部事件引发，正在运行的程序所不期望的，比如I/O中断，时钟中断，硬件故障中断等。中断可以支持CPU和设备之间的并行操作。

**异常（内中断，例外）：**内部事件引发，由正在执行的指令引发，比如系统调用，Page fault，保护性异常，断点异常，除零异常等。异常可以对程序执行的错误进行修复（或终止），也可以提供系统调用的支持。

	Unexpected	Deliberate
EXCEPTIONS(sync)	fault	syscall trap
INTERRUPTS(async)	interrupt	software interrupt

中断/异常的过程：CPU暂停正在执行的程序，保留现场后自动转去执行相应事件的处理程序，处理完成后返回断点，继续执行被暂停的程序。事件的发生改变了CPU的控制流。

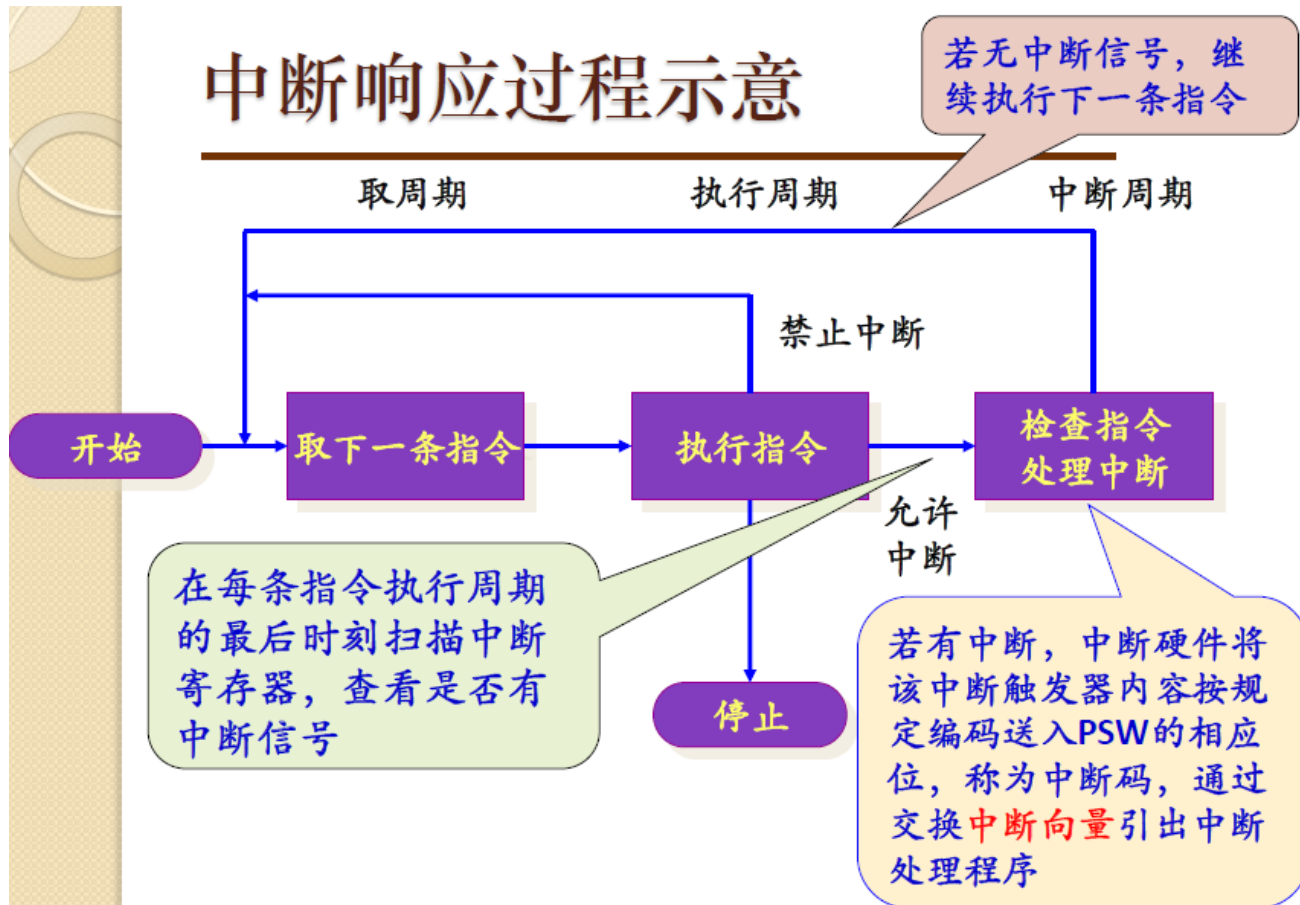
中断/异常的特点：随机发生的，自动处理的，可以恢复的。

类别	原因	异步/同步	返回
中断Interrupt	来自I/O或其它硬件设备	异步	总是返回到下一条指令
陷入Trap	有意识进行安排的	同步	返回到下一条指令
故障Fault	可以恢复的错误eg.缺页	同步	返回到当前指令
终止Abort	不可恢复的错误eg.除零	同步	直接终止，不返回

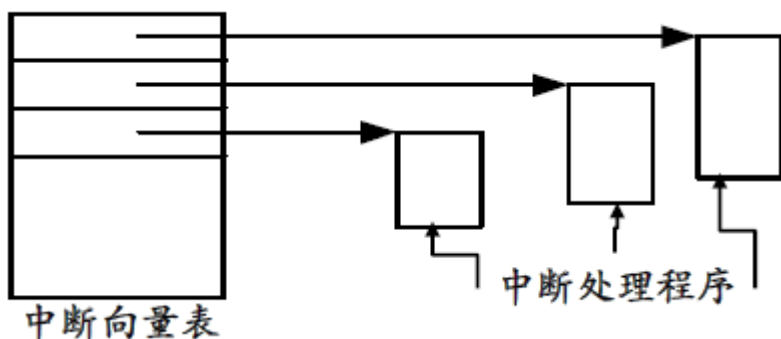
中断/异常处理：软硬件相互配合完成。硬件——中断/异常响应，捕获中断源发出的中断/异常请求，以一定的方式响应，将CPU控制权交给特定的终端程序；软件——中断/异常处理程序，识别中断/异常类型并完成相应的处理。

中断响应：发现、接收中断的过程，由中断硬件部件完成。中断寄存器的检查会在指令执行周期的最后时刻进行（如果允许终端的话），若有中断则设置PSW中断码，之后通过交换中断向量引出中断处理程序。

# 中断响应过程示意



中断向量：存放中断处理程序入口地址和程序运行所需的CPU状态字的内存单元。



中断响应的过程：设备发出中断信号；硬件保存现场；根据中断码查找中断向量表得到中断向量；把中断处理程序入口地址推送到相应的寄存器；执行中断处理程序。

中断处理程序：设置操作系统时，为每一类中断/异常事件都编好相应的处理程序，并设置好中断向量表。中断处理程序需要完成如下工作——保存相关寄存器信息，分析中断/异常的具体原因，执行对应的处理功能，恢复保存的现场，返回被打断的程序。

## 中断/异常机制示例

设备I/O中断：设备向CPU发送中断信号（硬件）；CPU执行完当前指令后检测到中断，判断出终端来源并向相关设备发送确认信号（硬件）；CPU为软件处理中断进行准备，将CPU状态切换至内核态并在系统栈中保存PC和PSW等被中断程序的上下文环境（硬件）；CPU根据中断码查中断向量表，获得中断处理程序入口地址，设置PC为该地址，将CPU控制权转移至中断处理程序（硬件）；中断处理程序开始工作，在系统栈中保存现场信息，之后开始进行中断处理，操纵I/O设备等（软件）；中断处理结束，CPU检测到中断返回指令，从系统栈中恢复被终端程序的

上下文环境，恢复PC和PSW，恢复CPU状态，CPU开始一个新的指令周期（硬件）。I/O操作正常结束的话，中断处理正常进行；I/O操作出现错误，重复执行失败的I/O操作直至达到重试次数上限，判定为硬件故障。

**时钟中断：**始终中断处理程序通常进行与系统运转、管理和维护相关的工作，具体包括——维护软件时钟，通过时钟中断对软件时钟进行维护和定时更新；**CPU时间调度**，维护当前进程的时间片软件时钟并进行调度；**控制系统定时任务**，检测死锁，系统记账等；**实时处理**。

**硬件故障中断：**硬件故障中断处理程序一般需要保存现场，使用警告手段，并提供辅助诊断信息，在高可靠系统中还需要评估系统可用性，如果可用还要尽快尽可能恢复系统。

**程序性中断：**由程序指令出错，指令越权或者指令寻址越界等引发。一种处理方法只能由操作系统的相关功能模块完成，如Page Fault；一种可以由程序自己完成，比如一些算术运算错误（溢出等）。

## IA32体系结构对中断的支持

**x86中断：**由硬件引发，分作可屏蔽中断和不可屏蔽中断。

**x86异常：**由指令执行引发（如除零异常），对于某些异常，CPU会在执行异常处理程序之前产生硬件出错码并压入内核态堆栈之中。

**x86系统调用：**异常的一种，是用户态到系统态的唯一的入口。

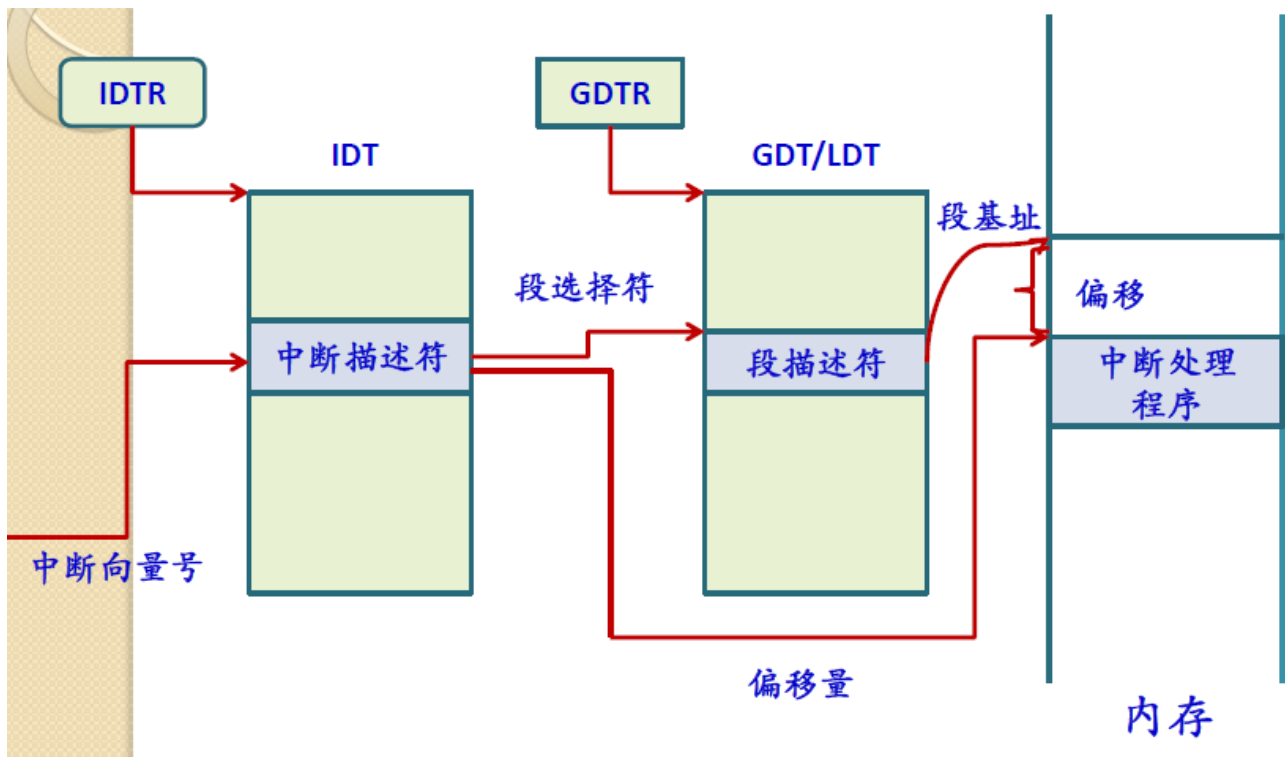
**PIC或APIC中断控制器：**负责将硬件的中断信号转换为中断向量，引发CPU中断。

**中断向量表：**存在于实模式之下，存放中断服务程序的入口地址，入口地址=段地址 $\ll 4$ +偏移地址，不支持CPU运行状态切换，中断处理与一般的过程调用相似。

**IDT中断描述符表：**存在于保护模式之下，采用门描述符数据结构来描述中断向量。包含四种类型门描述符——任务门（**Task Gate**），中断发生时，必须取代当前进程的的那个进程的TSS选择符存放在任务门之中（Linux没有使用）；中断门（**Interrupt Gate**），给出段选择符和段内偏移量，通过中断门后系统会自动地禁止中断；陷阱门（**Trap Gate**），类似中断门，但通过陷阱门后系统不会自动关终端；调用门（**Call Gate**）。

中断/异常的硬件处理过程：确定与中断/异常关联的向量i；通过IDTR找到IDT，获得表中的第i个中断描述符；从GDTR获得GDT地址，结合中断描述符中的段选择符，在GDT中获取对应的段描述符；从段描述符中获得中断/异常处理程序所在的段基址；特权级检查，只有特权级足够才可以通过；检查是否发生特权级变化，若是则切换堆栈，使用与新的特权级相关的栈；硬件压栈保存上下文环境，如果产生硬件出错码也需要压栈；如果是中断，清除IF位；通过中断描述符的段内偏移量和段描述符的段基址，找到中断/异常处理程序的入口地址，执行其第一条指令。

**特权级检查：**确保CPL（当前程序运行的权限级别）小于等于门描述符DPL，有效权限级别（数值越小权限越高）高于DPL才可以通过；确保CPL小于等于段描述符DPL，有效权限级别高于DPL才可以通过。



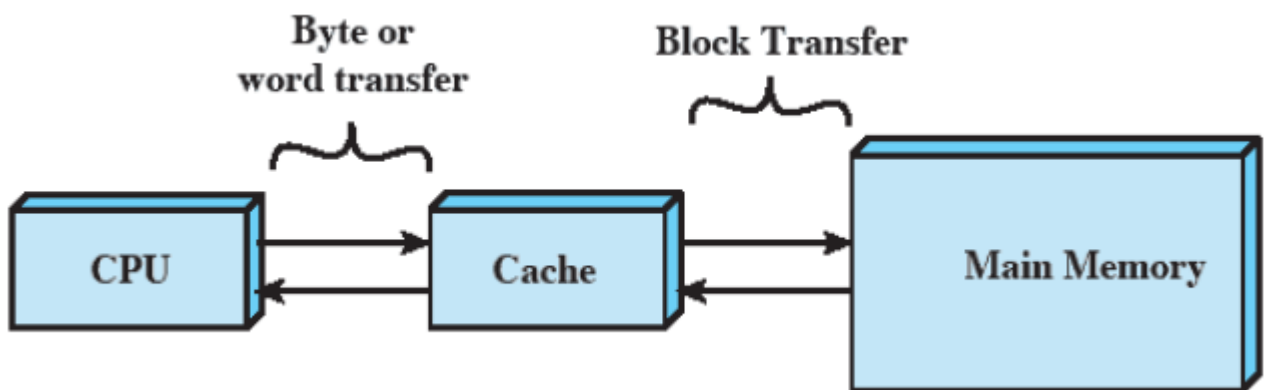
**存储系统：**支持OS运行的硬件环境。操作系统必须要管理、保护程序和数据，使之不受到破坏。

**存储层次结构：**自上而下速度由快到慢，容量由小到大，成本由高到低，自上而下分别为寄存器，高速缓存，主存（内存），磁盘（外存）。

**局部性原理：**在较短时间内，代码执行会重复在相同的指令集合内，数据存取稳定保持在一个存储器的局部区域；一段时间后，代码和数据集合可能会改变。利用局部性原理，是提高存储系统性能的关键。

**存储分块：**存储最小单位为bit，寻址最小单位为byte；为了简化分配和管理，存储器划分为Page，不同系统的Page的大小不定。

**高速缓存：**对操作系统不可见，是在CPU和内存之间设置的一个容量小但速度快的存储器，可以加速读写。Cache与主存间页传输，与CPU间字节或字传输。CPU读取byte/word时，需要先检查Cache，如果在Cache中可以直接读取，如果不在Cache中则Cache从主存调取block后放入Cache（替换以及被替换block写回），之后再进行传输。



**I/O访问：**程序控制方式，中断驱动方式，DMA（直接存储器存取）方式

**程序控制I/O技术：**CPU直接提供I/O相关指令，I/O处理单元处理请求并设置I/O状态寄存器相关位，不中断CPU，也不给CPU警告信息，CPU定期轮询I/O单元直到处理完毕。问题在于CPU必须时刻关注I/O单元状态，而耗费大量时间降低了系统性能。

**中断驱动I/O技术：**I/O处理单元会中断CPU，将CPU从轮询中解放出来，使得I/O操作和指令执行可以并行。问题在于CPU仍旧必须控制传送数据，仍然耗费时间，效率不高。

**DMA技术：**系统总线中有DMAC作为独立的数据传送控制单元，自动控制成块数据在内存和I/O单元间传送，提高了I/O性能。问题在于CPU和DMA传送并不能完全并行，因为CPU和DMA会竞争总线，尽管这种竞争不会引起中断也不引起程序上下文保存，CPU使用总线时会稍作等待，因而DMA传送时CPU访问总线的速度会变慢。

**DMA传输的流程：**CPU需要读写一整块数据时，向DMA控制单元发送一条命令，该命令包含读或写请求，I/O设备编址，开始读写的内存编址，传送数据长度等；CPU发送完命令之后可以去做其他事情；DMAC自动完成数据传送，当传送完成后，DMAC向CPU发送中断。

**时钟：**周期性信号源，用来处理与时间有关的时间。

**绝对时钟：**记录当前时间。一般而言，绝对时钟绝对准确，当停机后，绝对时钟仍然自动运行修改。

**相对时钟（间隔时钟）：**用时钟寄存器实现。设置时间间隔初值，每经过一个单位时间，时钟值减一，减至负数则触发时钟中断并进行相应的中断处理。

**硬件时钟：**使用某个寄存器模拟，根据脉冲频率定时增减。

**软件时钟：**用作相对时钟，时钟内存单元来模拟时钟。

**x86体系结构的定时硬件：**RTC实时时钟，TSC时间戳计数器，PIT可编程间隔定时器，SMP系统中的本地APIC定时器。

内核使用RTC和TSC跟踪当前时间。内核对PIT和本地APIC编程，使其以固定的、预先定义好的频率发出中断。

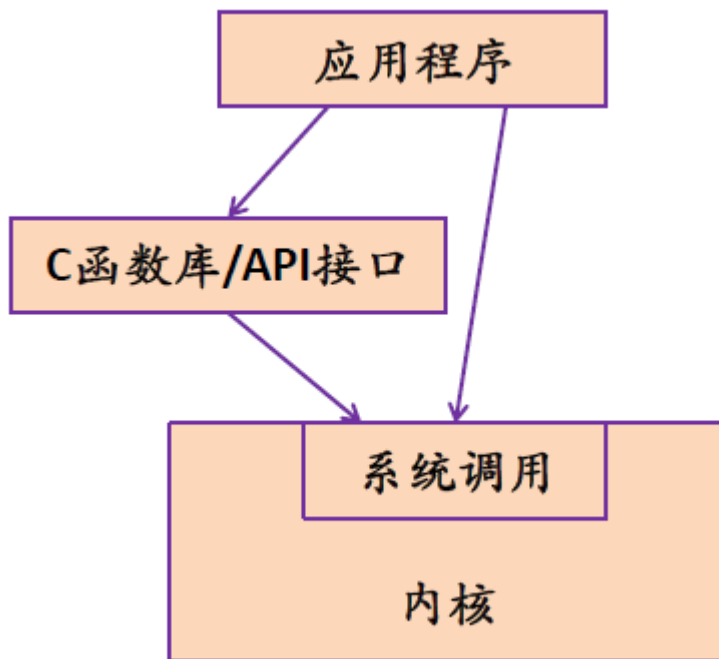
**Linux计时体系结构：**Linux提供一组与时间流相关的内核数据结构和函数。单CPU系统中各种计时活动由PIT产生的时钟中断触发；多CPU系统中，普通活动（如软定时器的处理）由PIT产生的中断触发，而具体的CPU活动（如监视当前进程的运行时间）由本地APIC产生的中断触发。

## 操作系统的运行机制

需要考虑系统调用的种种。

**系统调用：**用户在编程时可以调用的操作系统功能，是操作系统提供给编程人员的唯一接口，也是使CPU状态从用户态陷入到内核态的唯一途径。

**库函数/API接口：**在系统调用的基础之上，封装出的用户态的接口；应用程序可以使用库函数和API接口，也可以直接使用系统调用。



系统调用通过中断/异常机制进行实现。

陷入/访管指令：一条特殊的指令，可以引发异常，从用户态切换到内核态，是系统调用的实现指令。

系统调用表：存放系统调用服务例程的入口地址，每个系统调用都需要事先给定一个编号，称作系统调用号。

系统调用传参：指令自带参数，参数非常有限；通过通用寄存器传参，比较普遍，OS和用户都可以访问，但参数数目受寄存器数量限制；内存中开辟专用堆栈区传参。

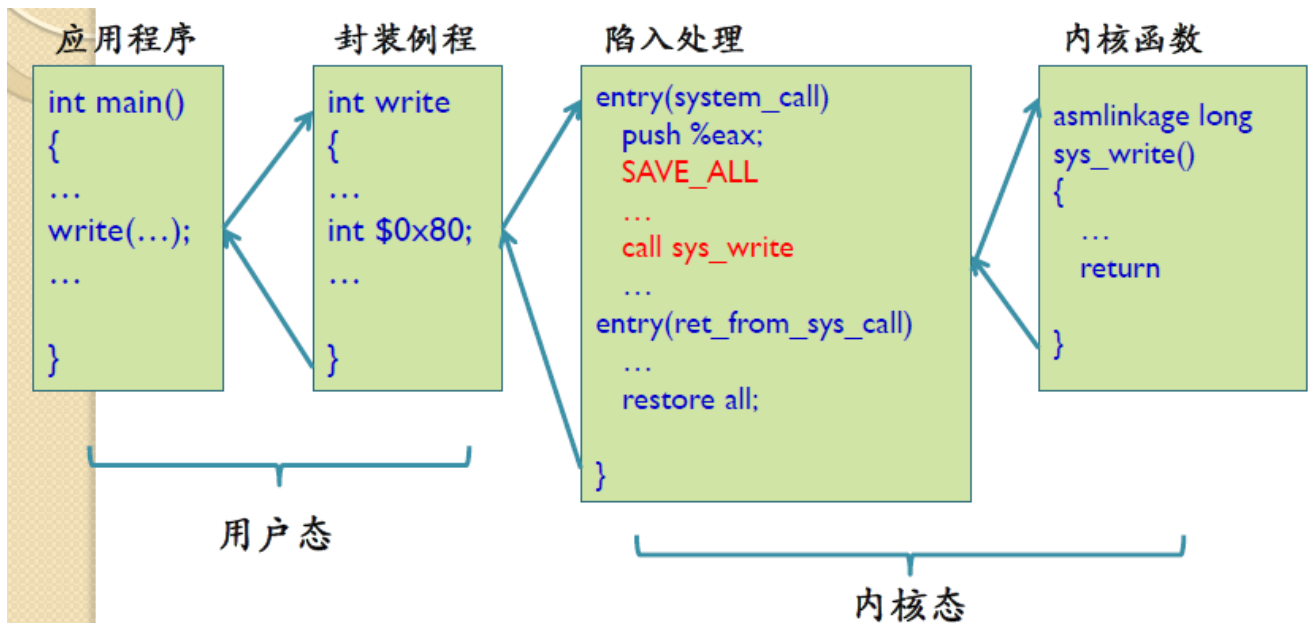
**Linux系统中使用int 0x80指令进行系统调用，%eax存放系统调用号。**

系统调用执行过程：中断/异常机制，硬件保存现场，查中断向量表将控制权转移给系统调用总入口程序；在系统调用总入口程序中，保存现场，将参数压进内存堆栈中，查系统调用表将控制权转移给响应系统调用例程或内核函数；执行系统调用历程；恢复现场，返回用户程序。

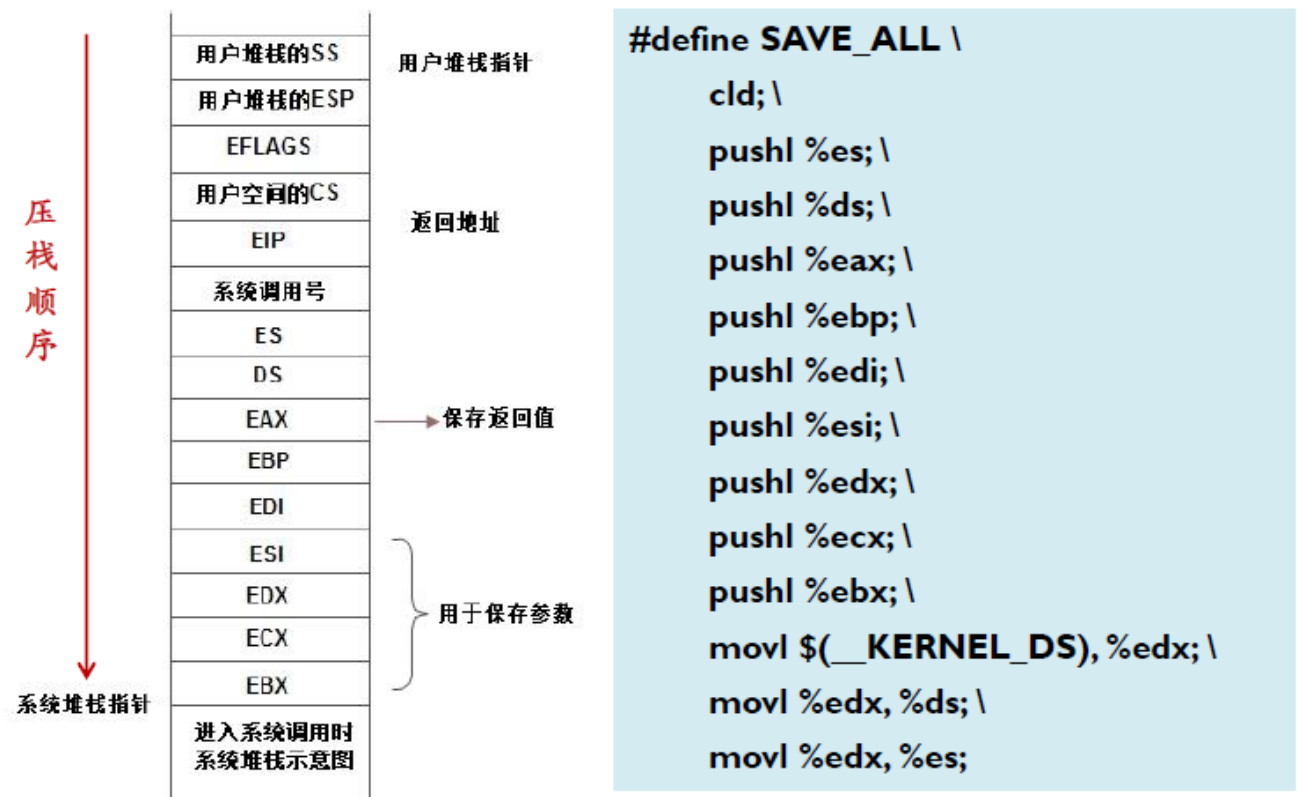
基于x86体系结构的Linux系统调用：int \$128或int \$0x80，使用陷阱门，DPL=3（与用户级别相同）。

int \$0x80执行过程：特权级改变，切换至新的特权级关联的栈（用户栈->内核栈）；将用户栈信息（SS:ESP），EFLAGS，用户态CS，EIP寄存器内容压栈留作返回时使用；EFLAGS压栈后复位TF，IF保持不变；在IDT中找到128位门描述符，找出段选择符装入CS；通过CS: offset定位入口地址；特权级检查；...





int \$0x80进入系统调用处理程序后的系统堆栈的结构为



## 机制与策略分离原则

“机制是接口，策略是数据流”。

机制可以被视作适用性很广的规则；而策略是实现机制的比较具体的做法。当具体情况有变时，规则不需要改变，但怎么实现规则会有所变化。

从软件开发的角度来考虑，机制可以理解为实现某个功能需要的原语操作和结构，而策略可以理解为实现某个功能的直接实现。

操作系统需要尽量地“提供机制，而不是策略”。

### 3. 进程线程模型

**KEY WORDS:** 进程，进程状态，进程状态转换，进程控制，进程控制块，进程地址空间，进程映像，线程，线程属性，Web服务器，用户级线程，Pthreads，核心级线程，原语，可再入程序

#### 进程模型

多道程序设计：为了提高系统效率，允许多个程序同时进入内存并运行。

并发环境：一段时间内，单CPU上有两个或以上的程序同时处于开始运行但尚未结束的状态，并且次序不是事先确定的。

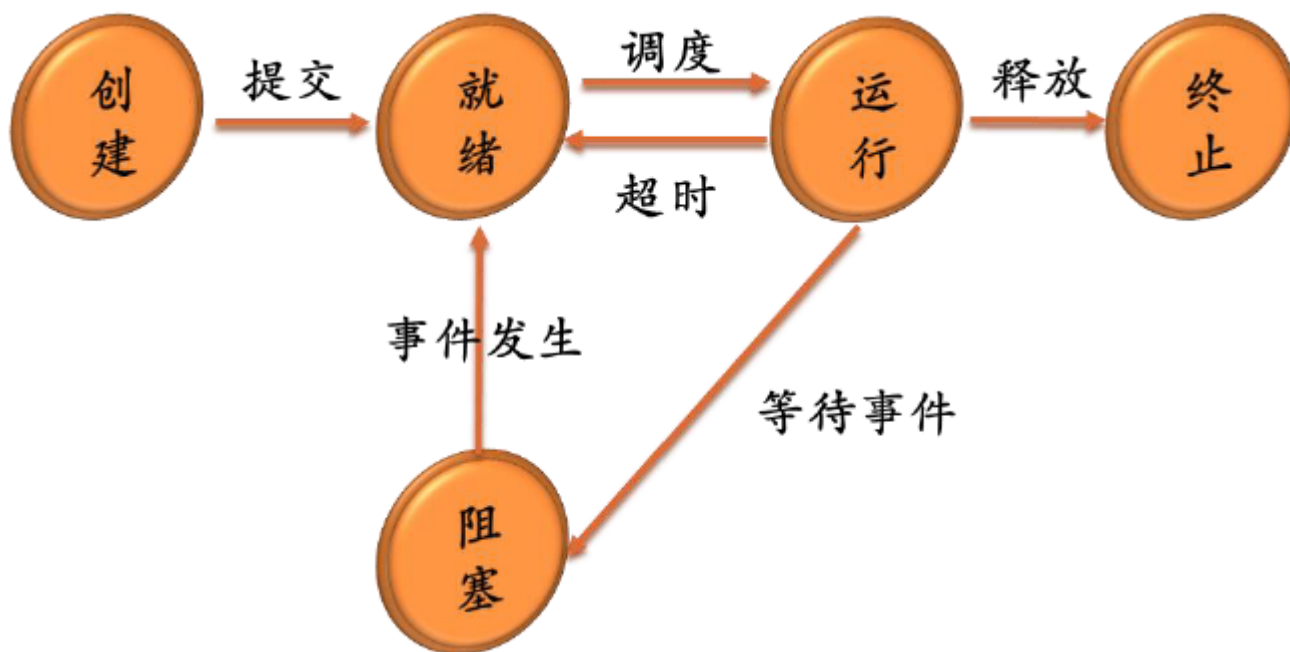
并发程序：在并发环境中执行的程序。

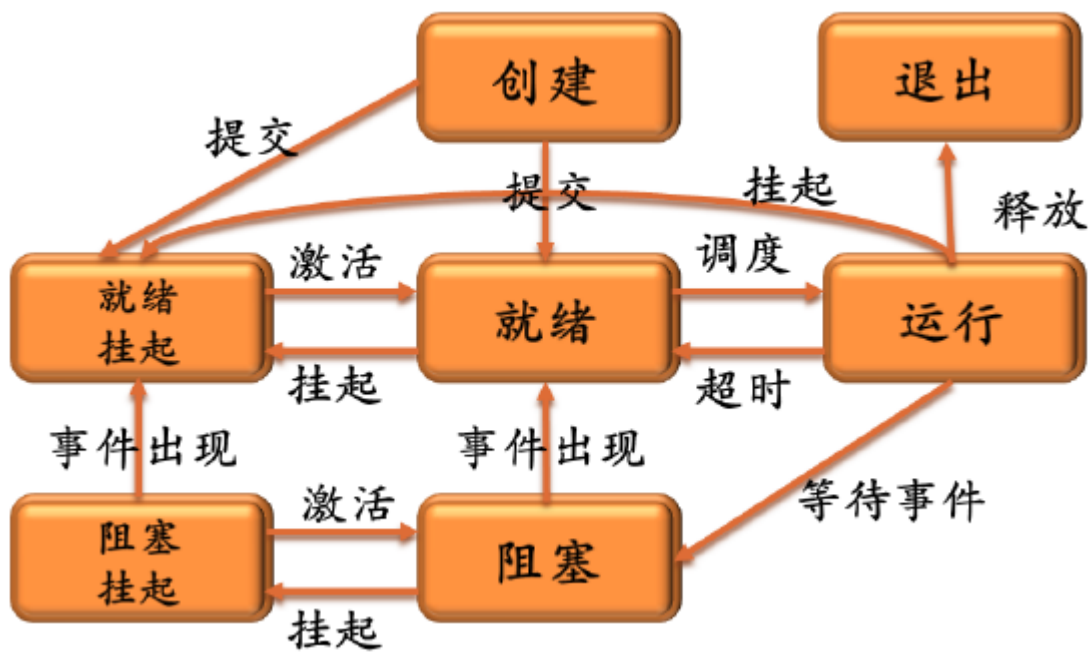
**进程Process:** 程序的依次执行过程，是对CPU的抽象，是正在运行程序的抽象，通过进程的抽象可以将一个CPU变幻为多个虚拟的CPU。进程是具有独立功能的程序关于某个数据集合上的一次运行活动，是系统进行资源分配和调度的独立单位。

系统资源（如内存，文件等）以进程为单位进行分配，每个进程具有相互独立的地址空间。

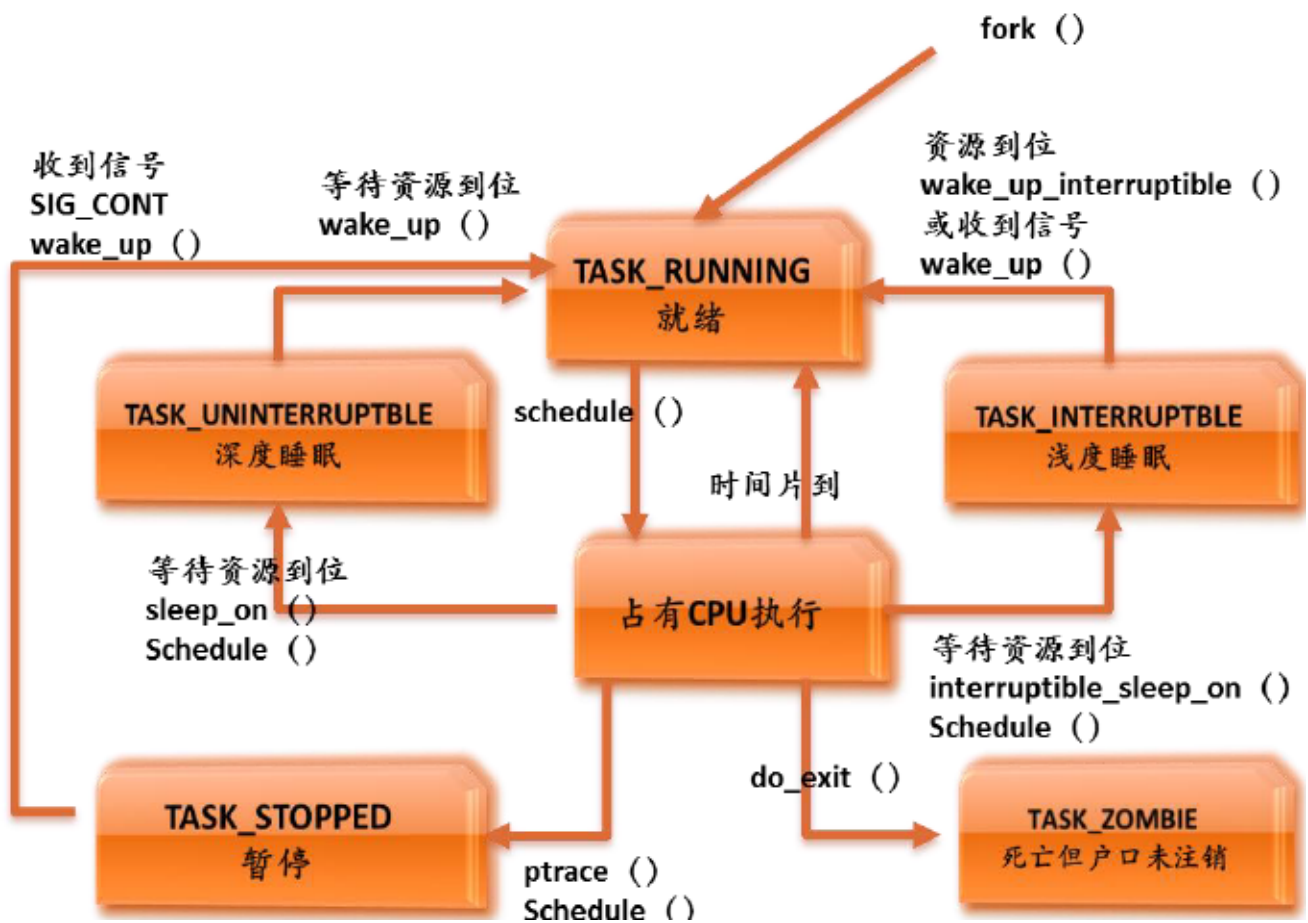
**进程状态:** 进程有三种基本状态——运行态Running（占有CPU并在其上运行），就绪态Ready（具备运行条件，但由于CPU不空闲而暂时无法运行），等待态Waiting/Blocked（因为等待某个事件而暂不能运行）。其他状态还有创建态New（完成创建进程但尚未进行执行），终止态Terminated（进程终止执行后，进程进入该状态，等待资源回收），挂起态Suspend（用于调节负载将一个进程从内存转移到外存，处于该态的进程不占用内存空间，其进程映像交换到磁盘上）。

进程状态转换





**Linux进程状态模型：**六个状态——就绪，执行，深度睡眠，浅度睡眠，暂停，僵尸态。深度睡眠和浅度睡眠都是等待资源，区别在于浅度睡眠可以被信号唤醒而深度睡眠不可；暂停类似于挂起；僵尸态是已经结束但未被回收的状态。

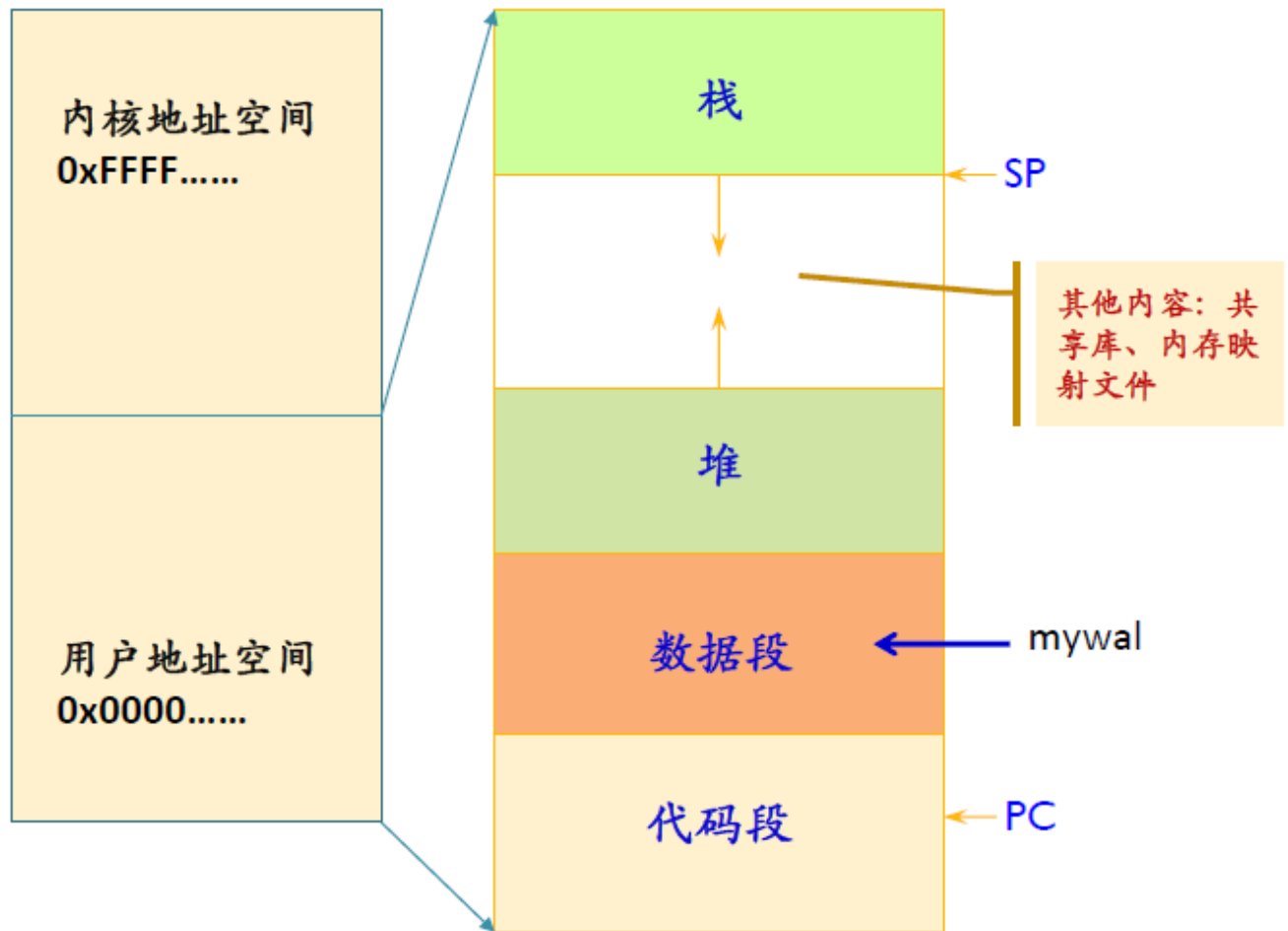


**PCB进程控制块：**是操作系统表示进程的一个专门的数据结构，用于记录进程的各种属性和描述进程的动态变化过程。

操作系统通过PCB控制和管理进程，PCB是操作系统感知进程存在的唯一标志和方式，PCB与进程一一对应。

PCB的内容包括：**进程描述信息**——PID进程标识符，进程的唯一编号；UID用户标识符，描述进程组关系；进程名，不唯一，一般来自于可执行文件名。**进程控制信息**——进程状态，优先级，代码执行入口地址，程序磁盘地址，运行信息统计，进程是否阻塞及阻塞原因，进程队列指针，消息队列指针等。**所拥有的资源和使用情况**——虚拟地址空间情况，打开文件列表等。**CPU线程信息**——通用寄存器，PC，PSW，堆栈指针，段/页表指针等，在进程不运行时，OS必须保存这一部分硬件环境。

**进程地址空间：**每个进程拥有独立的，互相不干涉的地址空间。地址空间的低位为用户地址空间，从低到高为代码段，数据段，堆，其他内容（共享库，内存映射文件等），栈；高位为内核地址空间。

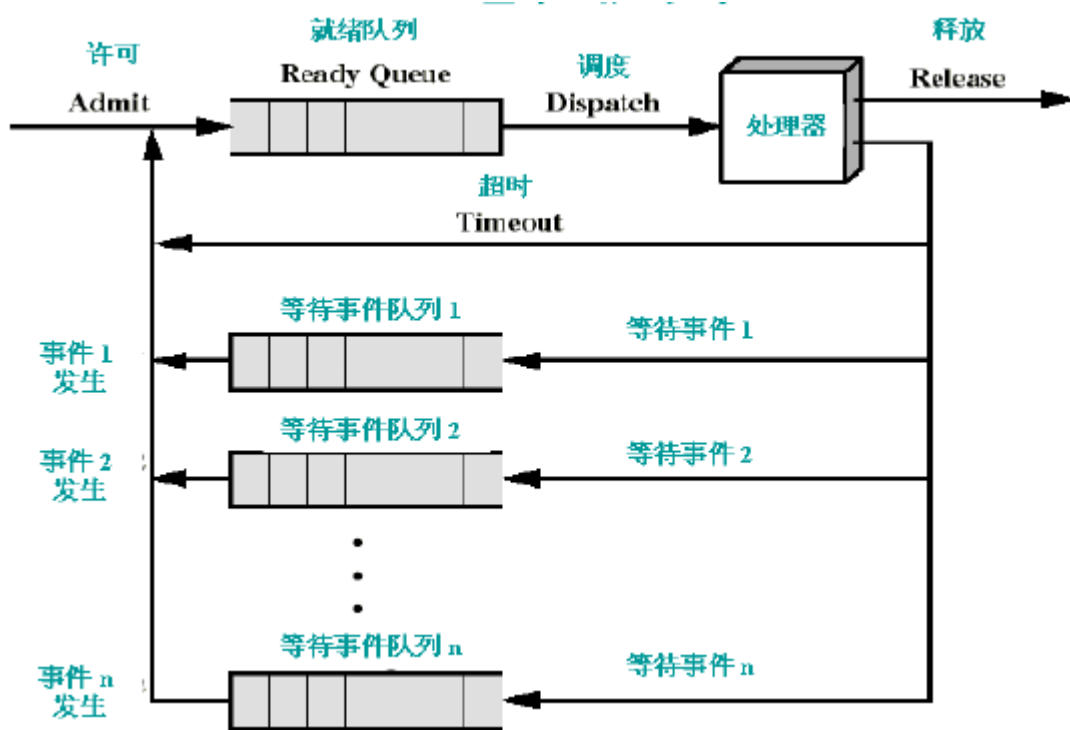


**上下文切换（Context switch）：**将CPU硬件状态从一个进程切换到另一个进程的过程称为上下文切换。进程运行时，硬件状态保存在PC，PSW，堆栈指针，通用寄存器，其他控制寄存器等CPU上的寄存器之中；当进程不在执行时，这些寄存器的值应该保存在PCB之中。

**进程映像：**程序+数据+栈+PCB。

**进程队列：**按照进程类别（一般为进程状态）建立的队列，每一类进程对应一个或多个队列。进程队列的元素为PCB，随着进程状态的改变，PCB会从一个队列进入到另一个队列之中。

一般情况下，单CPU只有一个就绪队列，但可以由多个等待队列去等待不同的事件。



## 进程控制

原语：完成某种特定功能的一段程序，具备不可分割性或不可中断性。原语的执行必须是连续的，在执行过程中不允许被中断。

原语是一种原子操作。

创建进程：分配唯一标识PID和PCB；分配地址空间；初始化PCB，设置为默认值；设置相应的队列指针；创建/扩充其他数据结构。

撤销进程：结束子进程或线程；会受进程占有的资源（关闭打开的文件，回收分配的内存等）；撤销PCB，注销PID。

进程阻塞：处于运行态的进程，在运行时期待着某一事件发生，但被期待的时间未发生时，由进程自己执行阻塞原语并使得自己转为阻塞态等待事件。

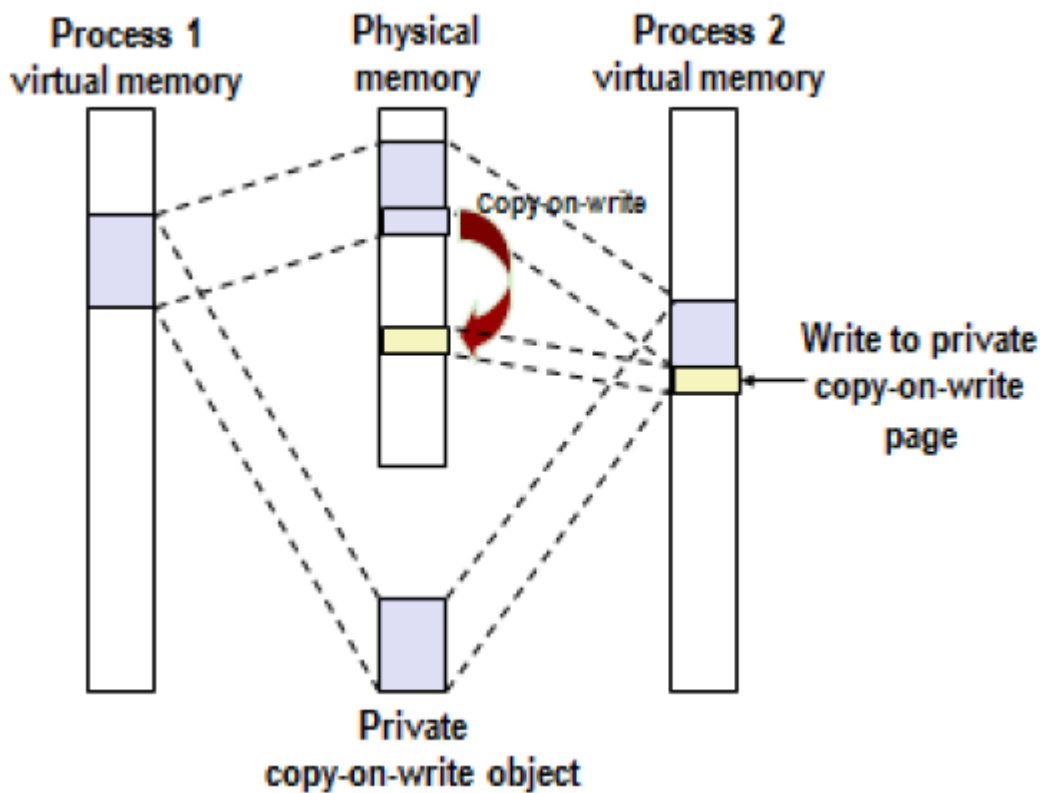
进程唤醒：处于阻塞状态的进程等待的事件发生，进程被唤醒，重新进入就绪队列中。

**UNIX的进程控制操作：**均通过系统调用完成进程状态转换

**fork():** 通过复制调用进程来建立新的进程，是最基本的进程建立的过程；**exec():** 包括一系列系统调用，用一段新的代码覆盖原来的内存空间，从而实现进程执行代码的转换；**wait():** 提供初级进程同步措施，使得一个进程等待，直到另外一个进程结束为止；**exit():** 终止一个进程的运行。

**UNIX的fork():** 首先为子进程分配一个唯一标识PID并分配一个PDB；从父进程继承打开的文件等共享资源；采用Copy-on-Write技术，按一次一页的方式复制父进程地址空间；将子进程状态设置为就绪，插入至就绪队列；对子进程返回0，对父进程返回子进程的PID。

**私有写时复制共享（COW）：**父子进程共享一段物理内存空间，二者PTE在这段空间上被设置为只读；当父/子进程在这段空间上有写指令时，触发保护性错误**Protection Fault**，之后异常处理函数创建一个新的R/W页并为子进程分配该页；处理函数返回后重新执行写指令；COW完成。



**COW**使得线程在`fork()`后可以有相同的虚拟地址而共享物理地址，避免了分配物理地址引起的时空浪费。若`fork()`后`exec()`则子进程全部物理地址都要进行分配；否则只有`fork()`的话，子进程在后续执行过程中会分配数据段和堆栈段，但代码段还是共享的。一般在`fork()`后直接进行`exec()`来防止出现浪费，因为UNIX默认优先执行子进程，若子进程没有及时`exec()`而父进程进行了写操作，那么会导致父进程进行无用的写时复制。

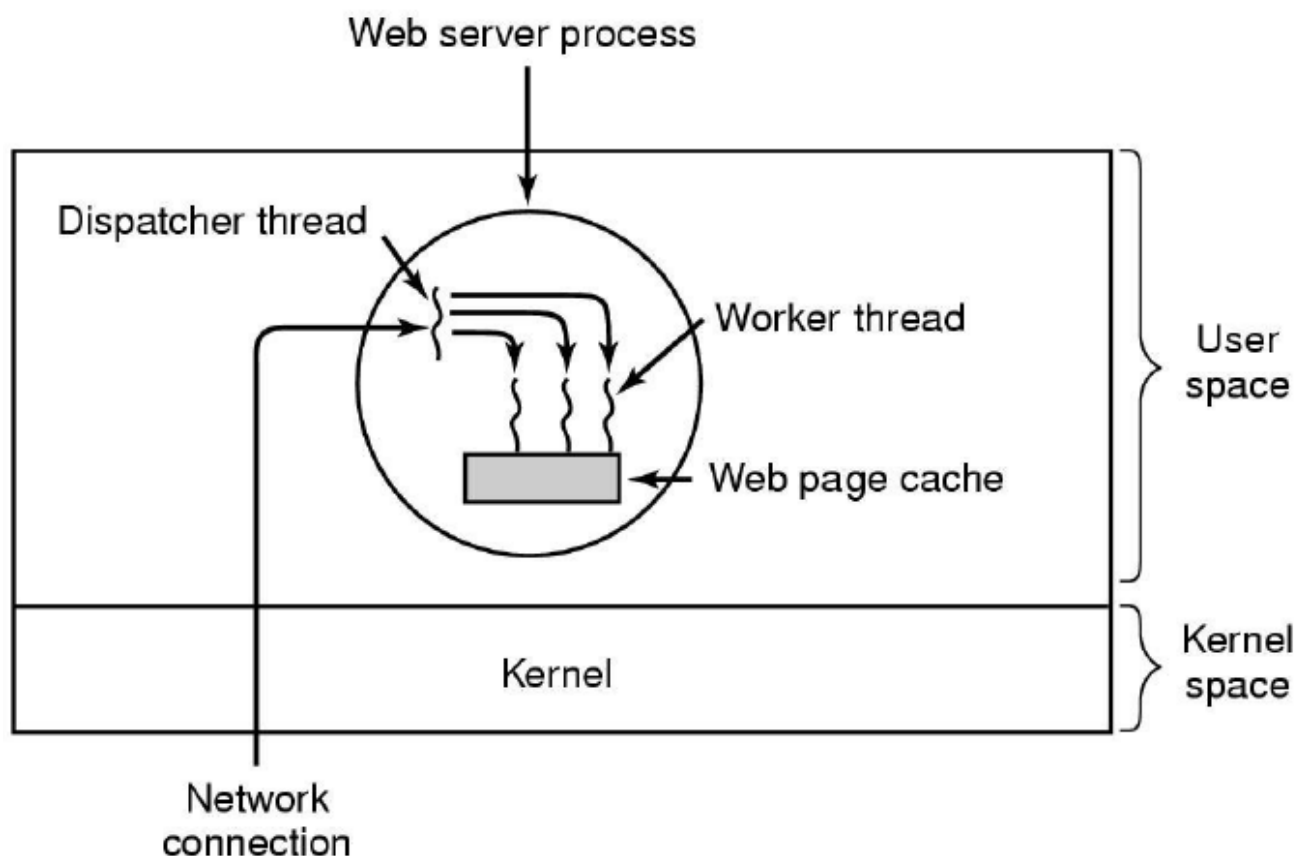
**可再入（可重入）程序**：可以被多个进程同时调用的程序，具备如下性质——纯代码的，即在执行过程中自身不改变；调用它的进程应当提供数据区。

## 线程模型

已经有了进程，为什么要派生线程：应用的需求，开销的考虑，性能的要求。

**应用的需求**：比如**Web服务器**，需要同时高速处理大量用户的简单请求，如果没有线程的话，有两种解决方案——一个服务进程（编程较容易，但性能下降）；有限状态机，采用非阻塞I/O（性能基本相同，但编程模型复杂）。





## 一个多线程的Web服务器

选自Tanenbaum《

模型	特性
多线程	并行，阻塞系统调用
单线程进程	无并行，阻塞系统调用
有限状态机	并行，非阻塞系统调用，中断

多线程可以天然应用在多道程序设计系统之中，用于：解决前后台操作处理的问题，解决应用中的异步处理问题，解决应用程序的执行速度问题，解决程序的模块化设计问题。在多线程应用中还可以应用**SPOOLing**技术。线程并不是越多越好，一般具有一个上限，可以通过线程池的方式来循环利用有限个线程。

开销的考虑：进程创建、撤销、通信、切换的时空开销大，会限制并发度的提高；线程的开销小——创建撤销简单，线程切换时间花费也短，线程间共享内存和文件通信不需要通过内核。

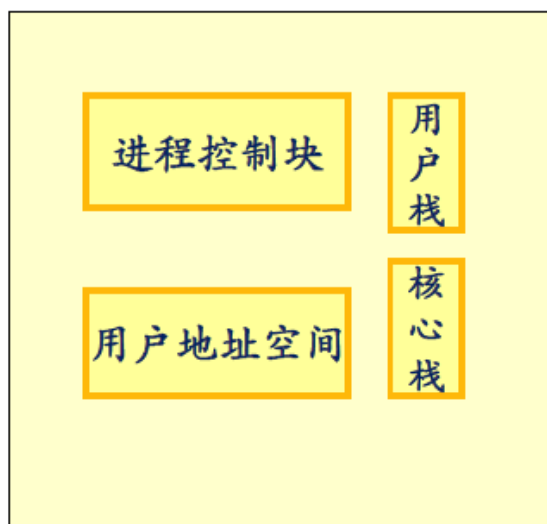
性能的考虑：类比多个CPU；多个线程，有的可以计算，有的可以I/O。

**线程：**线程是进程中的一个运行实体，是CPU的调度单位。又称“轻量级进程”

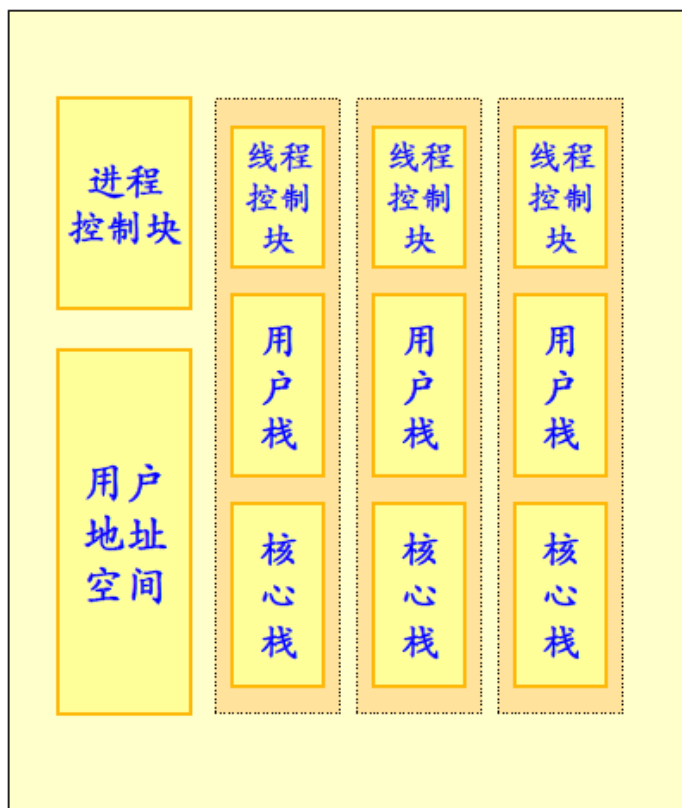
进程是资源的拥有者和调度的单位。

线程有状态和状态转移，有自己的栈和栈指针，但共享所在进程的地址空间和其他资源，不运行时要保存上下文；相同进程内，一个线程可以创建和撤销另一个线程（程序以单线程进程的方式开始运行）。

## 单线程进程模型



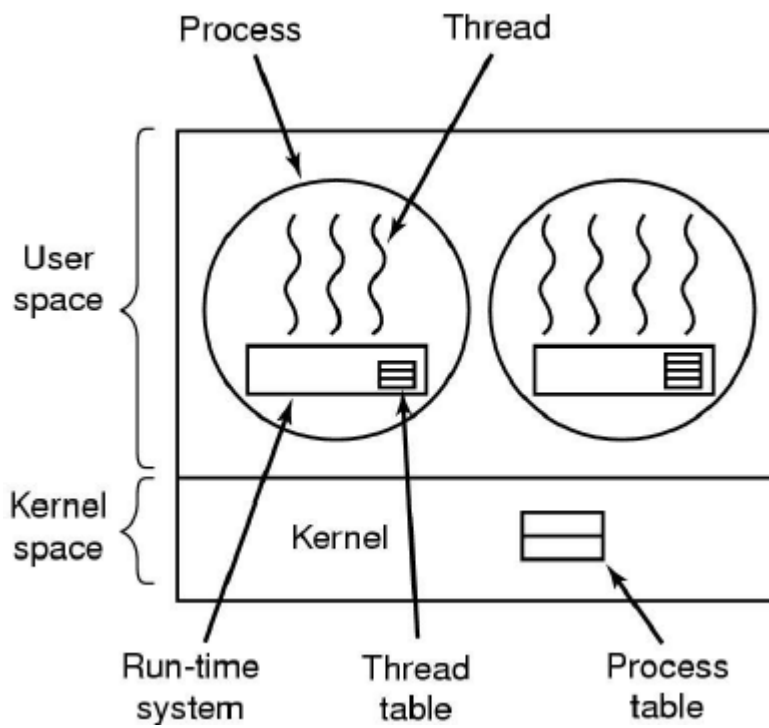
## 多线程进程模型



线程的实现：用户级线程，在用户空间实现；核心级线程，在内核中实现；混合的方法，在内核中实现，支持用户线程。

用户级线程：在用户空间建立线程库，对线程进行管理；内核只管理进程，不知道线程的存在；线程切换在进程里完成，不需要内核态特权。优点：线程切换可能会快；调度算法可由应用程序定制；方便跨OS移植（只要实现线程库即可）。缺点：由于大多数系统调用是阻塞的，因此内核阻塞进程也阻塞了线程中的所有进程；内核只会将CPU分配给进程，同一个进程中的两个线程不可能同时在两个CPU上运行。

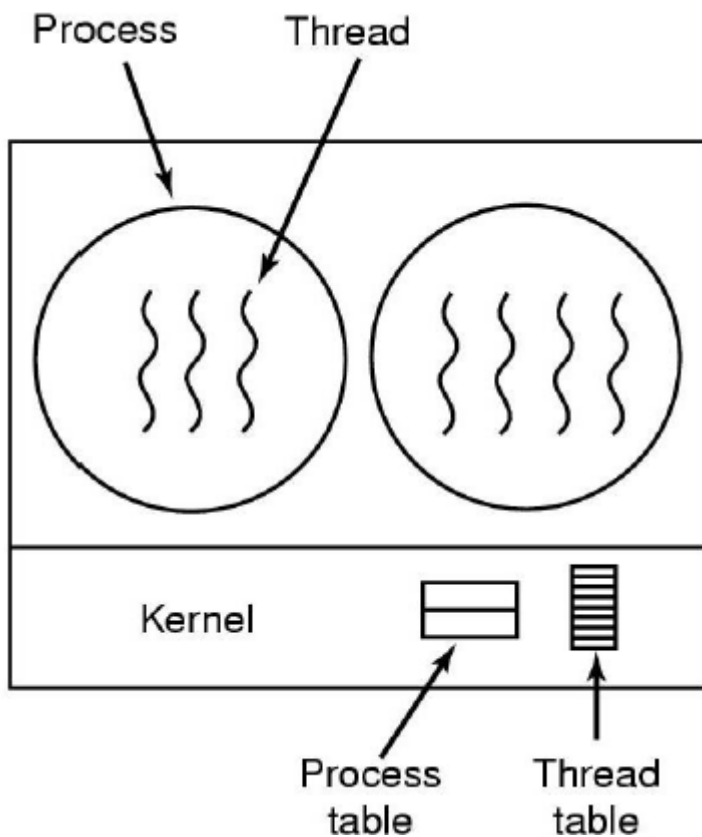




当用户级线程遇到阻塞系统调用时，可以选择修改系统调用为非阻塞的，也可以重新实现对应系统调用的I/O库函数，使之使用非阻塞的系统调用。

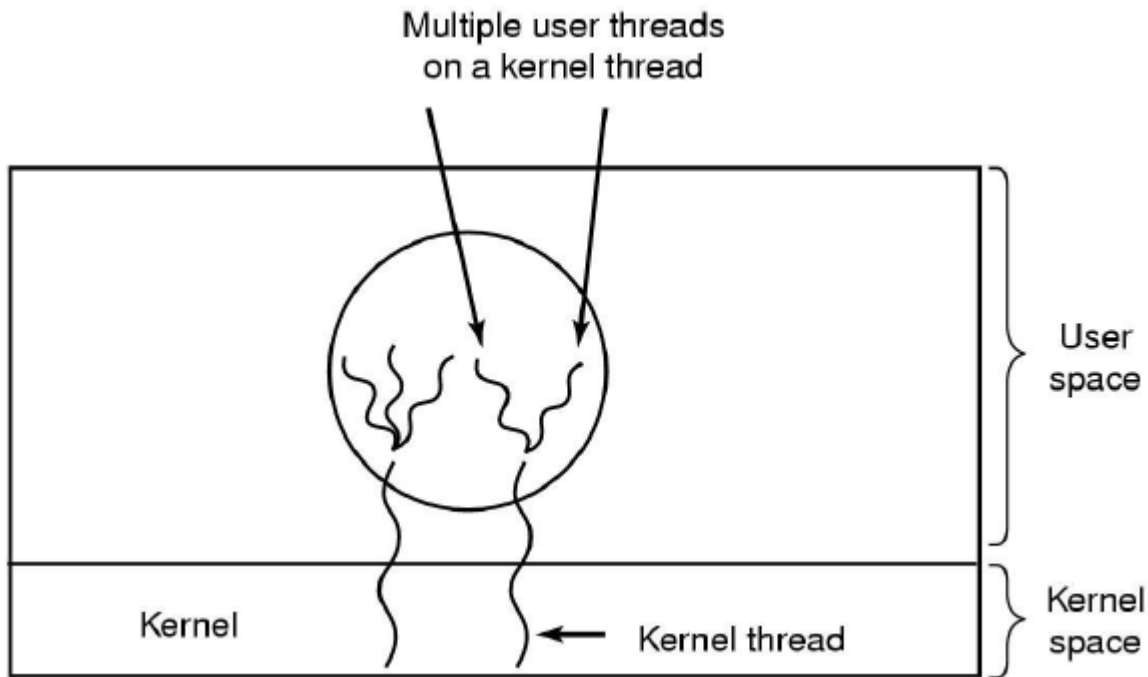
**POSIX线程库，Pthreads:** 用户级线程库的典型示例，提供多线程编程接口，其中有create（创建），exit（撤销），join（等待某一线程结束后再执行），yield（释放CPU让其他线程执行），attr\_init（创建初始化线程属性结构体），attr\_destroy（移除线程属性结构体）

**核心级线程:** 内核管理和维护所有线程，并向应用程序提供API接口；线程切换需要内核支持，以线程为基础进行调度。eg. Windows

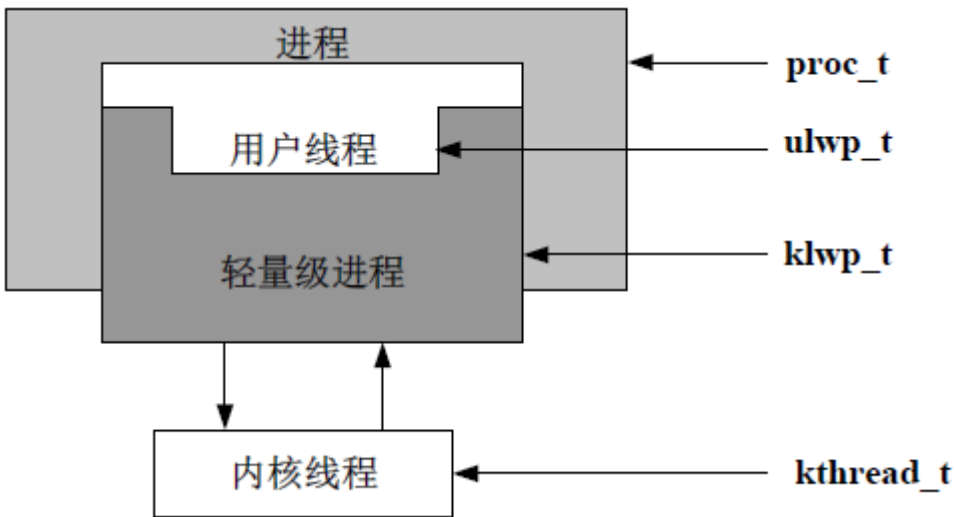


**Windows线程库：**采用核心级线程。Win32 API为内核和应用程序的接口，对内核提供的功能进行了函数封装；MFC用类库的方式对Win32进行封装，将类提供给开发者。

**混合模型：**线程创建在用户空间里完成，线程调度在核心态完成。eg. Solaris



**Solaris进程线程模型：**进程，内核线程，用户线程，LWP轻量级进程。进程是资源分配和管理的单元，内核级线程是内核的调度单元，用户级线程是程序执行在用户态的抽象，LWP把用户线程和内核线程绑定在一起。



对象	定义	名字
进程	一个执行环境——用于执行线程的状态容器	proc_t
用户线程	在进程内为用户提供内核状态的对象	ulpw_t
轻量级线程	为用户线程提供内核状态的对象	klwp_t
内核线程	内核中调度和执行的基本单位	kthread_t

**Windows进程线程模型**

Windows进程对象：属性包括PID，资源访问令牌，进程的基本优先级，默认亲和处理机集合等。

**EPROCESS**：执行体进程块，表示Windows中的Win32进程。内容包括线程块列表，虚拟地址空间描述表，对象句柄列表（当进程创建或打开一个对象时，就会得到一个代表该对象句柄用于访问对象）。

**KPROCESS**：内核进程块，起PCB功能，内核中的进程对象。

**PEB**：进程环境块

EPROCESS和KPROCESS位于内核空间，PEB位于用户空间。



Win32子系统的进程控制系统调用：CreateProcess，ExitProcess和TerminateProcess。

CreateProcess(): 用于创建进程及其主线程，来执行指定的程序。新进程不可以继承优先权类，内存句柄，DLL模块句柄。

ExitProcess(): 种植一个进程和它的所有线程；它的终止是完整的，可以关闭所有对象句柄，结束所有线程等。

TerminateProcess(): 终止指定进程和它的所有线程；它的终止是不完整的（比如不向相关DLL通报关闭情况），通常只用于异常情况下对进程的终止。

Windows的线程：执行体线程块为执行体线程对象，ETHREAD，其中包括内核线程块KTHREAD和指向线程环境块TEB的指针。

ETHREAD和KTHREAD在内核空间中，TEB在用户空间中

Windows线程状态：就绪状态Ready，备用状态Standby，运行状态Running，等待状态Waiting，转换状态Transition，种植状态Terminated，初始化状态Initialized。

Ready: 已经获得了所有运行所需资源，等待CPU执行。

Standby: 特定处理器执行对象，系统中每个处理器上只能有一个Standby线程。

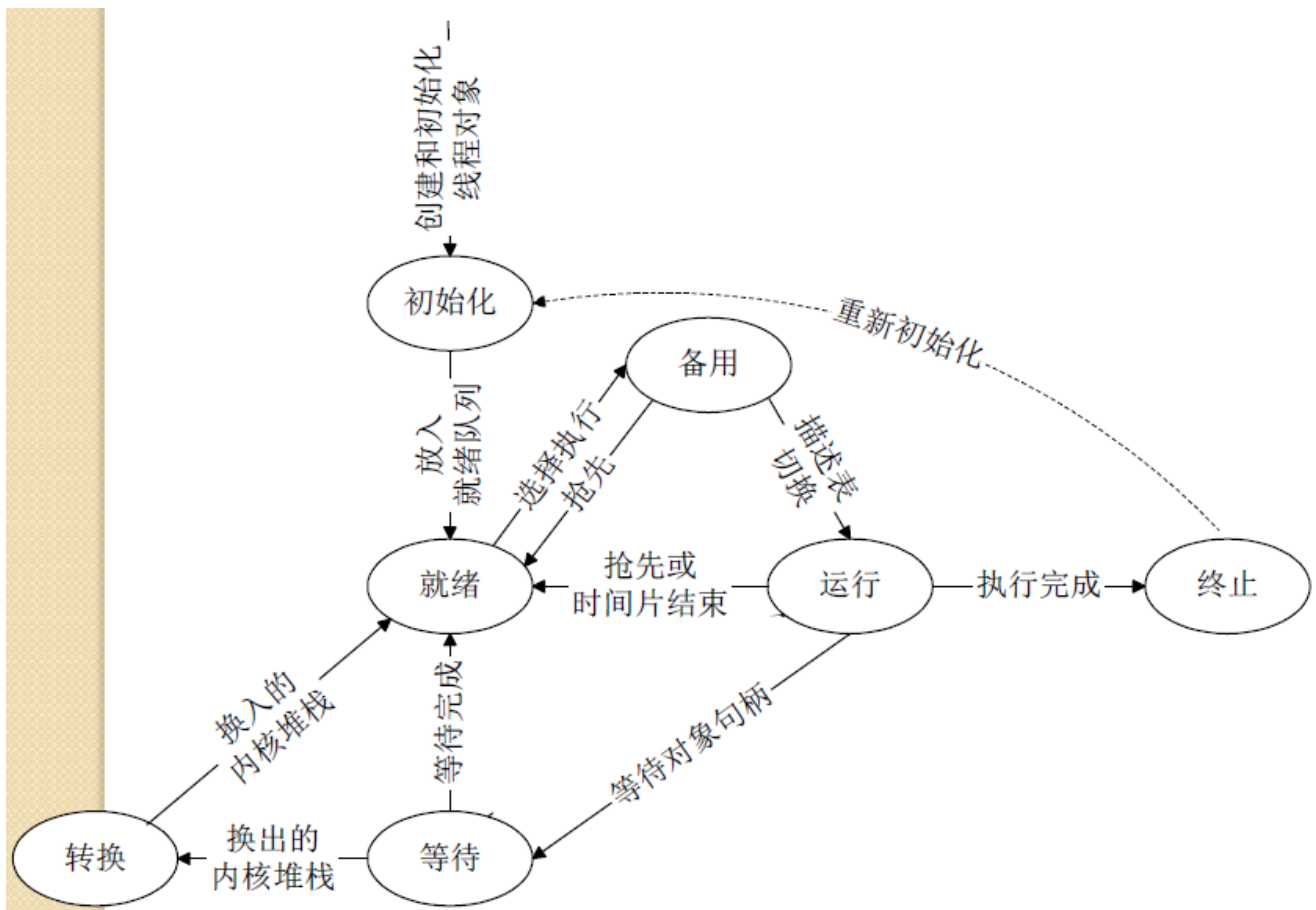
Running: 完成描述表切换，线程进入运行状态。

Waiting: 等待对象句柄以同步它的执行，等待结束时根据优先级进入Running或Ready状态。

Transition: 线程在准备执行而其内核栈被换出内存是进入该状态，当内核栈调回内存，线程进入Ready状态。

Terminated: 执行完毕进入改状态，如果执行体有一个指向线程对象的指针，可以将线程对象重新初始化并再次使用。

Initialized: 创建过程中的中间状态。



Windows线程API:

CreateThread(): 在调用进程的地址空间上创建一个线程以执行指定的函数，返回所创建线程的句柄。

ExitThread(): 结束本线程。

SuspendThread(): 挂起指定线程。

ResumeThread(): 递减指定线程挂起计数，挂起计数为0时，线程恢复执行。

## 4. 线程进程调度

**KEY WORDS:** 调度层次，调度时机，进程切换，调度算法设计原则，抢占和非抢占，时间片，饥饿，调度算法，优先级反转，吞吐量，周转时间，响应时间，Linux调度算法，Windows线程调度

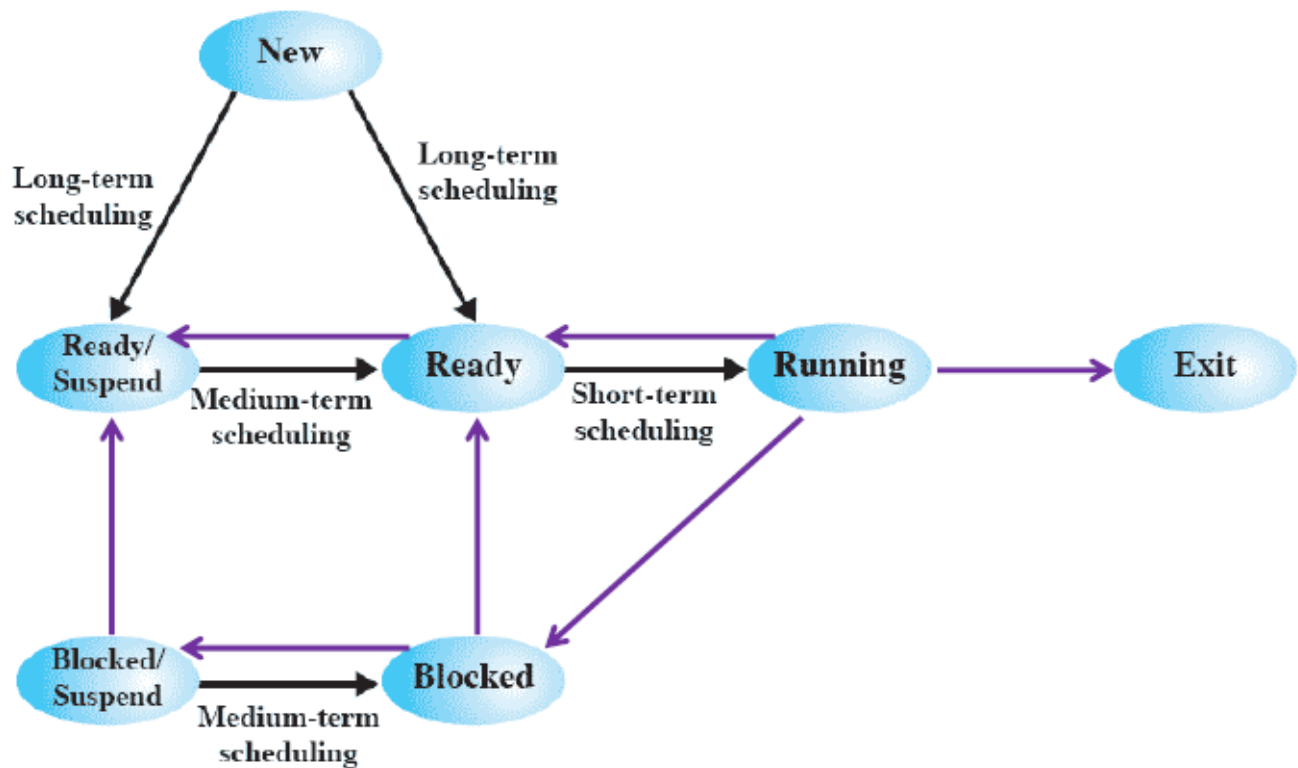
## CPU调度

调度层次：长程调度，中程调度，短程调度。

长程调度：作业调度或宏观调度，指在创建新进程时，决定其是否进入当前活跃进程集合。

中程调度：进程在内外存之间交换，为当前运行进程的执行提供内存。

短程调度（微观调度）：从CPU资源分配的角度，选择就绪进程/线程进入运行状态。由于短程调度时间尺度为毫秒级，因此短程调度算法要求实现时高效。



处理器调度：控制和协调进程对于CPU的竞争。按照一定的调度算法从就绪队列中选择一个进程，把CPU的使用权交给被选中的进程；如果没有就绪进程，那么OS安排一个系统空闲进程或idle进程。

调度程序：挑选就绪进程的内核函数。需要考虑调度算法（依据什么原则挑选进程），调度时机（何时分配CPU），调度过程（如何分配CPU）

调度时机：进程正常终止或因为某种错误终止；新进程创建或一个等待进程变为就绪；进程从运行态进入阻塞态；进程从运行态进入就绪态。

事件发生 → 当前运行的进程暂停运行 → 硬件机制响应 → 进入操作系统，处理相应事件  
→ 某些进程的状态在结束处理后发生变化，也可能又创建了一些新的进程  
→ 就绪队列有调整 → 需要进程调度根据预设调度算法从就绪队列选择进程

进程切换：一个进程让出CPU，由另一个进程占用CPU的过程。需要切换全局页目录来加载一个新的地址空间；需要切换内核栈和硬件上下文来更换进程运行状态。

进程切换的开销：直接开销为内核完成切换所用的CPU时间，具体包括恢复和保存寄存器的时间+切换地址空间的时间；间接开销为进程切换引起的Cache和TLB等失效而产生的开销。

## CPU调度算法

调度算法需要针对不同类型的OS仔细斟酌。交互式进程需要花很多时间等待用户输入输出，同时需要快速响应，要考虑响应时间、均衡性等；批处理进程不必进行用户交互，可以在后台运行，不必快速响应，要考虑吞吐量、周转时间、CPU利用率等；实时进程有实时需求，不应被低优先级的进程阻塞，要求响应时间尽量短，要考虑最后期限、可预测性等。

**吞吐量Throughput:** 单位时间完成的进程数目。

**周转时间TT:** 每个进程从提出请求到运行完成的时间。

**响应时间RT:** 从提出请求到第一次回应的的时间。

**CPU利用率:** CPU进行有效工作的时间比例。

**等待时间:** 每个进程在就绪队列中等待的时间。

**进程优先级（数）:** 静态优先级在进程创建时指定，运行过程中不再改变；动态优先级在进程创建时指定，运行过程中也可以动态变化。

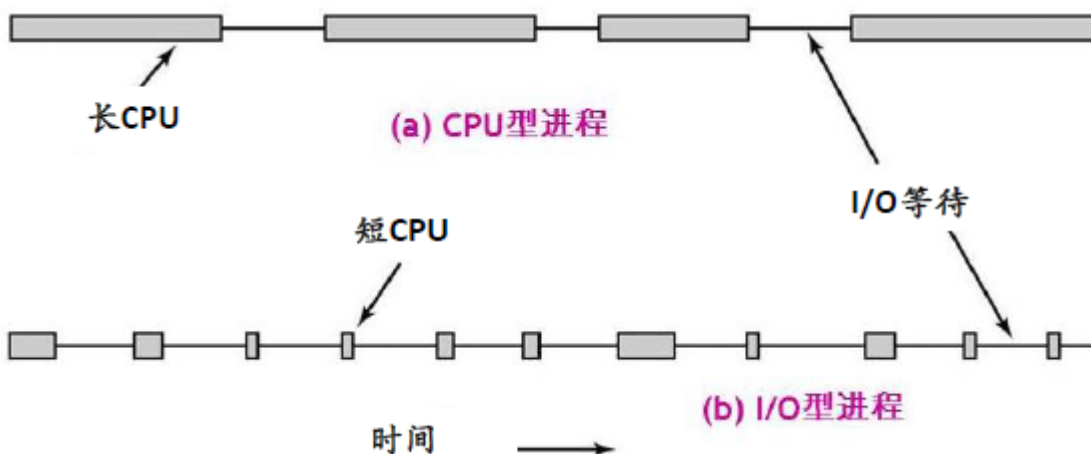
**进程就绪队列的组织形式:** 就绪队列可采用单级也可以采用多级，多级队列中从第一级队列取优先级最高的进程，若为空则取第二级队列优先级最高的进程，...当进程被重新插入就绪队列中时，被插入到下一级就绪队列中（来自最后一级就绪队列的进程依然插入到就绪队列）。

**抢占式Preemptive:** 当有优先级高过正在执行的进程的进程就绪时，OS强行剥夺正在执行的进程的CPU，将其提供给具有更高优先级的进程使用。

**不可抢占式Non-Preemptive:** 进程一旦被调度执行后，除非其自身原因不能再运行，否则将一直运行下去，没有别的进程可以显式地阻止它运行。

**I/O密集型进程:** 频繁地进行I/O，需要花很多时间当代I/O操作完成。

**CPU密集型进程:** 需要大量的CPU时间进行计算。



**时间片Time Slice / Quantum:** 分配给调度上CPU执行的进程，确定了允许该进程运行的时间长度。时间片的大小可以使固定的也可以是可调的，一般而言时间片大小与以下因素相关——进程切换的直接开销，响应时间的要求，就绪进程的个数，CPU能力，进程的行为特点。

**批处理系统中才用的调度算法:** FCFS先来先服务，SJF最短作业优先，SRTN最短剩余时间优先，HRRN最高响应比优先。

**FCFS:** 没有抢占，按照进程进入就绪队列的顺序先后使用CPU。公平简单；长进程调度执行后，短进程需要长时间等待，不利于交互；I/O资源和CPU资源的利用率较低。

**SJF**: 非抢占式，具有最短完成时间的进程优先执行，可以改善FCFS的短作业的TT和具有较多短作业的批作业的平均周转时间。所有进程同时可运行时，**SJF**可以保证最短的平均周转时间；对于长任务不公平，源源不断的短任务会导致长任务饥饿；需要“预测未来”才能知道作业完成时间。

**SRTN**: 抢占式版本的SJF，当一个新就绪的进程比当前运行的进程具有更短的完成时间时，抢占当前进程，调度新就绪的进程执行。SRTN可以保证最短的平均周转时间；对于长任务更加不公平，即使开始执行的长任务还可以被抢占。

**HRRN**: 综合算法，选择响应比最高的进程，抢占或不可抢占均可。更加需要预测未来，是不可能的一种折衷算法。

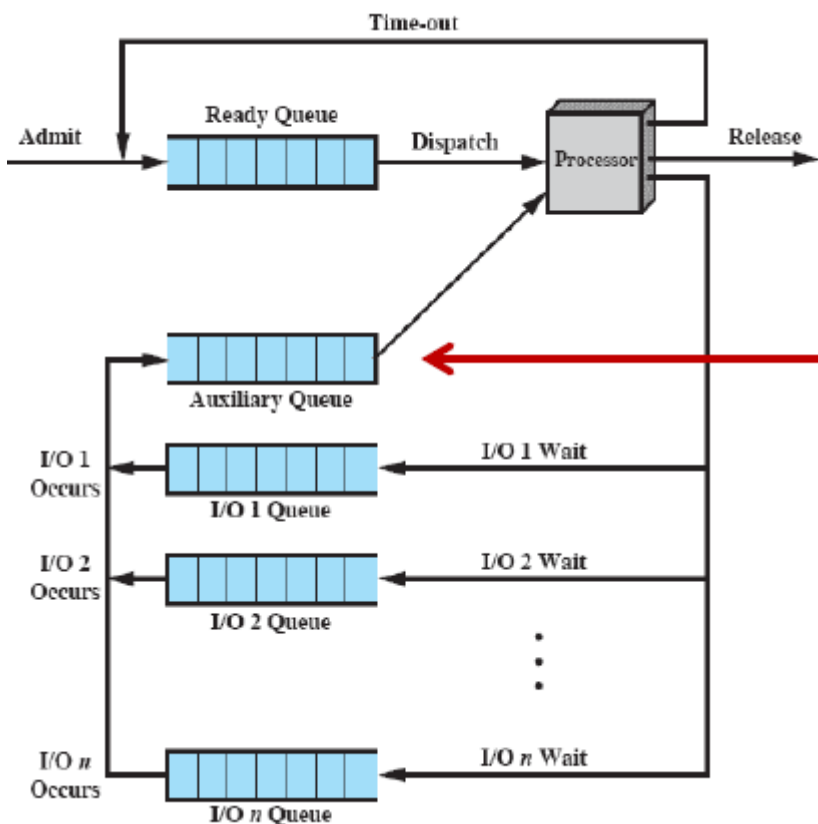
$$R = \frac{\text{作业周转时间}}{\text{作业处理时间}} = \frac{\text{作业处理时间} + \text{作业等待时间}}{\text{作业处理时间}} = 1 + \frac{\text{作业等待时间}}{\text{作业处理时间}}$$

交互式系统中的调度算法：RR轮转调度，HPF优先级调度，Multiple Queues多级队列与Multiple Feedback Queues多级反馈队列，SPN最短进程优先。

**RR**: 周期性地进行任务切换，每个进程分配一个时间片，时钟中断时检查时间片，如果用尽则切换。目的是为短任务改善平均周转时间（相较于FCFS）。公平；有利于交互式程序，响应时间快；由于进程切换较多，RR的开销较高；对与运行时间不同的进程有利但对相同运行时间的进程有害。RR是非抢占的。

RR时间片的选择：时间片过长，大于典型的交互时间，会使得RR退化为FCFS算法，延长了短进程的响应时间；时间片过短，小于典型的交互时间，会频繁切换进程浪费CPU时间。

**Virtual RR**: RR对于I/O型进程不公，因为I/O型进程基本总是用不完时间片，而CPU型进程总是可以用完整个时间片，因此对RR改进如下——进程用完时间片则重新加入就绪队列，如果没有用完时间片就放弃CPU则加入辅助队列，CPU调度优先从辅助队列中调度进程，当辅助队列为空才会调度就绪队列中的进程。





**HPF最高优先级调度算法：**选择最高优先级的进程投入运行，一般而言系统进程优先级高于用户进程，前台进程高于后台进程，这是由于交互型OS更偏好于I/O型进程。优先级可以使静态的，也可以是动态调整的。实现简单，不公平，会导致饥饿。

**优先级反转：**在基于优先级的抢占式系统中，一个低优先级进程持有一个高优先级进程所需要的资源，使得高优先级进程等待低优先级进程运行。考虑三个进程H，M，L优先级递降，起初H和M在等待事件到来，L执行进入临界区；之后被H的事件到来，转为就绪态，抢占L的CPU；由于H也需要进入临界区，失败后被阻塞等待；M的事件到来，上CPU执行使得L无法执行，进一步使得H也无法执行。如此错误使得高优先级进程停滞，导致系统性能降低甚至是系统错误。

**优先级反转的解决方法：****优先级天花板，**当任务申请某资源时，将该任务的优先级提升到可以访问这个资源的所有任务中的最高优先级，该优先级即为该资源的优先级天花板，在释放资源时恢复优先级，该方法逻辑简单；**优先级继承，**当高优先级任务申请共享资源时，发现低优先级任务正在使用，则将低优先级任务的优先级提升至自身的优先级，当原先的低优先级任务释放资源时再恢复，该方法逻辑繁复；**中断禁止，**禁止所有中断以保护临界区。

**多级队列调度算法：**使用多个就绪队列进行调度。需要考虑按什么规则分队列，怎样排队，那些进程优先级高，各个队列的调度策略是否相同。

**多级反馈队列调度算法：**设置多个就绪队列，各级就绪队列优先级递降，时间片递增，各级队列按照RR的方式进行调度；当一个进程用完时间片时，进入下一级就绪队列（来自最后一级就绪队列的进程依然进入最后一级就绪队列）；由于阻塞而放弃CPU的进程进入对应的等待队列，事件一旦发生则回到原来一级的就绪队列（需要考虑回到队列的什么位置，以及时间片如何设置）。MFQ是非抢占的。

**保证调度：**向用户做出明确的性能保证，然后去实现这一保证。做出保证的方法是跟踪各个进程自创建以来使用了多少CPU时间，然后除以n得到各个进程应获得的CPU时间。根据以使用的CPU时间和应获得的CPU时间比率进行调度，有效调度比率低的进程（比率低说明对其不公平）。

**彩票调度：**OS为进程提供各种系统资源的彩票，一旦需要进行调度决策时就进行一次摇奖，拥有获奖彩票的进程获得这一资源；重要的进程可以获得更多的彩票，并且彩票可以进行转手来实现优先级变更。

**公平分享调度：**按照用户设置时间上限，当用户达到时间上限时，其拥有的进程都不可再被调度执行。

**调度机制和调度策略分离的原则：**将调度算法以某种形式参数化，而这些参数可以由用户进程进行填写。比如内核采用优先级调度算法并提供了一个设置优先级的系统调用，那么父进程尽管不参与进程调度，也可以控制子进程的调度细节，调度机制由内核决定并存在于内核之中，而调度策略由用户进程决定。

## 线程调度

**用户级线程调度：**内核并不知晓用户线程的存在，它只对用户进程进行调度，当用户进程被调度执行后，进程内的线程调度器会对线程进行调度。由于多道线程中并不存在时钟中断，因而线程可以任意长时间地运行，当然也可以利用其它方法实现对长时间运行的线程的“中断”，从而实现RR或HPF的用户线程调度。

**内核级线程调度：**内核明确直到内核线程的存在，调度时其选择一个线程运行（可以考虑其所属进程也可以不考虑），赋予该线程一个时间片，并在耗尽时间片后挂起该线程。由于内核进行一个不同进程间的线程切换所需的开销，大于在进程内的线程切换所需的开销，并且内核完全知晓各个内核线程所属的进程，因此内核倾向于优先调度来自同一个进程的线程。

## Windows线程调度

Windows的调度单位是线程，采用基于动态优先级的、抢占式调度，并结合时间配额进行调整。就绪线程按照优先级进入相应的队列，系统总是选择优先级最高的就绪线程执行，同一优先级的各个线程按照RR进行调度，多处理器系统允许多线程并行。

**引发线程调度的条件：**一个线程的优先级发生改变，一个县城改变了它的亲和处理机集合。



**Windows线程优先级：**32个线程优先级，分为三类。

实时优先级：最高，16-31，处于该优先级的线程优先级不变。

可变优先级：1-15，可以范围内升降，有基本优先级和当前优先级。

系统线程优先级：0，比如零页线程（对系统中空闲物理页面清零）。

所有线程都运行在最低的两个中断优先级，线程优先级0-31对应中断优先级0和1；用户态线程运行在中断优先级0，内核态一部过程调用运行在中断优先级1。

**Windows线程的时间配额Quantum：**不是一个时间长度值，而是一个代表了CPU时间的整数，单位是配额单位。通过调整时间配额，可以在不改变优先级的情况下让线程多获得CPU时间，从而使得其可以更快地执行；如果一个线程用完了时间配额并且没有其他相同优先级的线程，则该线程重新分配到一个新的时间配额并继续执行。

Windows中维护一个就绪位图，每一位表示一个优先级队列中是否有就绪线程等待运行；Windows中维护一个空闲位图，每一位表示一个处理器是否处于空闲状态。

**Windows线程调度：**主动切换，抢占，时间配额耗尽

抢占：线程被抢占时，被放回相应优先级的就绪队列的队首。实时优先级线程被抢占时，时间配额重置为完整的时间配额；可变优先级线程被抢占是，时间配额保持剩余的时间配额不变。

时间配额耗尽：如果当前线程的优先级没有降低，且该优先级队列中有就绪线程，则选择下一线程，当前线程回到就绪队列队尾；如果当前线程的优先级没有降低，且该优先级队列中没有就绪线程；则OS给当前线程重新分配时间配额，使之继续运行；若A的优先级降低了，则选择一个更高优先级的线程运行。

线程优先级提升和时间配额变动：针对可变优先级范围内的线程，解决不公平带来的饥饿和改善I/O线程等。

I/O操作完成时，Windows临时提升等待该操作线程的优先级，一般相应要求越高优先级提升幅度越大；为了避免不公平性，I/O操作完成唤醒该线程时，时间配额需要减小（-1）。

等待事件和信号量完成后，优先级提升一个等级且提升后的等级不超过15，时间配额-1；在高优先级执行完后降低一个优先级，运行一个新的时间配额直至降低至基本优先级。

前台线程等待结束后，提升优先级；用户不可以禁止这种优先级提升。

窗口线程被唤醒后提升两个优先级，改善交互应用的响应时间。

“饥饿”线程（排队超过300各时钟中断间隔）被平衡集管理器发现，优先级提升至15，并获得4倍时间配额；当这各时间配额耗尽后，优先级立刻衰减到基本优先级。

空闲线程：优先级为0，只有在没有可运行线程时才会被调度执行，每个处理机有一个对应的空闲线程。空闲线程进行基本的检测等工作。

## Linux进程调度

**实时进程：**对调度延迟要求非常高，要求尽量快速响应并执行。调度策略一般为FCFS或RR。

**普通进程：**目前采用CFS，可以根据行为特点分为交互式进程和批处理进程。

**交互式进程：**间或处于睡眠等待，对响应速度要求比较高。

**批处理进程：**在后台执行，可以忍受相应延迟。

**Linux2.4调度器：**RunQueue+时间片counter+优先级

实时进程的优先级始终小于普通进程，是静态优先级，并且实时进程的counter用完后立刻重置并放入就绪队列；普通进程的优先级是动态优先级，由 $\text{counter} + \text{nice}$ （静态优先级）确定，当普通进程的counter用完后会插入到就绪队列中但不立刻重置，需要等待统一重新计算。睡眠状态（常常是交互式进程）的进程counter没有用完，在重新计算时会加上原来的没有用完的部分从而提高了优先级，系统如此便可以提高交互式进程的响应速度。

Linux 2.4调度器可扩展性不好， $O(n)$ 的计算counter的复杂度，对于交互式进程的优化不足，对于实时进程的支持不足。

**O(1)调度器：**修改Linux 2.4调度器，改进进程优先级的计算方法，提出了pick next算法。

优先级计算：对于普通进程来说通过下式计算动态优先级，期中bonus取决于平均睡眠时间，平均睡眠时间越长，bonus值越大。

$$\text{动态优先级} = \max\{100, \min\{\text{静态优先级} - \text{bonus} + 5, 139\}\}$$

因此可以对上式进行变形来得到一个不等式判断进程是否是一个交互式进程

$$\text{动态优先级} \leq 3 \times \text{静态优先级} / 4 + 28$$

实时进程的优先级是静态的，不会动态修改并且总是比普通进程的优先级高。

**pick next算法：**维护两个进程队列数组——active和expired，数组中每个元素保存某个优先级的进程队列指针，每次从active数组中找到最高优先级队列中的第一个进程；当进程时间片为0时，若判断该进程为交互式进程或实时进程则重置时间片并插入active数组，否则重置时间片并插入到expired数组，当进程占用CPU超过一个阈值后，即使是实时进程或交互式进程也会被插入到expired数组；当active数组中所有进程都被移到expired数组中后，交换active和expired数组。

O(1)调度器对于交互式进程和批处理进程的区分算法有了很大改进，但仍然会失效；代码计算动态优先级的经验公式很难理解，代码很难阅读维护。

**SD调度器：**楼梯调度（SD）抛弃了动态优先级的概念，而采用了一种完全公平的思路。

**SD算法：**采用FCFS或RR的调度策略不变；SD算法为每个优先级维护一个列表并将其组织在active数组之中，选取调度执行的进程时，从active数组中直接选取；当用完时间片后，会被加入到active数组的低一级列表中，如此“下优先级楼梯”，当进程下到最后一级台阶时，会再回到初始优先级的下一级任务列表中。

可以避免低优先级进程饥饿，因为高优先级进程会最终与低优先级进程竞争，使之获得执行机会；交互式进程进入睡眠后，它的同等优先级一个一个下楼梯，但它被唤醒时还在高处的楼梯，因此可以优先执行，加快响应。

**RSDL调度器：**对SD算法的改进，重新引进expired数组，每个优先级都分配了组时间配额 $T_g$ ，同一优先级的每个进程都拥有同样的时间配额 $T_p$ 。

**Minor rotation ( $T_p$ )：**进程用完自身的 $T_p$ 后，进入active数组下降一级的队列。

expired：进程时间片用完后，进入expired数组中其初始台阶。

$T_g$ ：高优先级进程组用完了 $T_g$ ，所有该组的进程都被强制降低到下一级台阶，如此便保证了低优先级任务可以在一个可预期的时间里得到调度。

**Major rotation：**当active空，或所有进程都降低到最低优先级时，交换active数组和expired数组，使得所有进程都恢复到初始状态。

$T_p/T_g$ 太小依然会导致饥饿，解决方法是协调好 $T_p/T_g$ 和时间片的关系，在一开始就创建一个不会被分配但优先级更低的空优先级。

**CFS调度器**：完全公平的调度器，不追踪进程的睡眠时间，也不试图区分交互式进程。通过vruntime来表示进程运行了多少时间，vruntime与实际运行时间成正比，与优先级成反比，每次选择vruntime最小的进程来运行。

核心思想：根据进程的优先级按比例分配运行时间。总的调度周期是确定的。进程权重根据NICE值确定，可以通过查表（数组）得到不同NICE的权重。

$$\text{分配给进程的运行时间} = \text{调度周期} \times \text{进程权重} / \text{所有进程权重之和}$$

**vruntime**：其大小表示实际运行时间的大小。

$$\begin{aligned} \text{vruntime} &= (\text{调度周期} \times \text{进程权重} / \text{所有进程权重之和}) \times \text{NICE\_0\_LOAD} / \text{进程权重} \\ &= \text{调度周期} \times \text{NICE\_0\_LOAD} / \text{所有进程权重之和} \end{aligned}$$

数据结构：通过以vruntime为顺序的RB-tree来插入或删除任务，因为每次都会从左边取进程，右边加入进程，所以需要使用平衡的二叉树结构。

新进程的vruntime：不是0，通过当前最小的vruntime确定，一般为min\_vruntime+Δ；父子进程交换vruntime当且仅当OS设定为子进程优先执行，父子进程在同一个CPU，父进程不为空，父进程vruntime<子进程vruntime。

苏醒的睡眠进程的vruntime：取该进程vruntime和min\_vruntime-Δ中的较大值作为该进程的vruntime。

在不同CPU间迁移的vruntime：不会保持不变，每个CPU有自己独立的vruntime RB-tree，因此迁移后要使得该进程vruntime相对于min\_vruntime的差保持不变。

## 多处理器调度

**SMP（对称多处理器）调度**：每个处理器运行自己的调度程序，调度程序对共享资源的访问进行同步。

多处理器调度算法不仅要决定那一个进程执行，还要决定在哪一个CPU上执行；此外由于在CPU间迁移进程存在开销，因此要尽可能得在同一个CPU上执行；但尽量不切换CPU又会导致负载不均，还需要考虑负载均衡问题。

静态进程分配：进程从开始到结束被分配在一个固定的CPU上，每个CPU都有自己的独立的就绪队列，调度靠小小但负载不均。

动态进程分配：进程在执行过程中可以分配到任意空闲的CPU执行，所有CPU共享一个就绪队列，调度开销大但负载均衡。

## 5.进程同步互斥机制

**KEY WORDS**：临界区，进程互斥，进程同步，锁，信号量，PV操作，生产者消费者问题，读者写者问题，管程，条件变量，wait / signal，Hoare管程，Pthreads

### 进程并发执行

顺序环境：在计算机环境中只有一个程序在运行，该程序独占系统中所有资源，执行不受外界影响。顺序环境的特征为：程序执行封闭，程序执行结果确定，调度顺序无关。

并发环境：程序执行过程不确定且不可再现，并发环境下程序间断执行，资源共享，程序独立执行但程序执行相互制约，程序和计算不再一一对应。

竞争条件：多个进程读写共享数据，最终的结果取决于他们的指令执行顺序。

进程互斥：由于各个进程要求使用共享资源，而这些资源要求排他性使用，各个进程之间竞争这些排他性资源的关系称为进程互斥。

临界资源**critical resource**：系统中一次只允许一个进程使用的资源称为临界资源或互斥资源或共享资源。

**临界区critical section:** 各个进程中对某个临界资源实施操作的程序片段。

临界区的使用原则是：没有进程在临界区，想进入临界区的程序可以进入；两个程序不可以同时进入临界区中；临界区外运行的进程不得阻塞其他进程进入临界区；不得使进程无限期等待进入临界区。


**进程同步:** 系统中多个进程中发生的事件存在时序关系，需要相互合作共同完成一项任务。

实现进程互斥的软件解法：Dekker解法，Peterson解法。

After you问题：两个进程相互等待对方，产生死锁。


P:

```
... ..  
pturn = true;  
while (qturn) ;  
    临界区  
pturn = false;  
... ..
```

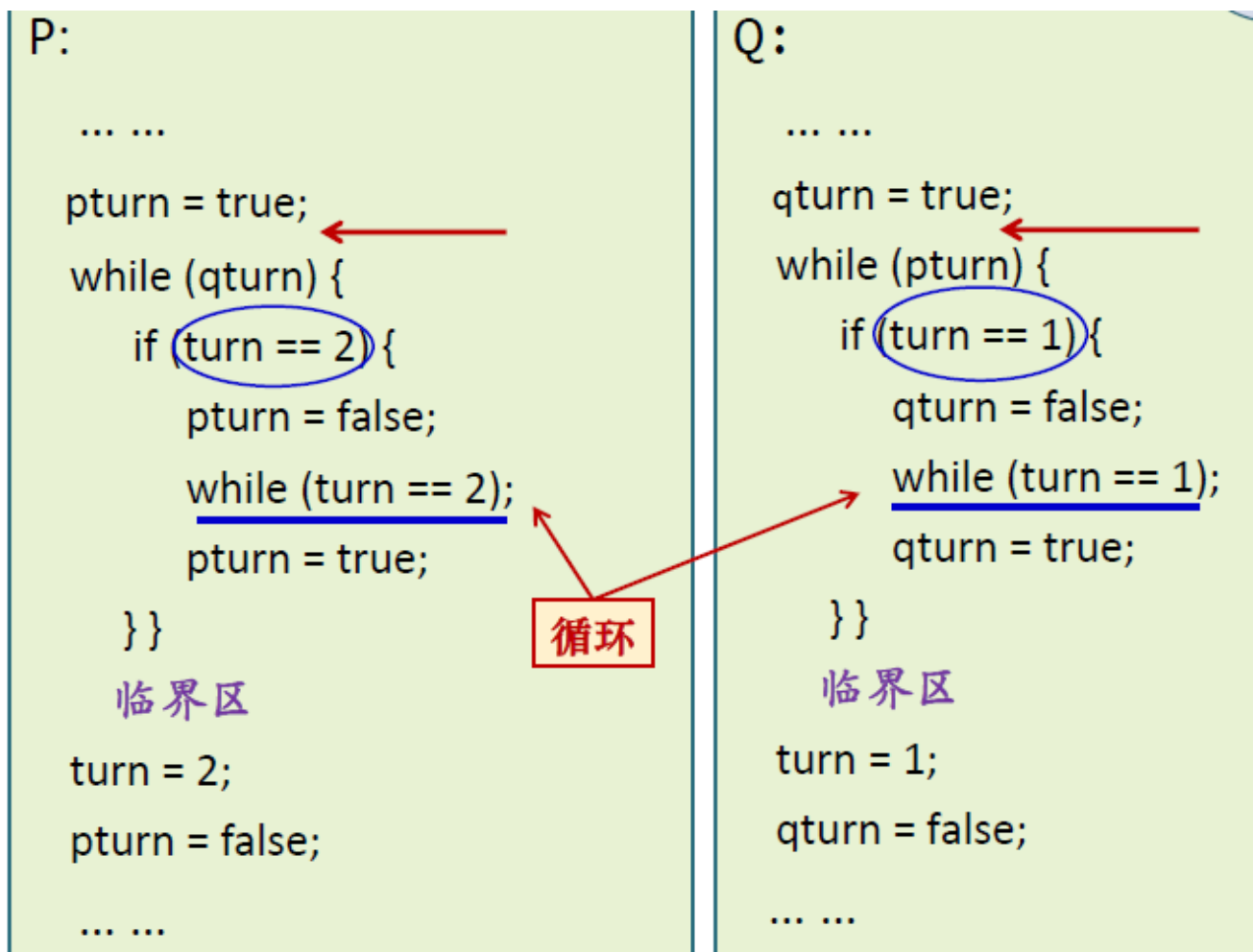


Q:

```
... ..  
qturn = true;  
while (pturn) ;  
    临界区  
qturn = false;  
... ..
```



**Dekker算法:** After you问题是因为两个进程进入循环等待彼此才引起的，因此可以引入一个turn变量，使得确定的一方“真的请对方进入临界区”，，从而实现强制轮流，从而化解这种死锁。



**Peterson算法：** 使用interested数组表示某个进程是否希望进入临界区，turn变量表示有权进入临界区的线程/进程号。在尝试进入临界区时，首先将自己的interested位置为true，之后将turn设置为自己，循环检查turn是否为自己且另一线程/进程的interested位为true，若是则继续循环。如此便可使用忙等待解决互斥访问，并且克服了强制轮流的缺点，使得进入临界区可以是随机的。

```

#define FALSE 0
#define TRUE 1
#define N      2    // 进程的个数
int turn;          // 轮到谁?
int interested[N];
// 兴趣数组, 初始值均为FALSE

void enter_region ( int process)
    // process = 0 或 1
{
    int other;
    // 另外一个进程的进程号
    other = 1 - process;
    interested[process] = TRUE;
    // 表明本进程感兴趣
    turn = process;
    // 设置标志位
    while( turn == process &&
interested[other] == TRUE);
}

```

循环

```

void leave_region ( int process)
{
    interested[process] = FALSE;
    // 本进程已离开临界区
}

```

进程i:

```

... ..
enter_region ( i);
    临界区
leave_region ( i);
... ..

```

Peterson算法解决了互斥访问的问题, 而且克服了强制轮流法的缺点, 可以完全正常地工作 (1981)

实现进程互斥的硬件解法: 中断屏蔽方法, 测试并加锁指令

**中断屏蔽方法:** 首先执行关中断指令, 然后进行临界区操作, 最后执行开中断指令。该方法简单高效, 但代价很高 (临界区不能大), 不适用于多CPU系统, 只适用于操作系统本身, 不适用于用户进程。

**“测试并加锁”指令:** TSL指令, TEST AND SET LOCK, 复制锁到寄存器并将锁置为1; 之后判断寄存器内容即可, 若不是0则说明锁已经被获取, 继续循环TSL指令; 若是0说明已经获取了锁, 可以进入安全区; 出临界区后将锁置为0即可。

## 信号量和PV操作

**信号量Semaphore:** 一个用于进程间传递信号的整数值特殊变量, 由Dijkstra提出。信号量内部包含一个整型的值count和一个用于等待的队列queue。

**P操作:** 原语操作, 将count减1, 若count小于0则说明不能访问临界区, 将进程插入queue队尾; 否则进入临界区。

**V操作:** 原语操作, 将count加1, 若count小于等于0说明有进程在等待访问临界区, 从队首唤醒一个进程; 否则直接出临界区。

用PV操作解决进程互斥问题: 分析并发进程的关键活动, 设置临界区, 设置信号量mutex, 初值为1, 在临界区前P, 出临界区后V即可。

## 管程

信号量机制存在的不足: 程序编写困难复杂, 效率低。

**管程定义：**管程是一个特殊的模块，由关于共享资源的的数据结构及其上的一组过程组成。进程只能通过调用管程中的过程来间接地访问管程中的数据结构。

管程必须是互斥进入的，这是为了保证其中的数据结构的数据完整性。

管程中可以设置条件变量和等待/唤醒操作来解决同步问题。

多个进程在管程中的情况：Q进程进入管程并执行等待操作，释放管程的互斥权；在Q后面进入的P进程执行唤醒操作（P唤醒Q），则此时会有两个进程活动在管程之中。解决方法：**Hoare**，P等待Q执行；**MESA**，Q等待P继续执行；**Hansen**，并发pascal，规定唤醒为管程中最后一个可执行的操作，保证P唤醒Q之后一定立刻出管程。

**条件变量：**在管程内部说明和使用的一种特殊类型的变量。具有wait和signal操作。

**wait操作：**如果紧急队列非空，则唤醒第一个等待者；否则释放管程的互斥权；执行wait操作的进程进入queue尾部等待。

**signal操作：**如果queue为空，相当于空操作；否则唤醒第一个等待者，执行signal操作的进程进入紧急等待队列的尾部。

**锁（互斥量）：**一个锁只有两种状态，加锁或解锁，值为0表示解锁，值为1表示加锁。

**加锁操作：**mutex\_lock，lock或acquire。Pthreads中使用Pthread\_mutex\_lock。

**解锁操作：**mutex\_unlock，unlock或release。Pthreads中使用Pthread\_mutex\_unlock。

**Hoare管程：**入口外设置入口等待队列，管程内设置紧急等待队列。当进程试图进入已经被占用的管程时，在入口等待队列等待；当进程被唤醒时，可能会出现“递归的唤醒”，需要在管程内的紧急等待队列等待以保证顺序正确；紧急等待队列优先级高于入口等待队列。

实现管程的途径：直接构造，效率高；间接构造，使用某种已经实现了的同步机制去构造（比如信号量和PV操作）。

Hoare管程的signal存在缺陷：需要进行两次额外的进程切换；是否会将条件队列中的进程永久刮起。

**Mesa管程：**解决Hoare管程中signal存在的缺陷，将其改为notify。

**Notify原语：**当管程中的进行执行Notify操作时，使得条件队列收到通知，而发信号的进程继续执行。最终位于条件队列头的进程在未来的合适时间并且CPU可用时会恢复执行。

由于不能保证接收Notify信号的进程之前没有别的进程进入管程，必须对齐重新检查条件，因此用while循环判断替换if判断。这导致了至少多一次的额外的检测，但不再有额外的进程切换；此外，会存在进程每次判断条件时都不能通过，而被无限期推迟导致饥饿的情况。

改进的Notify操作：给每个条件原语关联一个监视计时器，不论条件是否被通知，一个等待时间超时的进程就被设置为就绪，该进程被调度时就会再次检查条件，如果满足就继续执行。这样改进可以避免饥饿的情况。

**Broadcast原语：**使得所有在该条件上等待的进程都进入就绪状态。

**Mesa管程相比于Hoare管程，出错更少。**

PTHREADS中的锁和条件变量

Pthreads中使用Pthread\_mutex\_init创建互斥量并赋初值为0或1，Pthread\_mutex\_destroy撤销已有的互斥量，Pthread\_mutex\_trylock尝试加锁，如果不能获取锁则返回failure而不是进入blocked状态。

Pthreads中的条件变量：Pthread\_cond\_init和Pthread\_cond\_destroy创建和撤销条件变量；Pthread\_cond\_wait和Pthread\_cond\_signal进行wait操作和signal操作；Pthread\_cond\_broadcast进行广播操作，唤醒所有等待的进程。

Pthread\_cond\_wait操作：首先解锁，之后等待直至收到解除等待的信号，最后立即上锁。

## 锁的实现

可以通过开关中断方法实现。

```
lock()
{
    disable interrupts
    while (value != FREE) ;
    value = BUSY
    enable interrupts
}
```

可以通过频繁快关中断的方式，使得锁不忙等待。

```
lock()
{
    disable interrupts
    while (value != FREE)
    {
        enable interrupts
        disable interrupts
    }
    value = BUSY
    enable interrupts
}
```

可以通过TSL指令实现锁，用户空间内即可实现，线程包就可以使用这一方法。

## 进程通信

在多处理器情况下，原语失效，因此必须进行进程通信。

基本的进程通信方式：消息传递，共享内存，管道，套接字，RPC远程过程调用

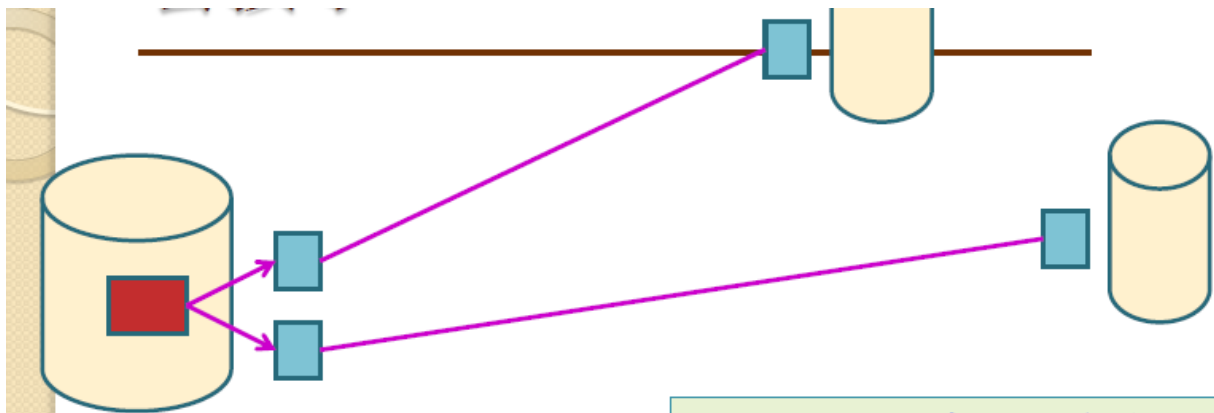
**消息传递：**发送进程S使用系统调用陷入内核，内核复制消息至消息缓冲区，接收进程R的PCB中的消息队列指针接收到消息信号，使用系统调用复制消息至其用户空间。

**共享内存：**相互通信的内存建立公共的内存区域，有的向该区域写，有的从该区域读。问题在于如何建立公共的内存区域，即进程地址空间独立，如何改变这种关系。

**管道：**利用缓冲传输介质（内存或文件）联结两个相互通信的进程，按照字符流的方式写入和读出，先进先出。

**套接字：**服务器/客户端模型。





服务器：创建一个套接字，并将其与本地地址/端口号绑定；监听；当捕获到一个连接请求后，接受并生成一个新的套接字，调用 `recv()/send()` 与客户端通信，最后关闭新建的套接字

客户端：为请求服务也创建一个套接字，并将其与本地地址/端口号绑定；指定服务器端的地址和端口号，并发出套接字连接请求；当请求被接受后，调用 `recv()/send()` 与服务器通信，最后关闭套接字连接