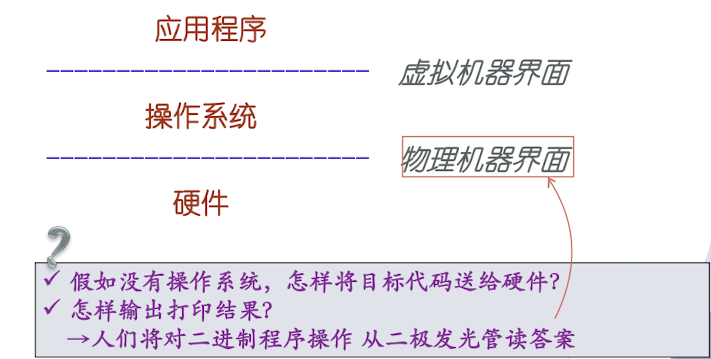
# Lecture0 课程介绍

## helloworld程序的执行

1. 用户告诉操作系统执行helloworld程序
2. 操作系统找到helloworld程序的相关信息，检查其类型是否是可执行文件；并通过程序首部信息，确定代码和数据在可执行文件中的位置，计算出对应的磁盘块地址
3. 操作系统之后创建一个新的进程，并将helloworld可执行文件映射到该进程结构（完成进程虚拟地址空间的初始化），表示由该进程执行helloworld程序
4. 完成进程初始化后，操作系统将该进程加入到某种形式的就绪队列中，运行调度程序，进行进程调度
5. 若调度程序选中hello程序，操作系统为helloworld程序设置CPU上下文环境，并跳转到程序开始处
6. CPU执行helloworld程序的第一条指令，MMU进行地址转换时发生缺页异常（PTE无效）
7. 操作系统为该进程分配一页物理内存，并将代码从磁盘读入内存，然后继续执行helloworld程序
8. CPU重新执行引发缺页异常的指令，之后helloworld程序执行puts函数（系统调用），在显示器上写一字符串
9. 操作系统找到要将字符串送往的显示设备，通常设备是由一个进程控制的，所以操作系统将要写的字符串送给该进程
10. 控制设备的进程告诉设备的窗口系统它要显示字符串，窗口系统确定这是一个合法的操作，然后将字符串转换成像素，将像素写入设备的存储映像区
11. 视频硬件将像素转换成一组显示器可接收的控制/数据信号；显示器解释信号，激发液晶屏，我们在屏幕上看到了“hello world”

## 操作系统的地位

## 操作系统的主要工作

1. 程序的执行
   1. 启动程序、执行程序以及程序结束的工作
2. 完成个性的工作（为硬件提供抽象）
   1. 在各种硬件之上为用户提供一个抽象层，使用户不用直接和复杂、不一致的物理机器界面交互
   2. 增强了硬件的可移植性，将有限的硬件资源抽象为接近无限的虚拟资源（如对CPU和物理内存的抽象）
   3. 为硬件提供保护
   4. 实现该工作的过程代码（驱动程序）和硬件因素密切相关，即需要设置与测试、使用物理地址、设备接口寄存器等等

* 复杂繁琐、代码量大
* 代码不通用

1. 完成共性的工作（为用户提供服务）
   1. 任何一个程序都需要的、最基本的工作。这些工作具有共性，运行过程相同，且与具体应用无直接关系（与用户所关心的应用目标无直接关系）
   2. 易于使用，基本服务，统一性

# Lecture1 操作系统概述

## 操作系统的定义、作用、设计目标与特征

1. 定义

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

* 1. 进程的角度：操作系统由一些可同时独立运行的进程和一个对这些进程进行协调的核心组成
  2. 资源管理的角度：是资源的管理者，有效、合理的管理资源。能以尽量有效、合理的方式（如资源分配策略与算法）组织和管理计算机的软硬件资源，实现资源在时间和空间上的共享、提高资源利用率
  3. 用户的角度：是服务的提供者，便于用户使用。合理地组织计算机的工作流程，控制程序的执行并向用户提供各种服务功能，使得用户能够灵活、方便的使用计算机，使整个计算机系统能高效地运行
  4. 虚机器的角度：可以将操作系统看作一个分层结构，每一层完成其特定构成一个虚机器，对上一层提供支持；通过逐层功能扩充，最终完成整个操作系统虚机器，而操作系统虚机器向用户提供各种功能，完成用户请求

1. 作用
   1. 虚拟化：操作系统将物理资源(如处理器、内存或磁盘)，转换成更通用、更强大、更易用的虚拟形式
   2. 提供API：为了满足用户的需求并且利用虚拟机的特性(比如运行程序、分配内存或访问文件)，操作系统还提供了一些可供用户调用的接口(APIs)以及应用程序的标准库
   3. 资源管理器：虚拟化允许多程序运行(如共享CPU)，允许多程序并发地访问自己的指令和数据(如共享内存)，允许多程序访问设备(如共享磁盘等等)
2. 设计目标

抽象、性能、保护、可靠、安全、移动性

1. 特征
   1. 并发（concurrency）

处理多个同时进行的活动的能力

* 1. 共享（sharing）

操作系统与多个用户程序共享计算机系统中的资源，在共享的过程中操作系统需要尽量最优化资源的分配与对资源的保护

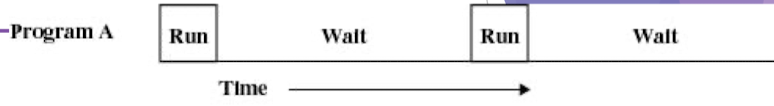
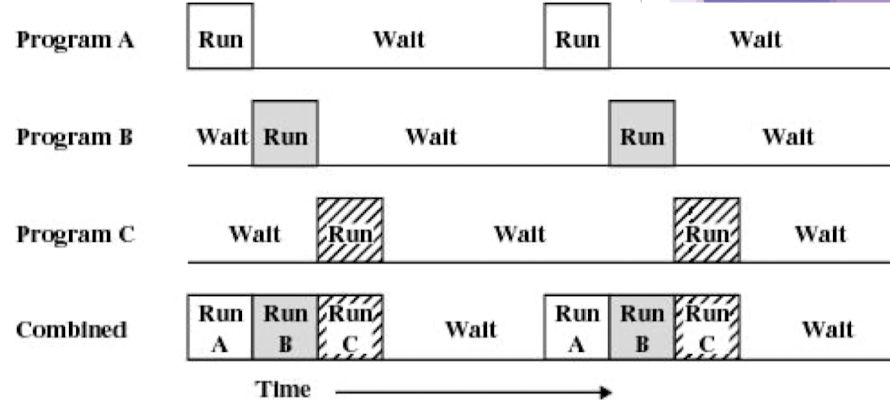
* 1. 虚拟（Virtual）

将一个物理实体映射为若干个逻辑实体，通过分时或分空间使用逻辑实体实现对物理实体的共享

* 1. 随机性（不确定性）

操作系统必须准备好随时对以不可预测的次序发生的事件进行响应

## 批处理操作系统的工作方式、分类

1. 工作方式
   1. 用户将作业交给系统操作员
   2. 系统操作员将许多用户的作业组成一批作业，输入到计算机系统中，在系统中形成一个自动转接的连续作业流
   3. 系统操作员启动操作系统
   4. 系统自动、依次执行每个作业
   5. 由系统操作员将作业结果交给用户
2. 分类
   1. 单道批处理系统
   2. 多道批处理系统

## 批处理操作系统中采用的SPOOLing技术

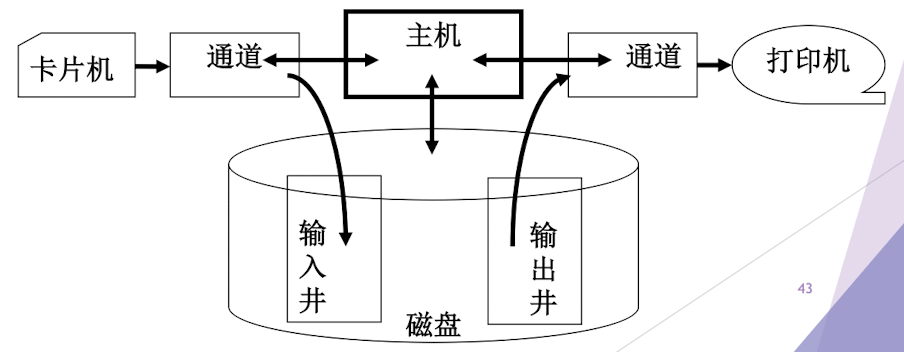
1. 定义

Simultaneous Peripheral Operation On-Line

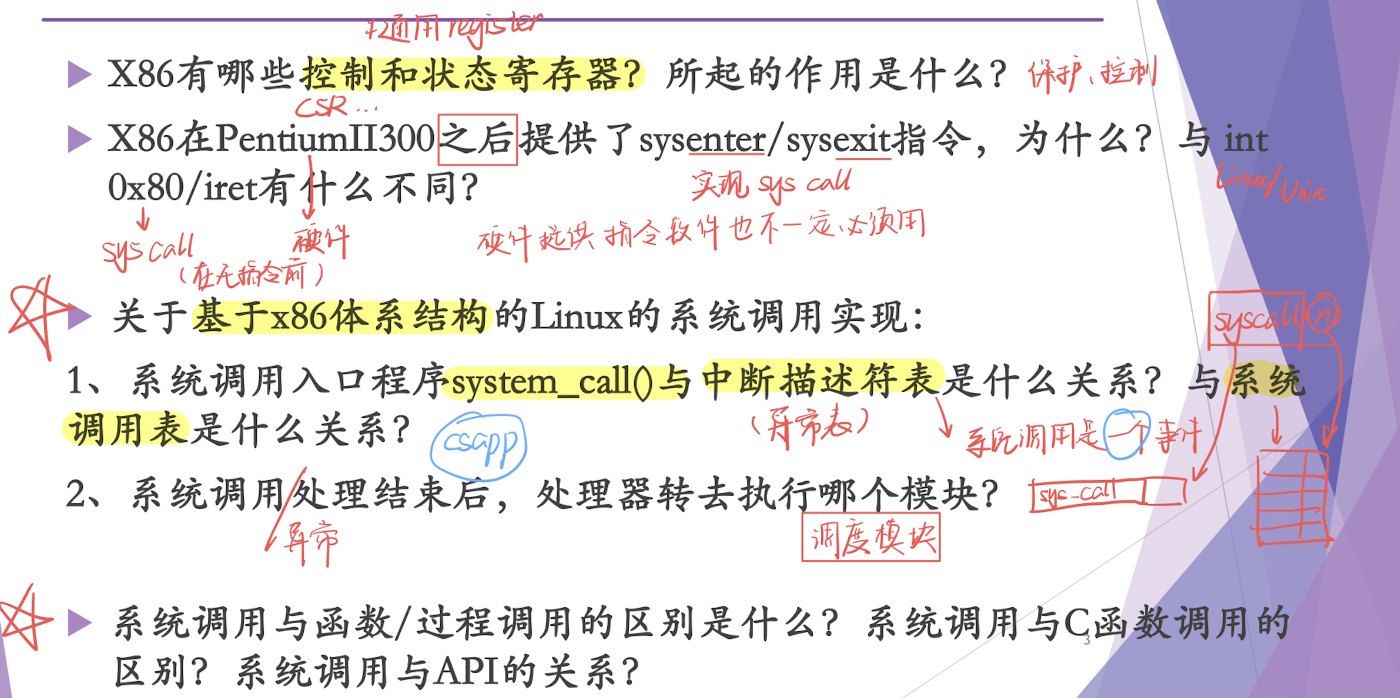
同时的外围设备联机操作——假脱机技术

1. 基本思想

利用磁盘/磁带作缓冲，将输入、计算、输出分别组织成独立的任务流（进程），分离I/O和计算，使二者真正并行

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

# Lecture2 中断异常机制（运行环境与机制）



## 寄存器

1. 分类
   1. 用户可见寄存器

数据寄存器、地址寄存器、条件码寄存器

机器语言在任何特权级别下均可以直接饮用

* 1. 控制和状态寄存器

程序计数器、指令寄存器、程序状态字（PSW）寄存器

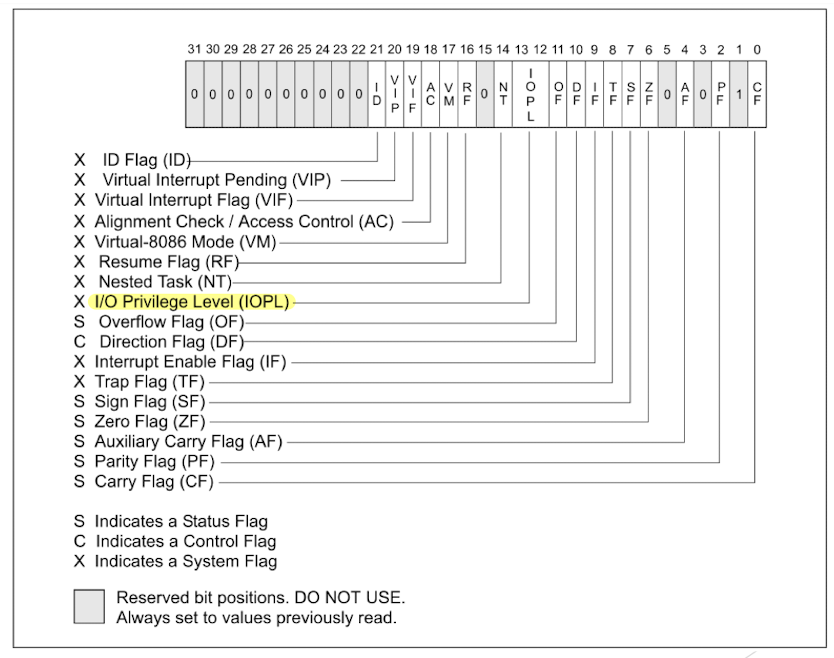
用于控制处理器，在某种特权级别下可以访问、修改

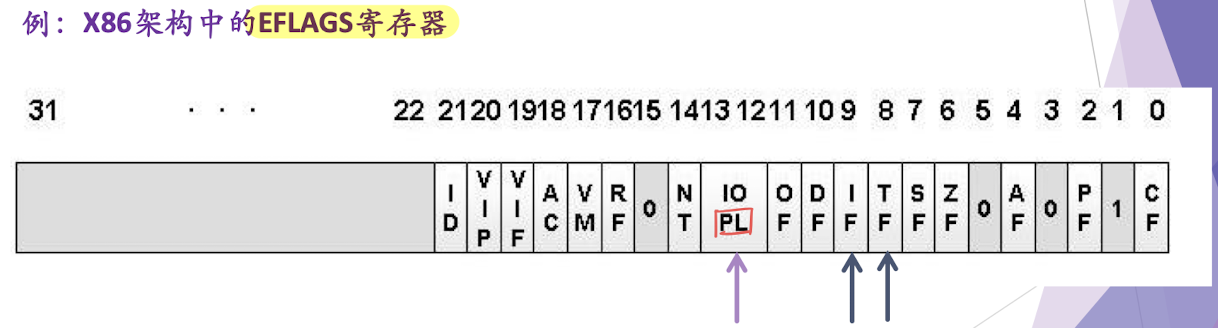
## CPU状态（特权级别）

1. 定义

处理器具有不同的运行模式，在不同的运行模式（特权级别）下运行的指令集合不同

1. 实现

在程序状态字寄存器PSW中专门设置一位，根据运行程序对资源和指令的使用权限而设置不同的CPU状态

1. 不同特权级下的指令
   1. 特权指令

只能由操作系统使用、用户程序不能使用的指令

* 1. 非特权指令

用户程序可以使用的指令

1. 状态的切换
   1. 用户态 → 内核态/监督模式/管态

只能通过中断/异常/陷入机制

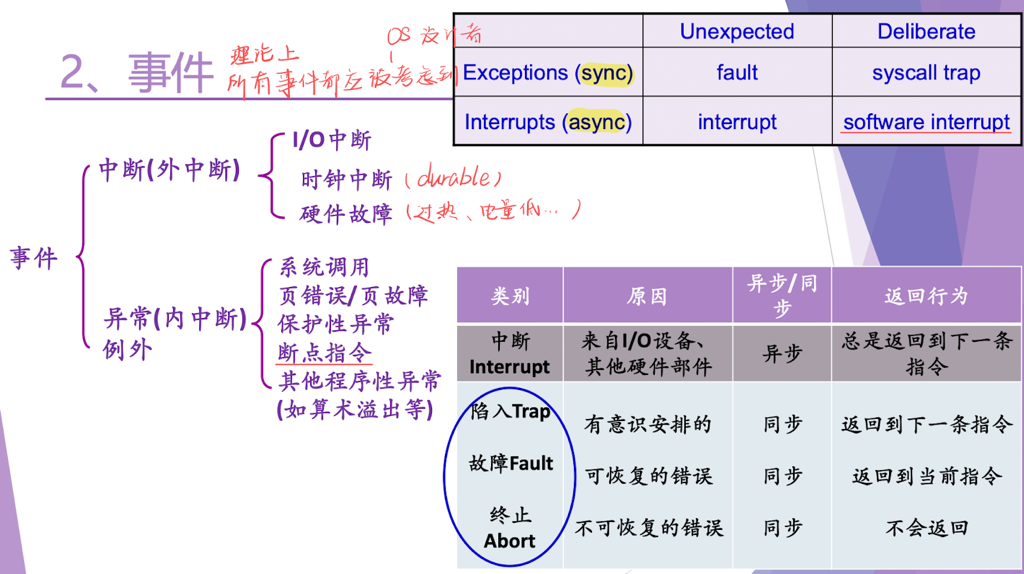
陷入指令（访管指令）：提供给用户程序的接口，用于调用操作系统的功能（服务）；例如int，trap，syscall，sysenter/sysexit，ecall

* 1. 内核态 → 用户态

设置程序状态字PSW

## 中断异常机制

操作系统是由中断/异常等事件驱动的

1. 事件的分类
2. 中断/异常的概念和区别
   1. CPU对系统发生的某个事件作出的反应，事件的发生改变了处理器的控制流
   2. 中断：来自CPU外，为了支持CPU和设备之间的并行操作而引入

例子：I/O与CPU计算的并行

* 1. 异常：表示CPU执行指令时本身出现的问题

例子：算术溢出、除零、取数时的奇偶错误，访存地址时越界或执行

了“陷入指令”等

1. 特点
   1. 随时发生
   2. 自动处理
   3. 被中断程序状态可恢复（在以后的某个时刻继续执行）
2. 作用
   1. 及时处理设备发来的中断请求
   2. 可使OS可以捕获用户程序提出的服务请求
   3. 防止用户程序执行过程中的破坏性活动

……

1. 工作原理

硬件和软件相互配合

* 1. 硬件：中断/异常响应

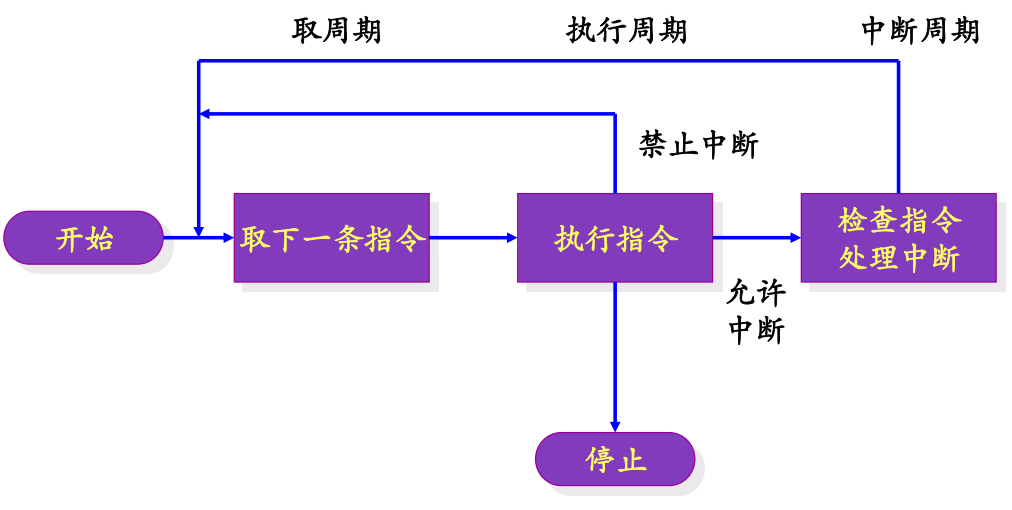
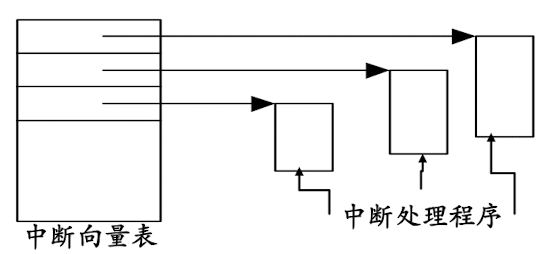
捕获中断源发出的中断/异常请求，以一定方式响应，将处理器控制权交给特定的处理程序

* 1. 软件：中断/异常处理

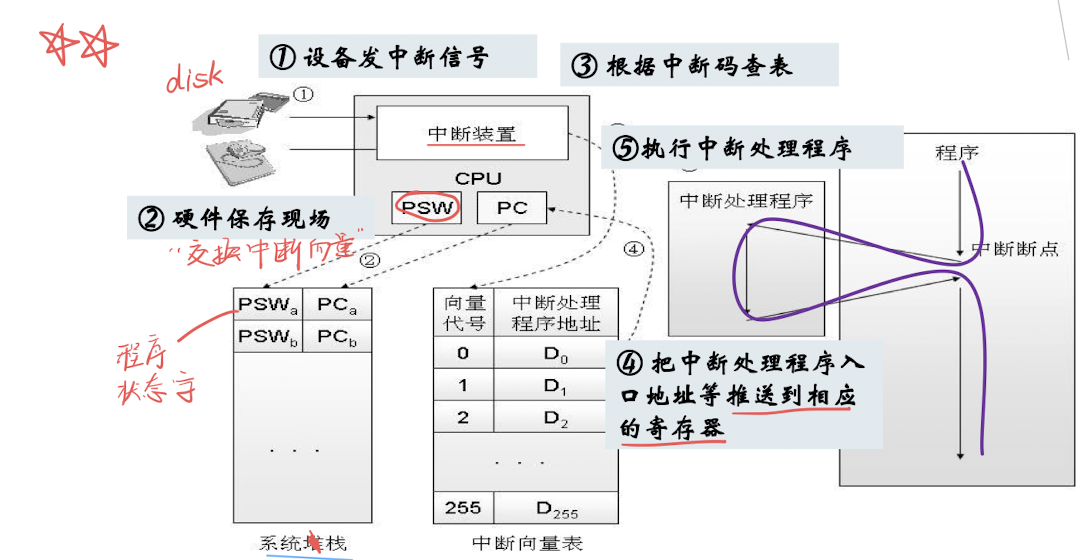
识别（在系统初始化时设置）中断/异常类型并完成相应的处理

1. 工作过程
   1. 框架

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（在此之前需要先调度该程序对应的进程），继续执行被打断的程序

* 1. 中断响应流程
     1. 硬件
* 在每一个指令执行周期中，CPU取出指令并执行
* 若中断被禁用，则在指令执行结束后直接取下一条指令，开始下一个指令执行周期；
* 若允许中断，则在指令执行周期的最后时刻扫描中断寄存器，查看是否有中断（当有多个中断时有中断处理逻辑来决定多个中断怎么处理）。
* 若有中断，则中断硬件将该中断寄存器的内容按规定编码后（称为中断码）送入PSW的相应位，通过“交换中断向量”引出中断处理程序（见4）
* 若无中断，则取下一条指令，开始下一个指令执行周期
* 中断向量表是一个内存单元，存放各中断处理程序入口地址和程序运行所需的处理机状态字（PSW）。计算机开机时操作系统按硬件要求初始化该表（Q：操作系统初始化与中断/异常有哪些关联？）

硬件执行流程按中断号/异常类型的不同，通过中断向量表转移控制权给中断处理程序

* + 1. 软件
* 保存相关寄存器（如通用寄存器，硬件只保存PSW和PC）
* 分析中断/异常的具体原因
* 执行对应的处理功能
* 恢复现场，返回被事件打断的程序

## 中断异常响应流程实例

以设备输入输出中断为例

1. 硬件
   1. 打印机给CPU发中断信号
   2. CPU处理完当前指令后检测到中断，判断出中断来源并向相关设备发确认信号
   3. CPU开始为软件处理中断做准备

* 处理器状态被切换到内核态
* 在系统栈中保存被中断程序的重要上下文环境，主要是程序计数器PC和程序状态字PSW
  1. CPU根据中断码查中断向量表，获得与该中断相关的处理程序的入口地址，并将PC设置成该地址，新的指令周期开始时，CPU控制转移到中断处理程序

1. 软件
   1. 中断处理程序开始工作，在系统栈中保存现场信息
   2. 检查I/O设备的状态信息，操纵I/O设备或者在设备和内存之间传送数据等
2. 硬件
   1. 中断处理结束时，CPU检测到中断返回指令，从系统栈中恢复被中断程序的上下文环境 ，CPU状态恢复成原来的状态，PSW和PC恢复成中断前的值，CPU开始一个新的指令周期（也可以调度其他进程，恢复的是被调度进程的上下文环境）

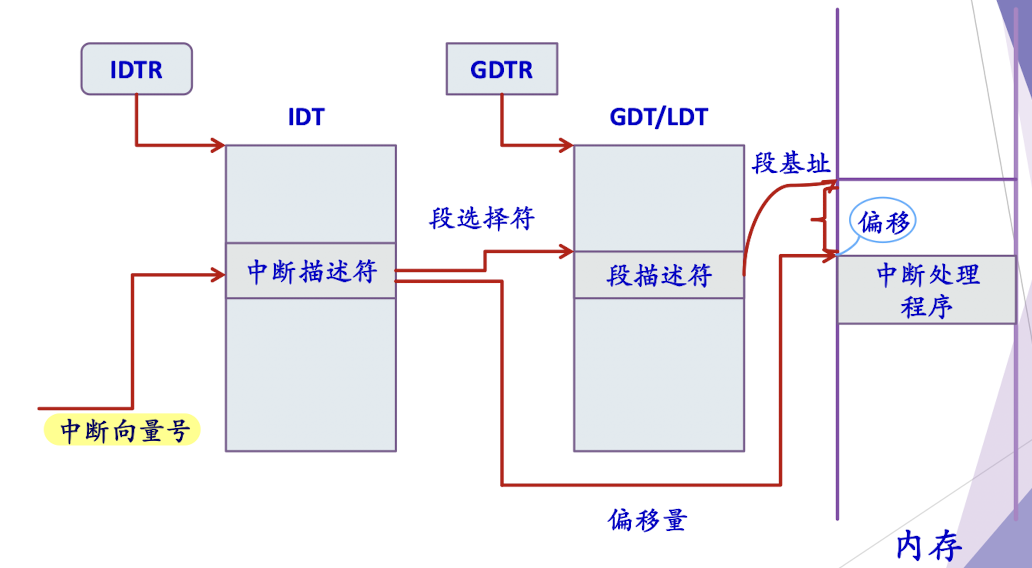
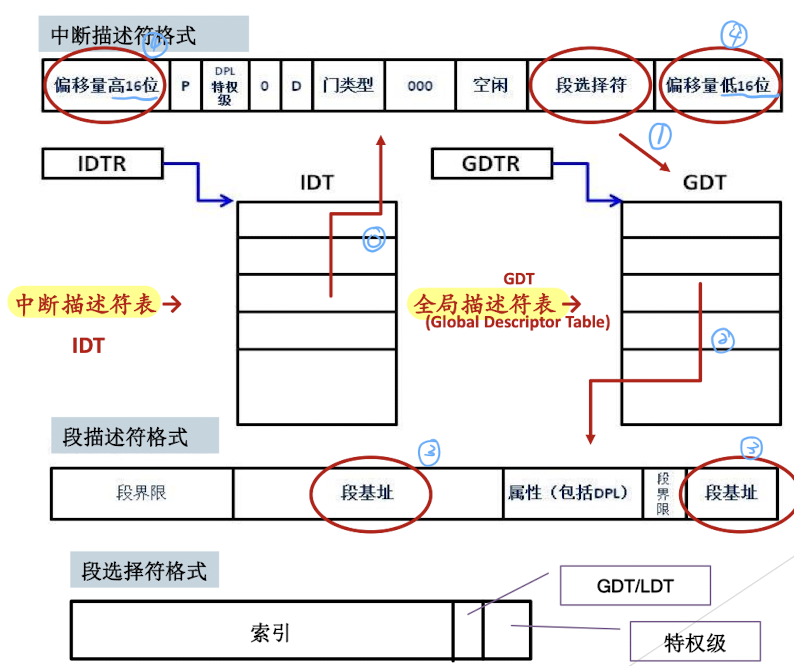
## IA32体系结构的基于x86处理器的中断异常机制

1. 有关硬件
   1. 中断控制器（PIC或APIC）
   2. 实模式：中断向量表（Interrupt Vector）

存放中断服务程序的入口地址

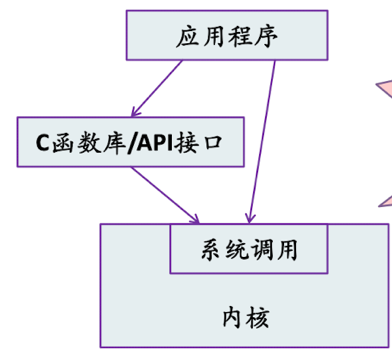
* 入口地址＝段地址左移4位＋偏移地址
* 不支持CPU运行状态切换
* 中断处理与一般的过程调用相似
  1. 保护模式：中断描述符表（Interrupt Descriptor Table）

采用门(gate)描述符数据结构描述中断向量

1. 中断/异常的硬件处理过程
   1. 确定与中断或异常关联的向量i
   2. 通过IDTR寄存器找到IDT表，获得中断描述符（表中的第i项）
   3. 从GDTR寄存器获得GDT的地址；结合中断描述符中的段选择符，在GDT表获取对应的段描述符；从该段描述符中得到中断或异常处理程序所在的段基址
   4. 特权级检查
   5. 检查是否发生了特权级的变化，如果是，则进行堆栈切换(必须使用与新的特权级相关的栈)
   6. 硬件压栈，保存上下文环境；如果异常产生了硬件出错码，也将它保存在栈中
   7. 如果是中断，清IF位
   8. 通过中断描述符中的段内偏移量和段描述符中的基地址，找到中断/异常处理程序的入口地址()入口地址 = 段基址＋偏移，执行其第一条指令

## 系统调用

1. 定义

用户在编程时可以调用的操作系统功能

1. 系统调用的作用（系统调用与C函数调用的区别）
   1. 系统调用是操作系统提供给用户的唯一接口
   2. 系统调用是操作系统提供给编程人员的唯一接口
2. 执行过程
   1. CPU执行特殊的陷入指令，引发异常，完成用户态到内核态的切换
   2. 中断/异常机制：硬件保护现场；通过查中断向量表把控制权转给系统调用总入口程序
   3. 系统调用总入口程序：保存现场；将参数保存在内核栈里；通过查系统调用表把控制权转给相应的系统调用处理例程或内核函数
   4. 执行对应的系统调用例程
   5. 恢复现场，返回用户程序
3. 基于x86处理器的Linux系统调用实现
   1. CPU执行系统调用指令int $0x80（%eax存放系统调用号，%ebx, %ecx, %edx, %esi, %edi…），产生异常

* 改变特权级，切换栈：用户栈→内核栈

从任务状态段TSS中装入新的栈指针（SS︰ESP），该指针指向内核栈

* 用户栈的信息（SS︰ESP）、EFLAGS(PSW)、代码段寄存器CS 、EIP 寄存器的内容压栈（返回用）
  + - 将EFLAGS压栈后，复位TF，IF位保持不变
    - 在IDT中找到该门描述符，从中找出段选择符装入代码段寄存器CS
    - 代码段描述符中的基地址 + 陷阱门描述符中的偏移量 → 定位system\_call()的入口地址
    - 执行system\_call中的指令
  1. system\_call()将参数保存在内核栈中，检查是否是合法的系统调用号，若是则根据系统调用号检索系统调用表，找到系统调用程序入口，比如sys\_func()
  2. sys\_func()执行完后返回system\_call()，将返回值从内核栈转移到%eax中，经过ret\_from\_sys\_call()例程（调度，处理信号）返回用户程序

# Lecture3 进程线程模型

## 进程模型

1. 基本信息
   1. 特点
      * 是正在运行程序的抽象，是程序的一次执行过程
      * 是对CPU的抽象，将一个CPU变换成多个虚拟的CPU，操作系统将CPU调度给进程
      * 每个进程具有独立的地址空间，系统资源以进程为单位分配
   2. 定义
      * 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
      * 如何查看当前系统中有多少个进程？Linux: ps -e
   3. 分类
      * 系统进程、用户进程
      * 前台进程、后台进程
      * CPU密集型进程、 I/O密集型进程
   4. 进程与程序的区别
      * 程序是静态的，进程是动态的：进程有生命周期的，有诞生有消亡，是短暂的；而程序是相对长久的
        + 一个程序可对应多个进程
        + 进程具有创建其他进程的功能
      * 进程更能准确刻画并发，而程序不能
2. 进程的状态
   1. 进程的三种基本状态
      * 运行态（Running）

占有CPU，并在CPU上运行

* + - 就绪态（Ready）

已经具备运行条件，但由于没有空闲CPU，而暂时不能运行

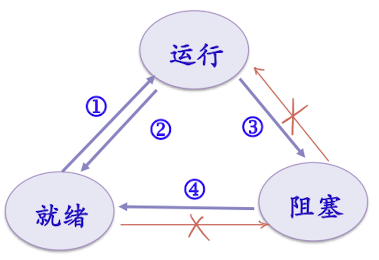
* + - 等待/阻塞态（Waiting/Blocked）

因等待某一事件（如完成I/O，等待读盘结果DMA）而暂时不能运行，可以细化为阻塞态、封锁态、睡眠态

* 1. 进程状态之间的转换

进程在消亡前处于且仅处于某一状态

不同系统设置的进程状态数目不同

* + - 三状态模型
    - 进程的其他状态
* 创建（new）

已完成创建一进程所必要的工作（如PID、PCB）

但尚未同意执行该进程（因为资源有限）

ps. 创建与就绪的关系

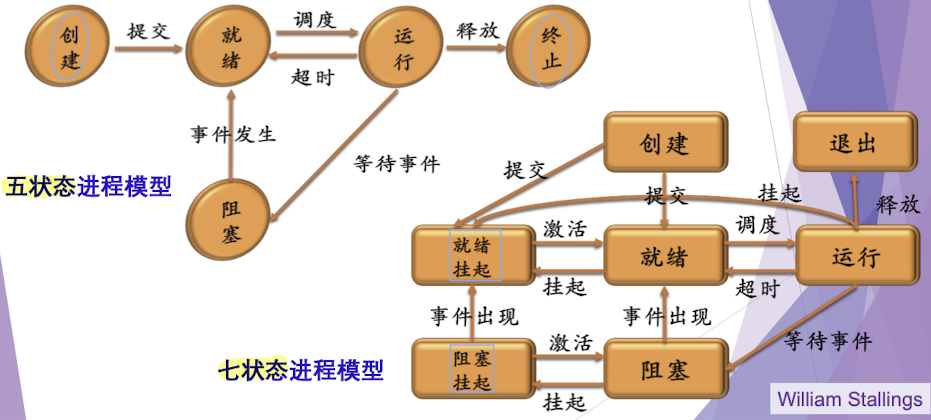
* + - * 终止（terminated）

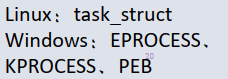
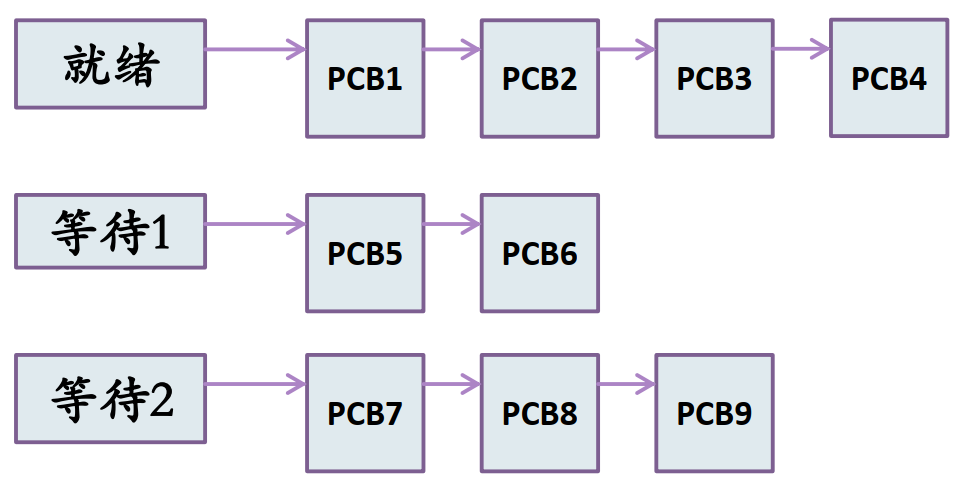
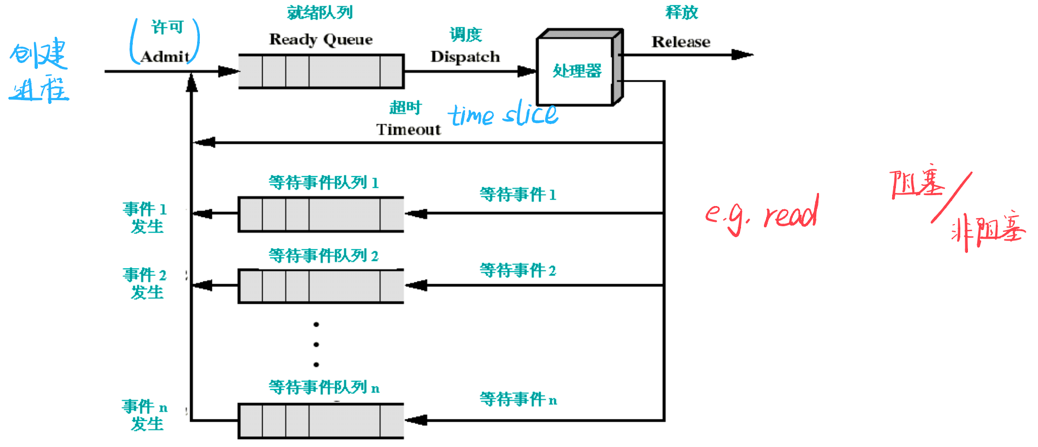
终止执行后，进程进入该状态

可完成一些数据统计工作与资源回收

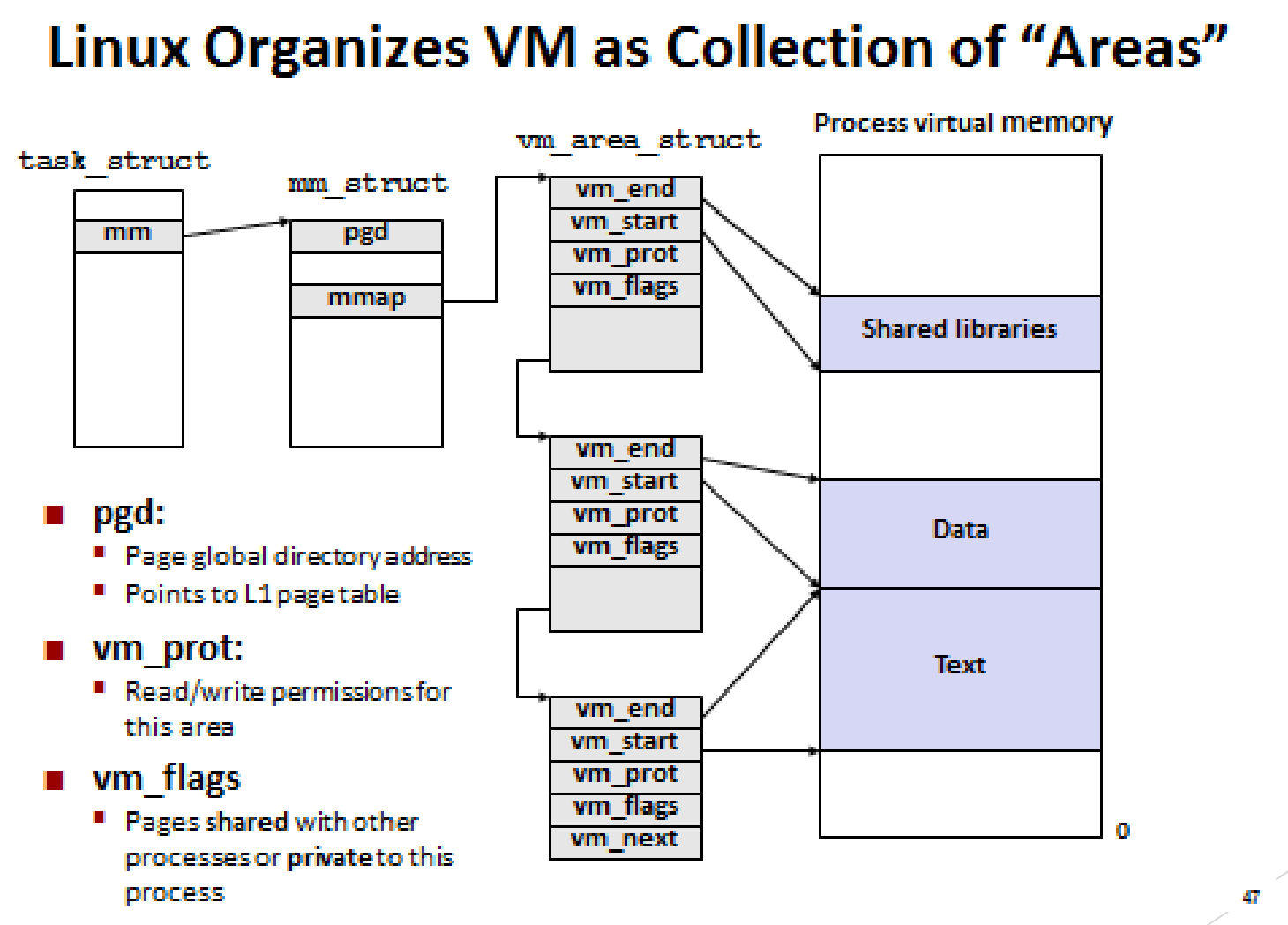
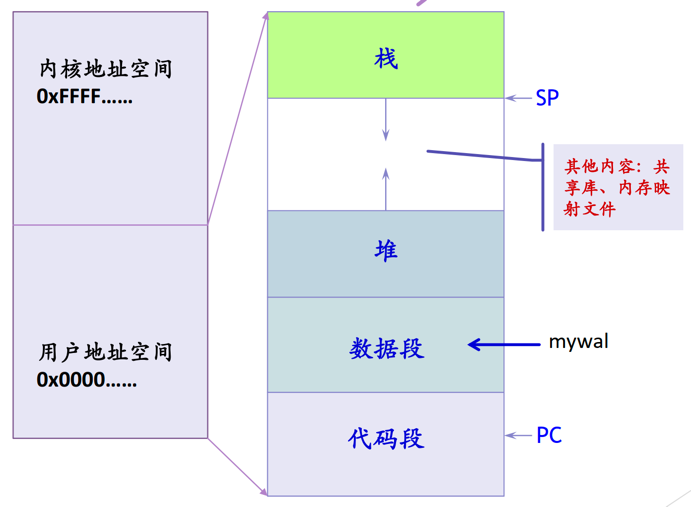
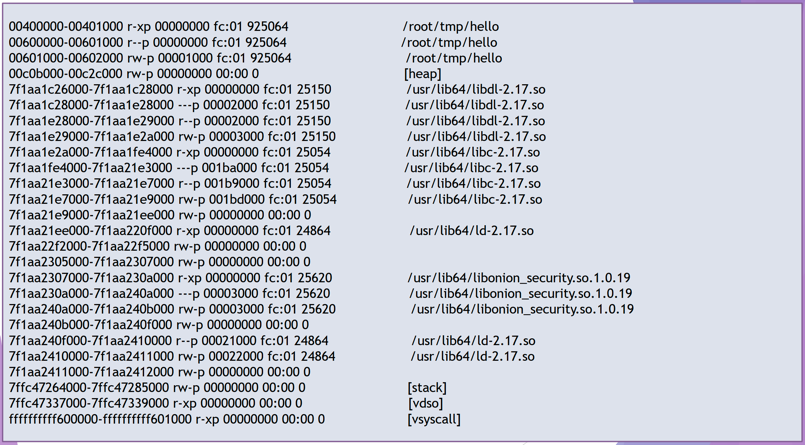
* + - * 挂起（suspend）

把一个进程从内存转到磁盘（进程不占用内存空间，其进程映像交换到磁盘上）——用于调节负载

* + - 五状态模型与七状态模型

1. 进程的数据结构
   1. 进程控制块PCB
      * PCB: Process Control Block
        + 进程描述符、进程属性
        + 操统系统中表示进程的数据结构
        + 记录进程的各种属性，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
        + 进程与PCB一一对应
        + PCB是系统感知进程存在的唯一标志
        + 进程表：所有进程的PCB的集合
   2. PCB主要内容
      * 进程描述信息
        + 进程标识符（process ID），唯一，通常是整数
        + 进程名，通常基于可执行文件名（不唯一）
        + 用户标识符（user ID）
        + 进程组关系
      * 进程控制信息
        + 进程当前状态
        + 优先级（priority）
        + 代码执行的入口地址
        + 对应的可执行文件名（磁盘地址）
        + 运行统计信息（e.g.执行时间片页面调度）
        + 进程间同步和通信；进程阻塞的原因
        + 进程的队列指针
        + 进程的消息队列指针
      * 资源使用情况
        + 虚拟地址空间的现状
        + 打开文件列表（文件描述符…）
      * CPU现场信息（上下文）
        + 寄存器值（通用寄存器，程序计数器PC，进程状态字PSW…）
        + 指向该进程段/页表的指针
   3. 内核对进程的组织与管理
      * 进程表（队列）
        + 操作系统为每一类进程建立一个或多个队列，队列元素为PCB
        + 伴随进程状态的改变，其PCB从一个队列进入另一个队列
        + 注： ﻿多个等待队列等待的事件不同；﻿就绪队列也可以多个；﻿单CPU情况下，运行队列中只有一个进程
      * 五状态进程模型的进程队列模型
        + 就绪队列无优先级（e.g. FIFO）
        + 事件发生时，进程从对应队列移动到就绪队列中
2. 进程的地址空间

**一致、私有、巨大、稀疏**

* + - * 1. 操作系统给每个进程都分配了一个地址空间，地址空间是对物理内存的抽象
        2. 进程地址空间的布局
* Linux：cat /proc/PID/maps
  + 显示进程映射的内存区域和访问权限
  + 在内核中，进程的段用vm\_area\_struct表示，所有地址空间存储在task(PCB)->mm->mmap链表中
  + 一个文件可以**映射到进程的一段内存区域中**，映射的文件描述符保存在vm\_area\_struct->vm\_file域中，这种内存区域叫做有名内存区域，另一个是匿名映射内存区域

1. 进程的控制
   1. 简介
      * 进程控制操作完成进程各状态之间的转换，由具有特定功能的原语完成
      * 原语（primitive）：﻿完成某种特定功能的一段程序，具有不可分割性或不可中断性，﻿即原语的执行必须是连续的，在执行过程中**不允许被中断**（atomic）
      * 进程创建原语；进程撤消原语；阻塞原语；唤醒原语；挂起原语；激活（解挂）原语；改变进程优先级
   2. 进程的创建
      * 创建时机
        + 系统初始化
        + 操作系统提供的服务
        + 交互用户登录系统（？）
        + 现有进程派生出一个新进程
        + 提交一个程序执行（e.g. 命令行）
      * **创建过程**

UNIX： fork/exec; WINDOWS： CreateProcess

* + - * 给新进程分配一个唯一标识(pid)以及进程控制块(PCB)，并为进程分配地址空间
      * 初始化进程控制块，设置默认值 (如：状态为New，......)
      * 设置相应的队列指针，如把新进程加到就绪队列的链表中
      * 创建或扩充其他数据结构
  1. 进程的终止

UNIX: exit; WINDOWS：ExitProcess

* 终止时机
  + 正常退出（自愿的）
  + 出错退出（自愿的）
  + 严重错误（非自愿）
  + 被其他进程杀死（非自愿）
* 终止/撤销过程
  + 结束子进程或线程（zombie in Linux）
  + 收回进程所占有的资源（关闭打开的文件，断开网络连接，回收分配的内存…）
  + 撤消该进程的PCB
  1. 进程阻塞和进程唤醒

UNIX: wait; WINDOWS: WaitForSingleObject

* 时机/过程

处于运行状态的进程，在其运行过程中期待某一事件发生，如等待键盘输入、等待磁盘数据传输完成、等待其它进程发送消息，当被等待的事件未发生时，由进程自己执行阻塞原语，使自己由运行态变为阻塞态

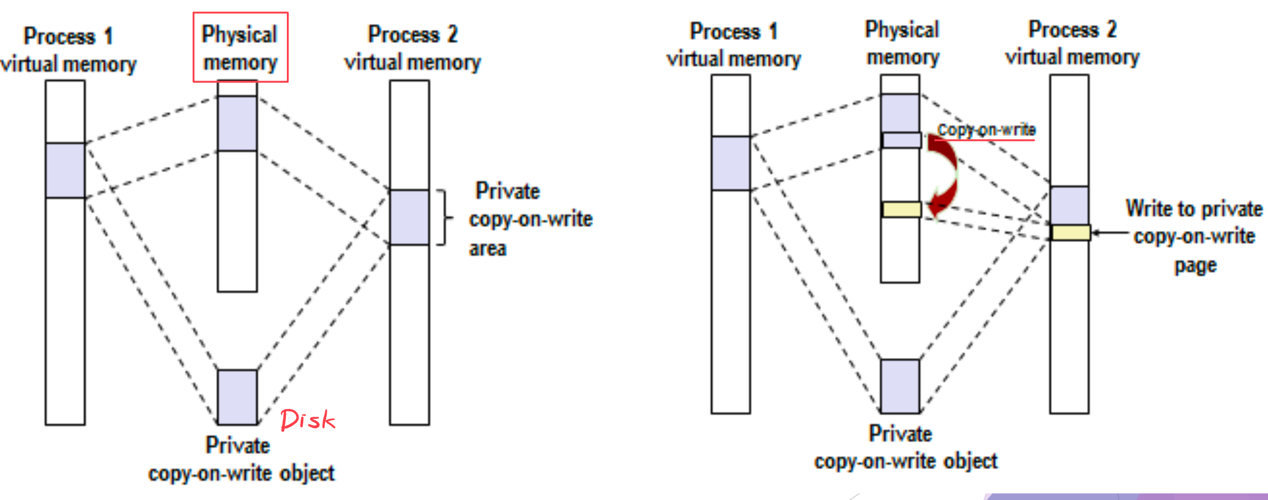
* 1. 其他事件

正常结束； 给定时限到；缺少内存；存储器出界；保护性出错（写只读文件）；算术错；超出时间（进程等待超过对某事件的最大值）； I/O 失败；无效指令（如试图执行数据）；特权指令；操作系统干预（如当死锁发生时）；父进程请求中止某一子进程；父进程中止（子进程也中止）

## Unix系统的进程控制操作

1. 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过程

* 1. **fork的实现**
* 为子进程分配一个空闲的进程描述符（proc结构）,并分配给子进程唯一标识pid
* 以一次一页的方式复制父进程地址空间（优化：﻿Linux的解决方案是利用存储管理模块中的“写时复制技术”COW（Copy-On-Write）对fork()进行了优化<虚拟内存机制>）
* 从父进程处继承共享资源，如打开的文件和当前工作目录等
* 将子进程的状态设为就绪，插入到就绪队列
* ﻿对子进程返回标识符 0，对父进程返回子进程的 pid
  1. **fork与COW**
     + 创建新的进程mm\_struct、vm\_area\_struct、页表的精确副本
     + ﻿将两个进程中的每个页面均标记为只读，﻿两个进程的每个vm\_area\_struct都标记为“私有的COW”
     + 于是返回时每个进程都有虚拟内存的精确副本，后续对内存的写使用COW机制创建新页面

1. exec()

包括一系列系统调用，它们都是通过用一段新的代码覆盖原来的内存空间，实现进程执行代码的转换

1. wait()

提供初级的进程同步措施，能使一个进程等待，直到另外一个进程结束为止

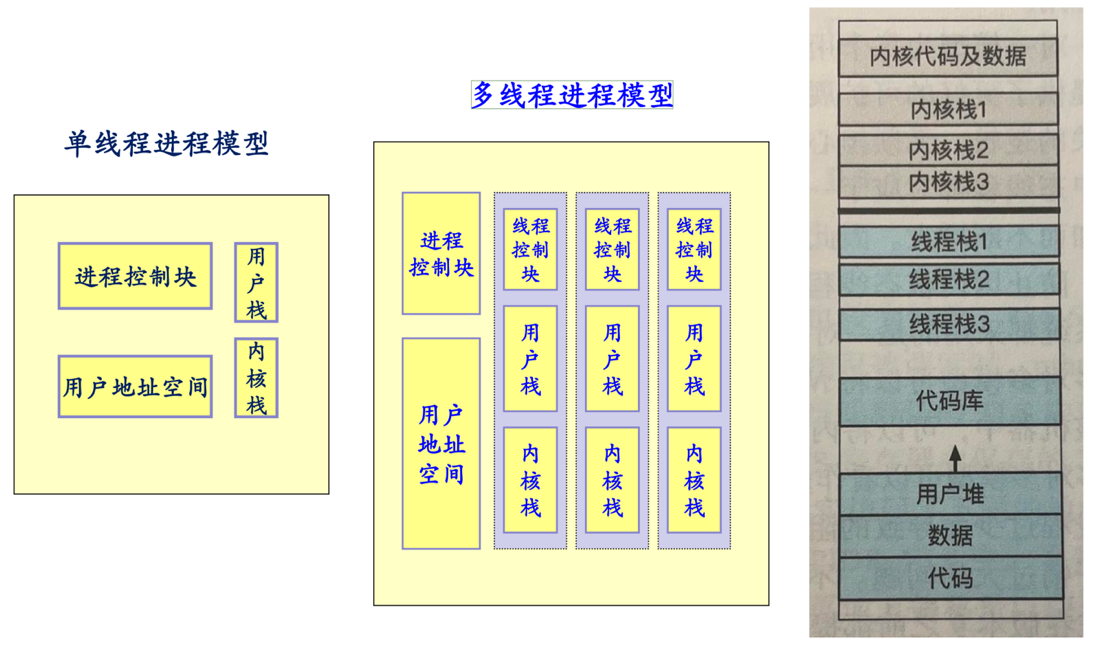
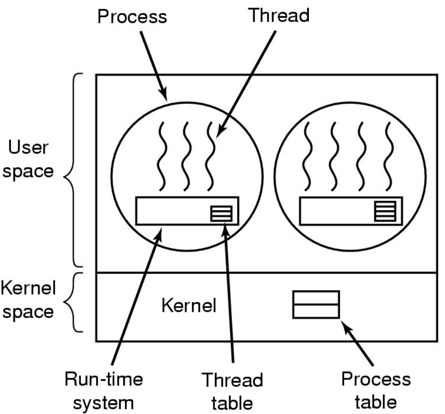
1. exit()

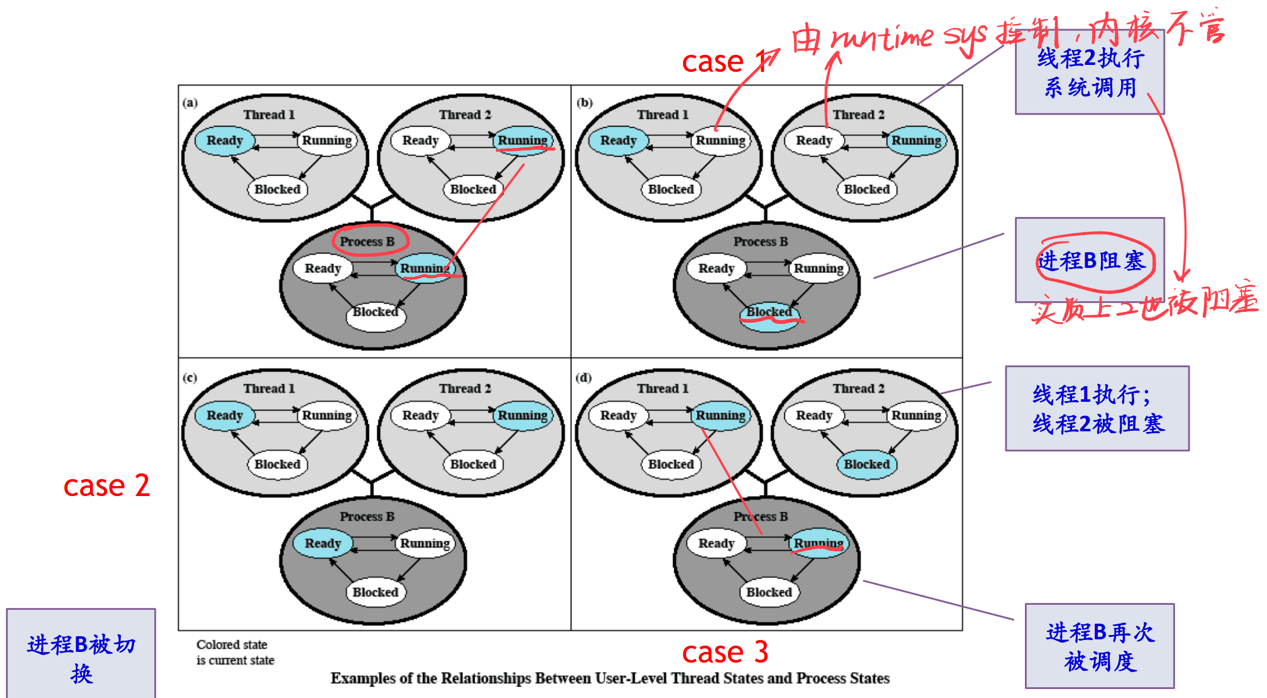
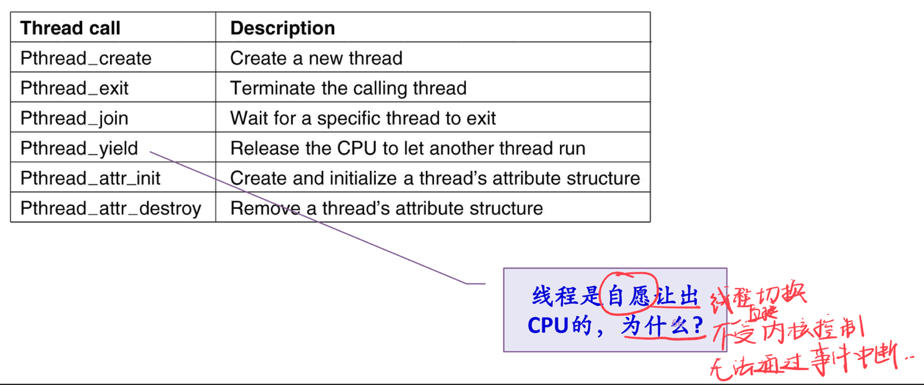
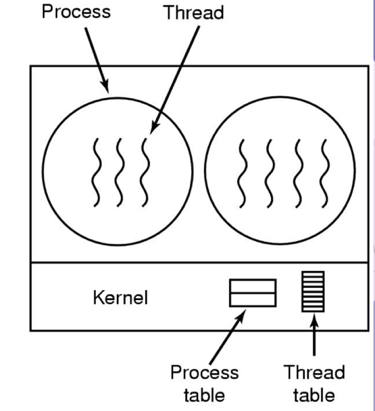
用来终止一个进程的运行

## 线程模型

1. 定义

进程中的一个运行实体，是CPU的调度单位，也叫﻿轻量级进程

1. 特点
   1. 有状态及状态转换 → 需要提供一些操作
   2. 不运行时需要保存上下文（相比进程而言更少﻿e.g.程序计数器等寄存器）
   3. 有自己的栈和栈指针（与进程栈“不交”）
   4. 线程间共享所在进程的地址空间和其他资源
   5. 可以创建、撤消另一个线程（程序开始时是以一个单线程进程方式运行的）
2. 用户级线程﻿（User Level Thread）
   1. 基本信息

* 在用户空间实现：在用户空间建立线程库，提供一组管理线程的函数
* 运行时系统完成线程的管理工作（操作、线程表）
* **内核**管理进程，**不知道（用户）线程的存在，**线程切换不需要内核态特权
  1. 实例：POSIX Pthreads
     + POSIX(Portable Operating System Interface)1003.4a
     + 多线程编程接口，以线程库方式提供给用户
  2. 优点
     + 线程切换快
     + 应用程序可以指定调度算法
     + 可运行在任何操作系统上（只需要实现线程库）
  3. 缺点
     + 内核阻塞进程（如进程进行系统调用）会导致进程中所有线程也被阻塞
     + 内核只将处理器分配给进程，因此同一进程中的两个线程不能同时运行于两个处理器上
     + 解决方案
       - 修改系统调用为非阻塞（﻿Jacketing/wrapper）
       - 重新实现对应系统调用的I/O库函数（syscall -> 切换线程而非进程）

1. 核心级线程
   1. 在内核中实现，内核管理所有线程，维护进程和线程的上下文，并向应用程序提供API接口
   2. 线程的切换需要内核支持，线程是调度的基本单元
   3. 例子： Windows
2. 混合式线程
   1. 在内核中实现，支持用户线程。线程创建在用户空间完成，而线程调度等在核心态（Kernel thread）完成
   2. 例子：Solaris

## 协程（coroutine）

1. 协程的定义

允许在一个进程/线程中实现多个执行流，并在这些执行流之间进行协作

1. 引入协程的原因

改善并发编程的性能，用于处理异步任务和大量并发连接，支持高并发

## 重点小结