# Lecture3 进程线程模型

## 基本问题

#### 问题1

##### 怎样理解“进程是对CPU的抽象”这句话？（VM，文件系统）

##### 何谓进程映像？进程有实体吗？在哪里？

##### ﻿怎样描述进程？一个进程都有什么（组成要素）？

##### ﻿创建进程主要完成哪些工作？

##### 进程的生命周期内都会经历哪些变化？怎样表示这些变化？（进程状态）

##### 进程有哪些状态？进程状态之间的转换（条件？操作？）

##### 进程状态转换的发生，是否一定导致另一个转换发生？

1. ﻿问题2

##### ﻿操作系统给进程提供内存空间，该空间的地址是虚拟地址还是物理地址？为什么？

##### ﻿操作系统如何描述进程的地址空间？

##### ﻿为什么有了进程后又引入线程？

##### ﻿怎样实现线程机制？为什么有各种支持线程的方式？

##### ﻿线程包Pthreads中相关的函数的功能？

1. ﻿问题3

##### ﻿中断/异常机制与进程线程模型的关联？

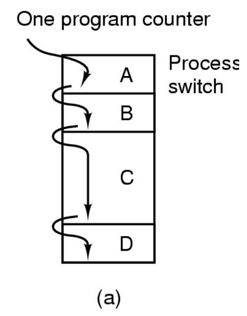
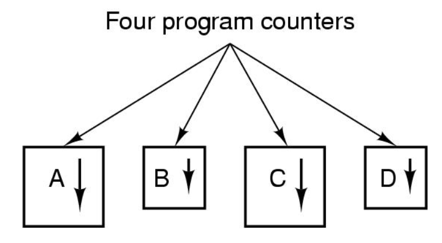
##### 机制和策略分离的原则在进程线程模型中的体现？

##### 协程是什么？为什么引入协程？协程怎么用？

## 进程模型

### 进程的概念

1. 一些基本概念
   1. 顺序程序与顺序环境

* 程序：指令或语句序列，体现某种算法
* 顺序环境的定义
  + 在﻿系统中只有一个程序在运行
  + ﻿该程序独占系统中的所有资源
  + 该程序的执行不受影响
* 顺序环境的特征
  + ﻿程序执行的顺序性：
  + ﻿程序执行的封闭性：﻿独占资源，执行过程不受外界影响
  + ﻿程序执行结果的可再现性：﻿程序运行结果与程序执行速度无关
  1. ﻿多道程序设计（Multiprogramming）
     + ﻿允许多个程序同时进入内存并运行以提高系统效率
  2. ﻿并发环境与并发程序
     + ﻿并发程序：在并发环境中执行的程序
     + ﻿并发环境的定义：一段时间间隔内，单处理器上有两个或两个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
     + ﻿并发环境的特征
       - 程序执行结果的不可再现性
       - 程序执行的间断性
       - 执行——停——执行
       - 资源共享
       - 独立性和制约性
       - 程序和计算不再一一对应

1. 进程
   1. ﻿定义： Process（Task/Job）
      * ﻿程序的一次执行过程
      * ﻿是正在运行程序的抽象
      * ﻿﻿将一个CPU变换成多个虚拟的CPU，是对CPU的抽象
      * ﻿系统资源以进程为单位分配，如内存、文件……
      * ﻿每个进程具有独立的地址空间
      * ﻿操作系统将CPU调度给进程
   2. 具体定义：﻿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

* 1. ﻿进程的分类
     + 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
  2. ﻿进程之间的关系
     + UNIX进程家族树：init进程为根
     + ﻿Windows：地位相同
  3. ﻿进程与程序的区别
     + ﻿进程更能准确刻画并发，而程序不能
     + 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

### 进程模型

1. 进程的三种基本状态

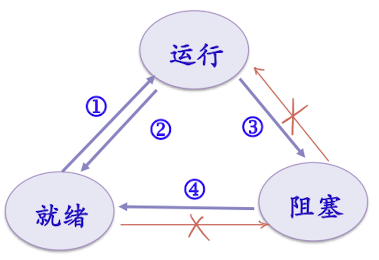
运行态、就绪态、等待态

* 1. 运行态（Running）
     + 占有CPU，并在CPU上运行
  2. 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
  3. 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

* 1. 三状态模型
  2. 进程的其他状态
     1. 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

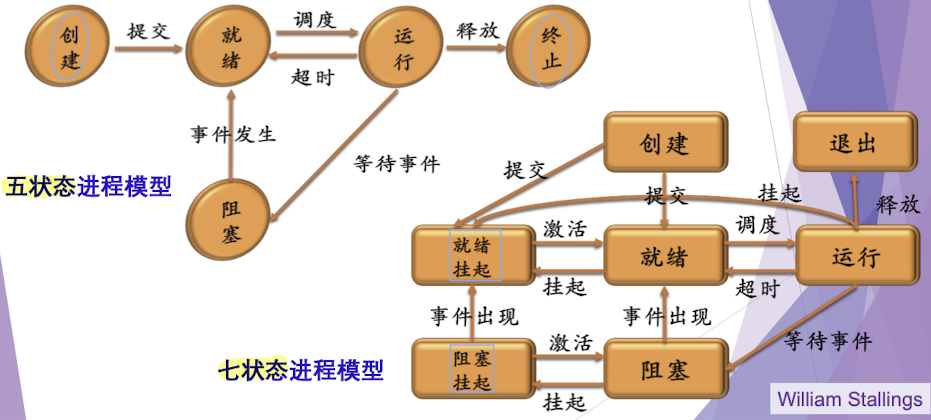
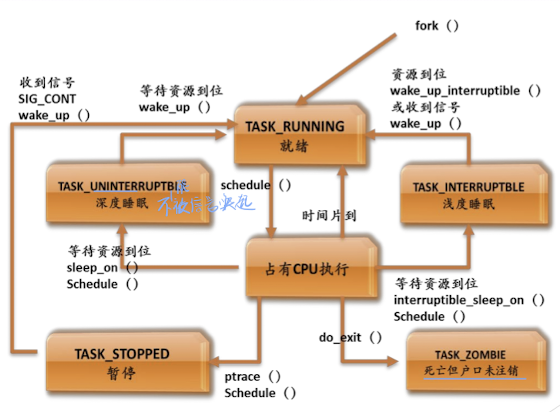
* + 1. 终止（terminated）

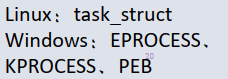
终止执行后，进程进入该状态

可完成一些数据统计工作与资源回收

* + 1. 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

* 1. 其他进程状态模型
  2. Linux进程状态

1. 进程的数据结构
   * + 1. ﻿进程控制块PCB
          1. ﻿PCB: Process Control Block

又称进程描述符、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

* + - * 1. ﻿操作系统通过 PCB 来控制和管理进程

﻿进程与PCB是一一对应的

﻿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

* + - 1. ﻿PCB主要内容
         1. ﻿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

* + - * 1. 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

* + - * 1. 所拥有的资源和其使用情况

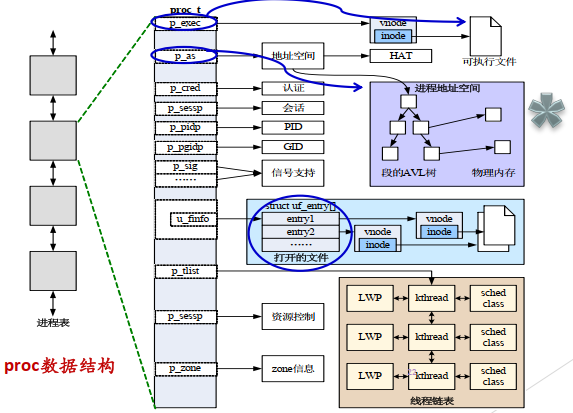
虚拟地址空间的现状

打开文件列表（文件描述符…）

* + - * 1. 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…）

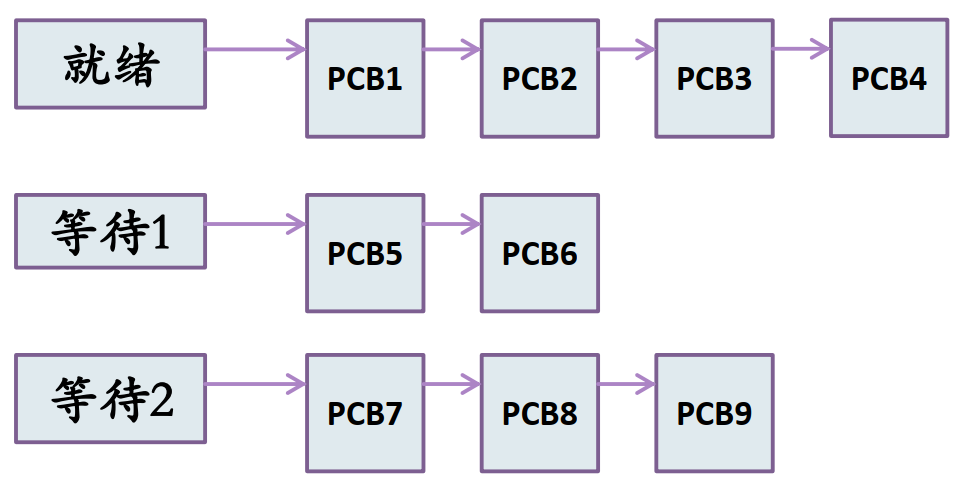
指向该进程的段/页表的指针

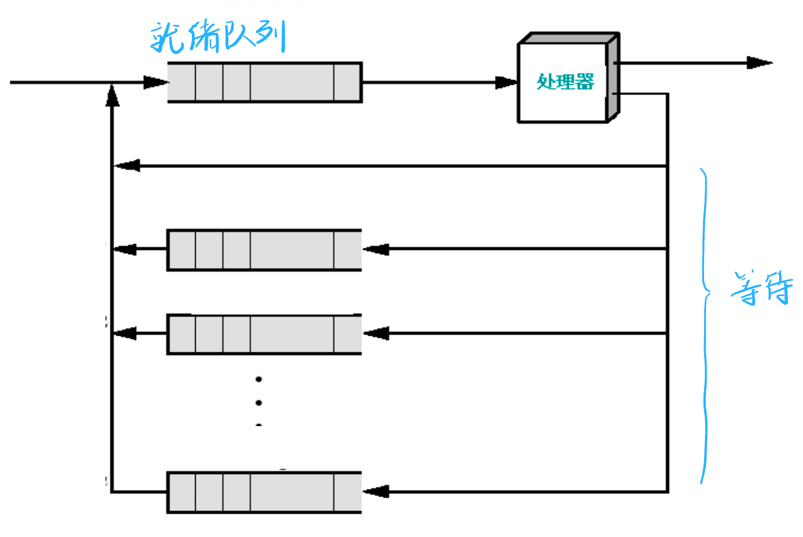
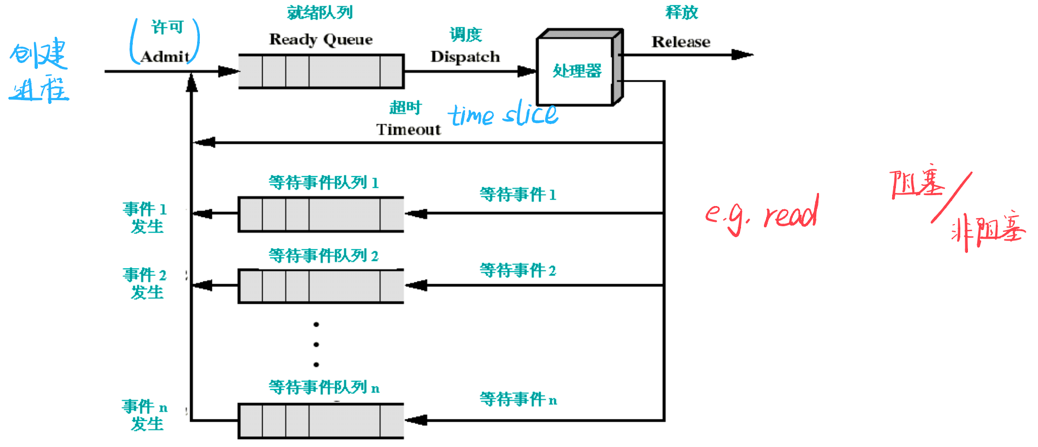
* + - 1. ﻿Solaris的进程控制块与进程表
         1. 所有资源均以内存为单位进行管理
      2. Linux task\_struct(PCB)
      3. ﻿内核对进程的组织与管理
         1. 进程表（队列）

﻿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

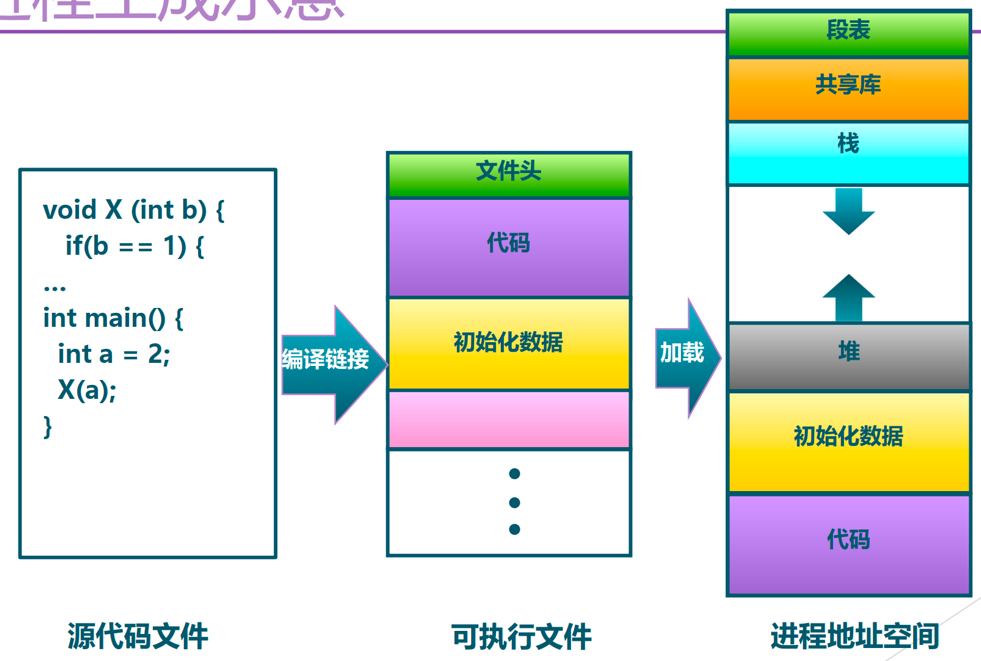
﻿伴随进程状态的改变，其PCB从一个队列进入另一个队列

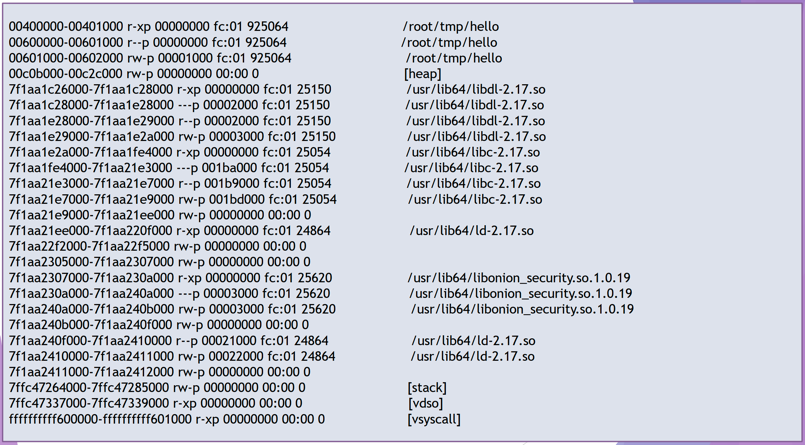
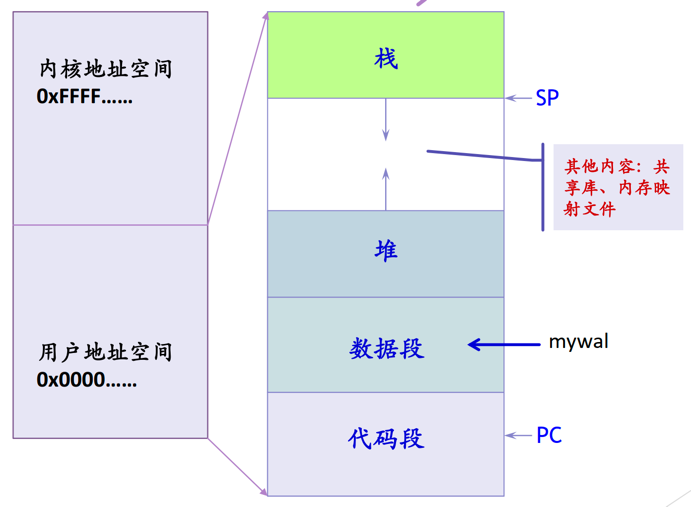
注： ﻿多个等待队列等待的事件不同；﻿就绪队列也可以多个；﻿单CPU情况下，运行队列中只有一个进程

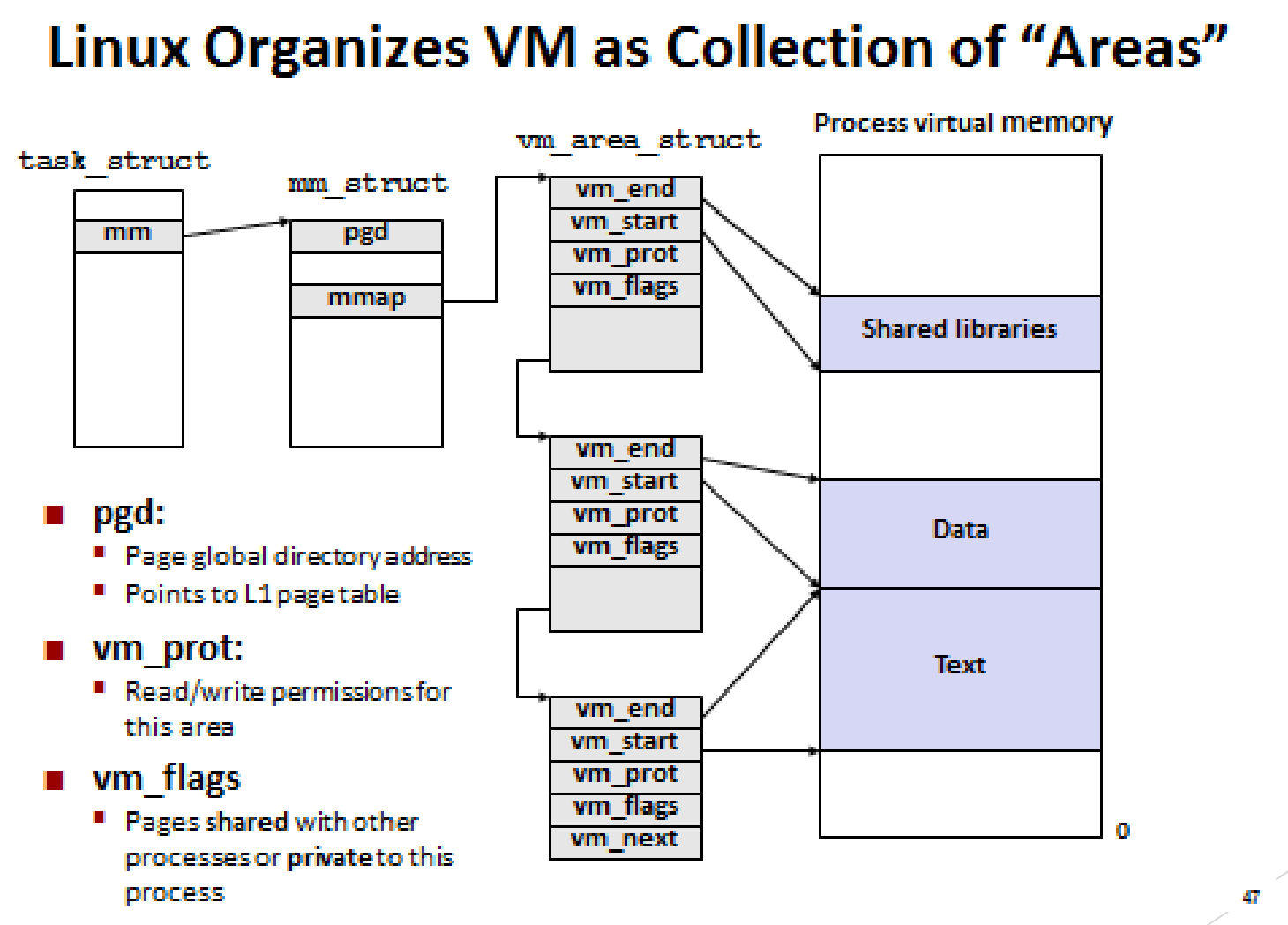
* 1. ﻿五状态进程模型的进程队列模型
     1. 就绪队列无优先级（e.g. FIFO）
     2. 当事件n发生时，对应队列移动到就绪队列中
  2. ﻿状态转换条件和转换函数

1. 进程的地址空间

**一致、私有、巨大、稀疏**

* + - 1. 简介
         1. 操作系统给每个进程都分配了一个地址空间
         2. 地址空间是对物理内存的抽象
      2. 进程的生成过程
      3. ﻿进程地址空间

1. Linux: cat /proc/PID/maps
   * 1. 显示进程映射的内存区域和访问权限
     2. 在内核中，进程的段区域用vm\_area\_struct表示，所有地址空间存储在task(PCB)->mm->mmap链表中
     3. 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. ﻿Linux虚拟内存组织

不一定以链表形式组织（如AVL树）

vm\_flags：用于实现COW等功能

1. 进程的控制
2. 简介
   1. ﻿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
      1. ﻿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）
      2. ﻿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
3. 进程的创建
   1. 创建时机

系统初始化时

操作系统提供的服务

交互用户登录系统

由现有的进程派生出一个新进程

提交一个程序执行（e.g. 命令行）

* + - * 1. **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)

﻿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

* ﻿e.g.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

* 1. 终止时机

1. 正常退出（自愿的）
2. 出错退出（自愿的）
3. 严重错误（非自愿）
4. 被其他进程杀死（非自愿）
   1. 终止/撤销过程
5. 结束子进程或线程（zombie in Linux）
6. ﻿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
* ……

1. ﻿撤消该进程的PCB
2. ﻿进程阻塞和进程唤醒

﻿UNIX: wait; WINDOWS: WaitForSingleObject

* 1. 时机

﻿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

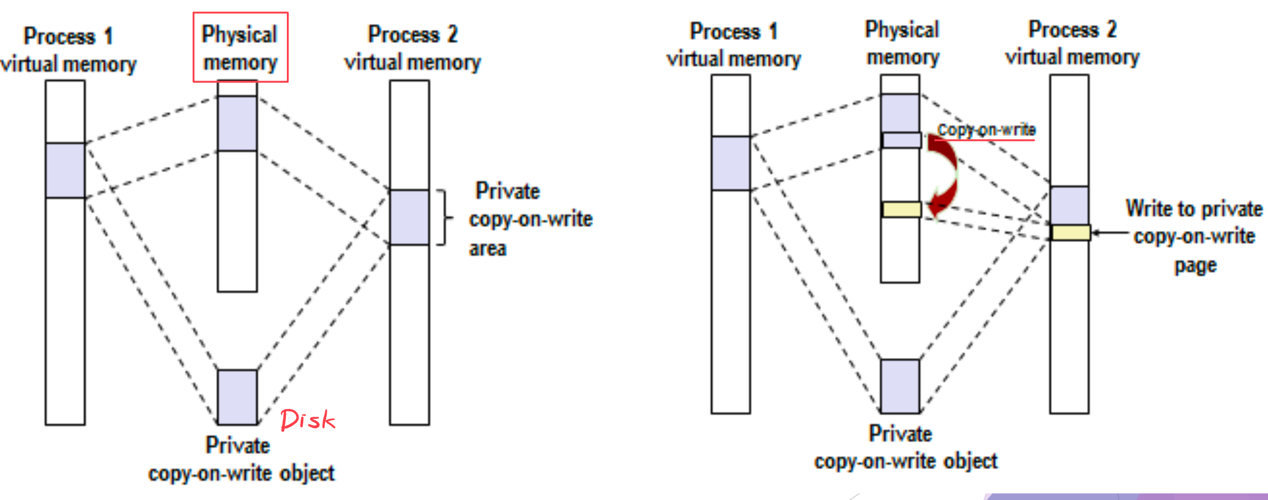
件未发生时，由进程自己执行阻塞原语，使自己由运行态变为阻塞态

1. 进程执行过程中的各种事件

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）

1. ﻿UNIX系统设计的进程控制操作
   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
* ﻿分配给子进程唯一标识 pid
* ﻿以一次一页的方式（完全）复制（事实上在物理内存中复制）父进程地址空间（如何优化？﻿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
* ﻿向父进程返回子进程的 pid
  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

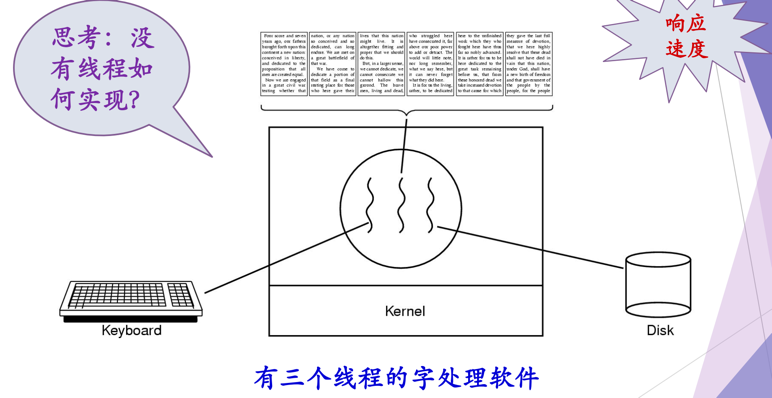
1. Unix fork()与虚拟内存机制
   1. ﻿VM和内存映射解释了fork如何为每个进程提供私有地址空间

﻿—— fork()后跟exec()的常见情况的完美方法

* 1. ﻿**为新进程创建虚拟地址空间的步骤**
     1. **﻿**创建新的进程mm\_struct、vm\_area\_struct、页表的精确副本
     2. ﻿将两个进程中的每个页面标记为只读
     3. ﻿将两个进程中的每个vm\_area\_struct标记为“私有COW”
  2. ﻿返回时，每个进程都有虚拟内存的精确副本
  3. ﻿后续对内存的写使用COW机制创建新页面

## 线程模型

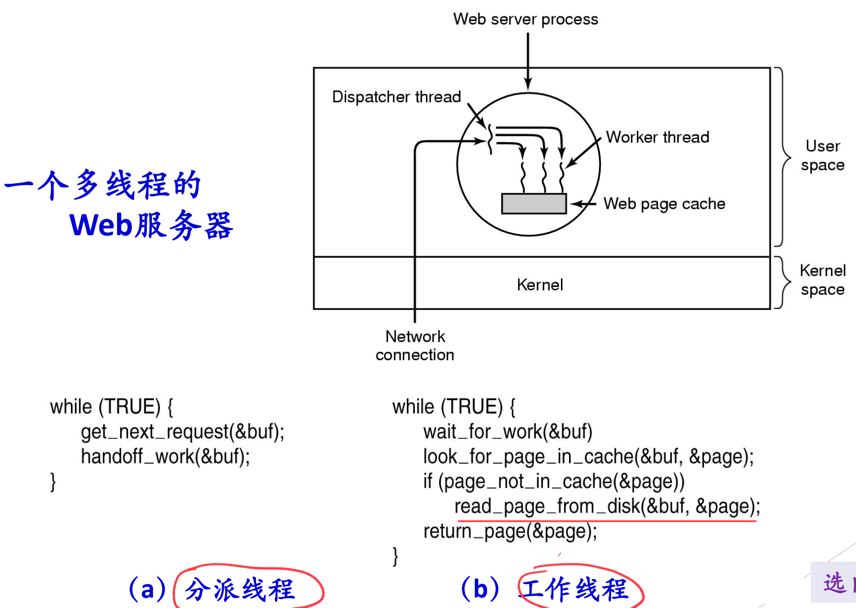
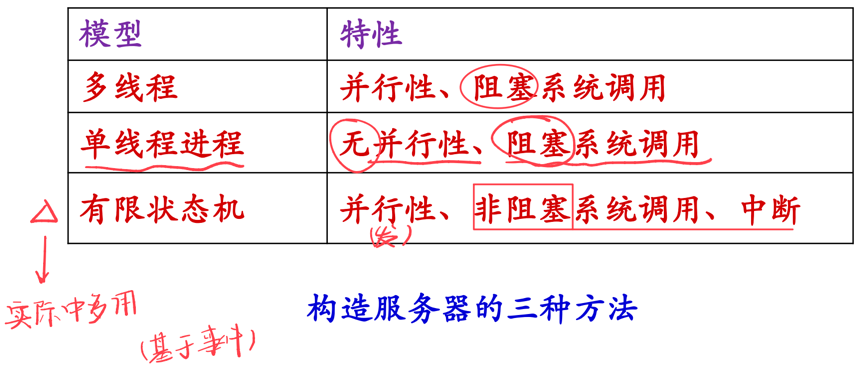
### ﻿线程的引入

* + - 1. ﻿为什么在进程中再派生线程？
         1. ﻿应用的需要（一个进程下有多个任务需要快速响应完成）

﻿Web服务器

* ﻿从客户端接收网页请求（协议？）
* ﻿从磁盘上检索相关网页，读入内存
* ﻿将网页返回给对应的客户端

﻿如何提高服务器工作效率？

* ﻿网页缓存(Web page Cache)
* ﻿多线程
* 各种模型的对比
  + - * 1. 开销的考虑

1. ﻿进程相关的操作（﻿创建进程、撤消进程、进程通信、进程切换…）﻿——时间/空间开销大，限制了并发度的提高
2. ﻿线程的开销小

* ﻿创建/撤销一个新线程花费时间少
* ﻿两个线程的切换花费时间少
* ﻿线程之间相互通信无须调用内核（同一进程内的线程共享内存空间和文件）
  + - * 1. 性能的考虑——多核、多处理器√

### ﻿线程的基本概念

#### ﻿进程的两个基本属性

﻿资源的拥有者

#### ﻿一个虚拟地址空间，一些占有的资源（文件， I/O设备），保护

﻿调度单位（对于单线程进程而言）

* ﻿一个执行（路径）轨迹，状态、优先级

#### ﻿将原来进程的两个属性分别处理——线程

线程：进程中的一个运行实体，是CPU的调度单位，也叫﻿轻量级进程

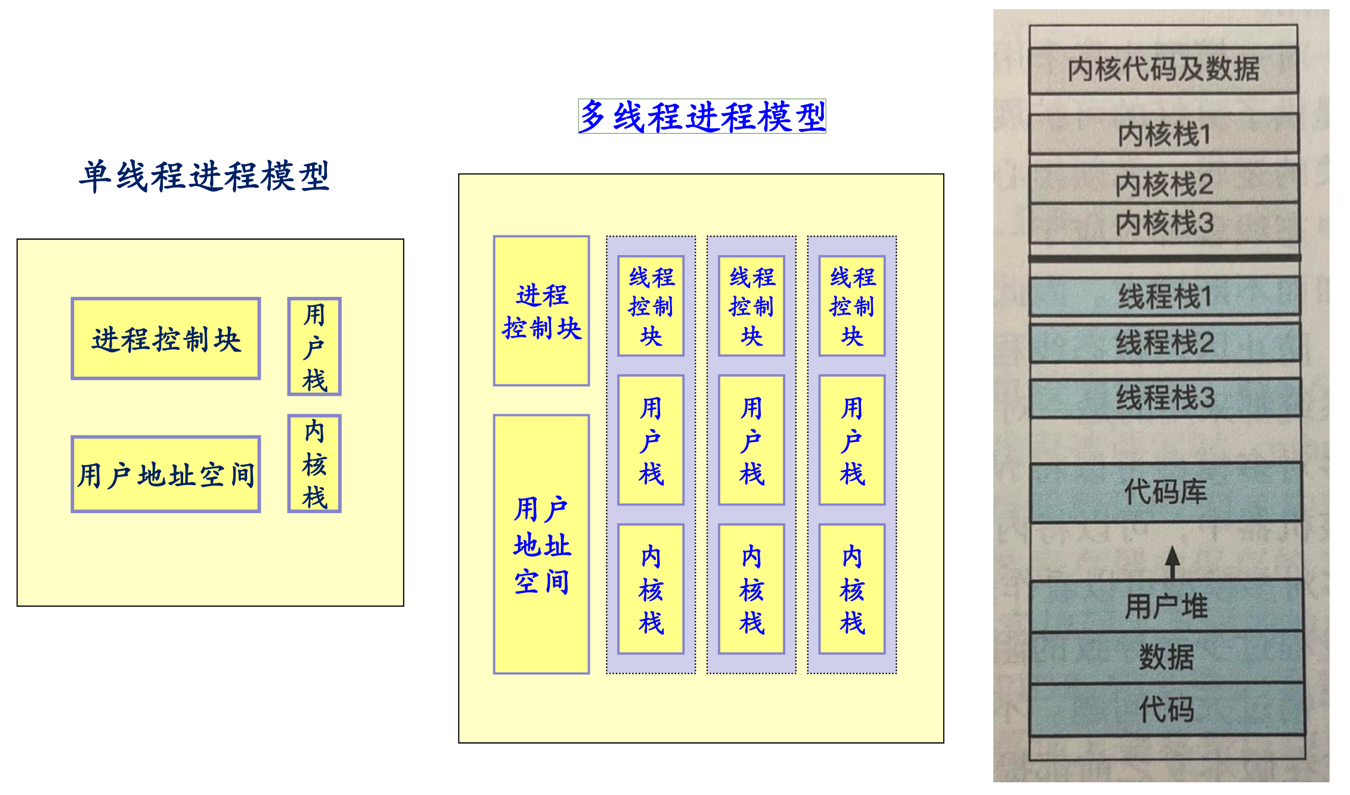
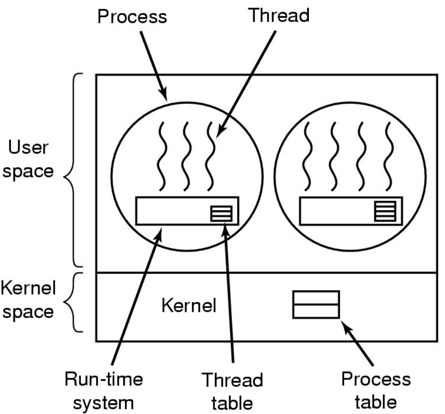
#### 线程的属性

﻿有状态及状态转换 → 需要提供一些操作

﻿不运行时需要保存的上下文（相比进程而言更少﻿e.g.程序计数器等寄存器）

﻿有自己的栈和栈指针（与进程栈“不交”）

﻿共享所在进程的地址空间和其他资源

﻿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）

### ﻿线程的实现

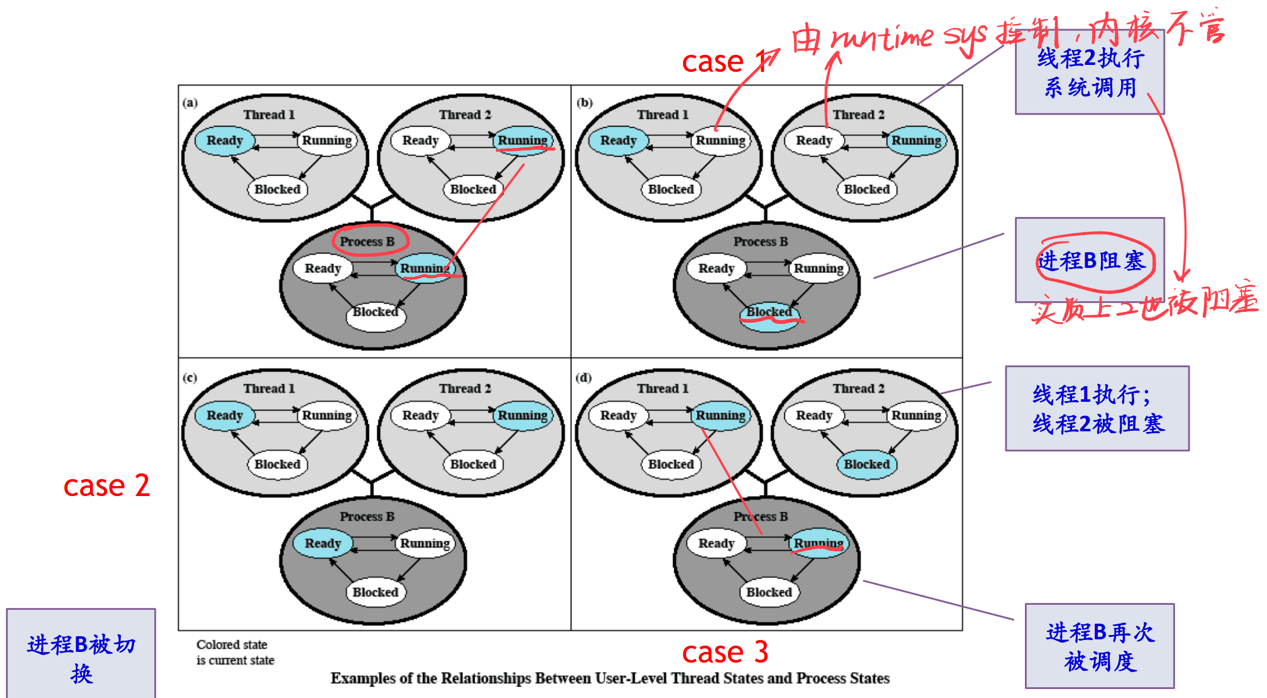
#### ﻿用户级线程﻿（User Level Thread）

﻿在用户空间实现

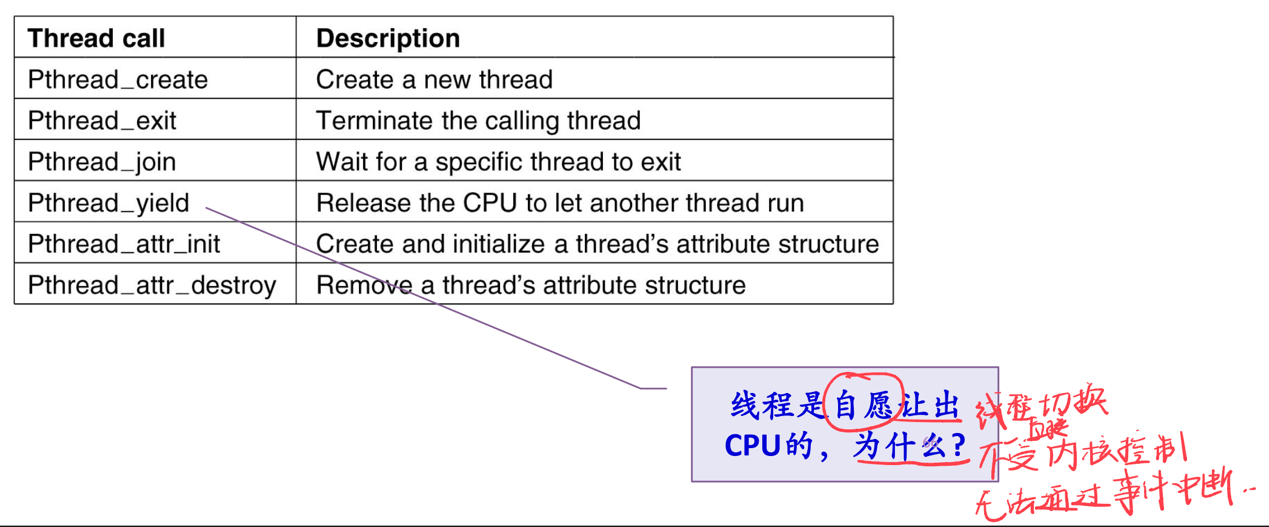
﻿在用户空间建立线程库：提供一组管理线程的函数

﻿运行时系统：完成线程的管理工作（操作、线程表）

﻿**内核**管理的是进程，**不知道（用户）线程的存在**

﻿线程切换不需要内核态特权

﻿例子：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

优点

#### ﻿线程切换快

#### ﻿调度算法是应用程序特定的

* + 1. ﻿用户级线程可运行在任何操作系统上（只需要实现线程库）

﻿缺点

﻿大多数系统调用是阻塞的，因此，由于内核阻塞进程，故进程中所有线程也被阻塞

﻿内核只将处理器分配给进程，同一进程中的两个线程不能同时运行于两个处理器上

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）

#### ﻿核心级线程

在内核中实现

﻿内核管理所有线程，并向应用程序提供API接口

内核维护进程和线程的上下文

线程的切换需要内核支持

以线程为基础进行调度

例子： Windows

#### 混合—两者结合方法

在内核中实现，支持用户线程

﻿线程创建在用户空间完成

﻿线程调度等在核心态（Kernel thread）完成

例子：Solaris

## 协程﻿（coroutine）



## 重点小结