

Operating System

进程

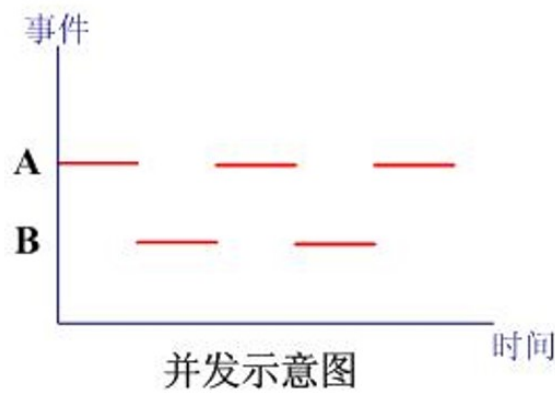
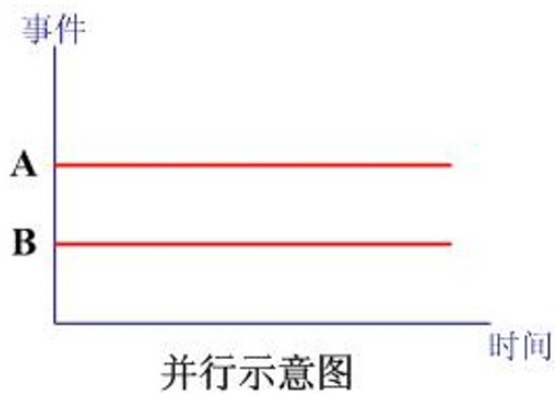
进程的概念

进程是一个具有一定独立功能的程序关于某个数据集合上的一次运行活动，进程是系统进行资源分配和调度的一个独立单位。

并发与并行

并发：多个进程在一个CPU上运行，但是任意时刻只有一个进程在CPU上运行。

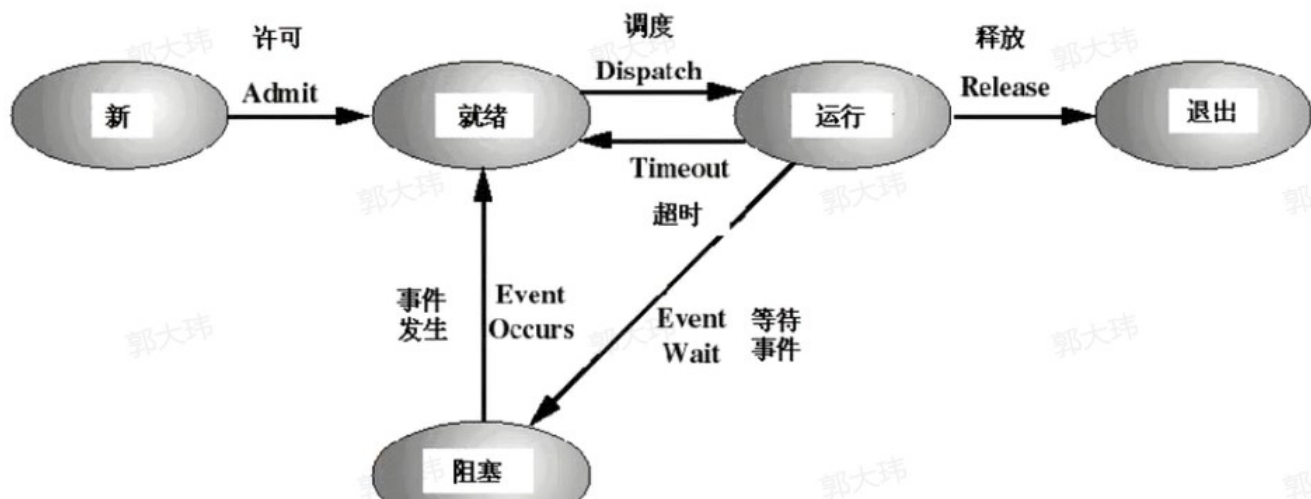
并行：多个进程同时运行。

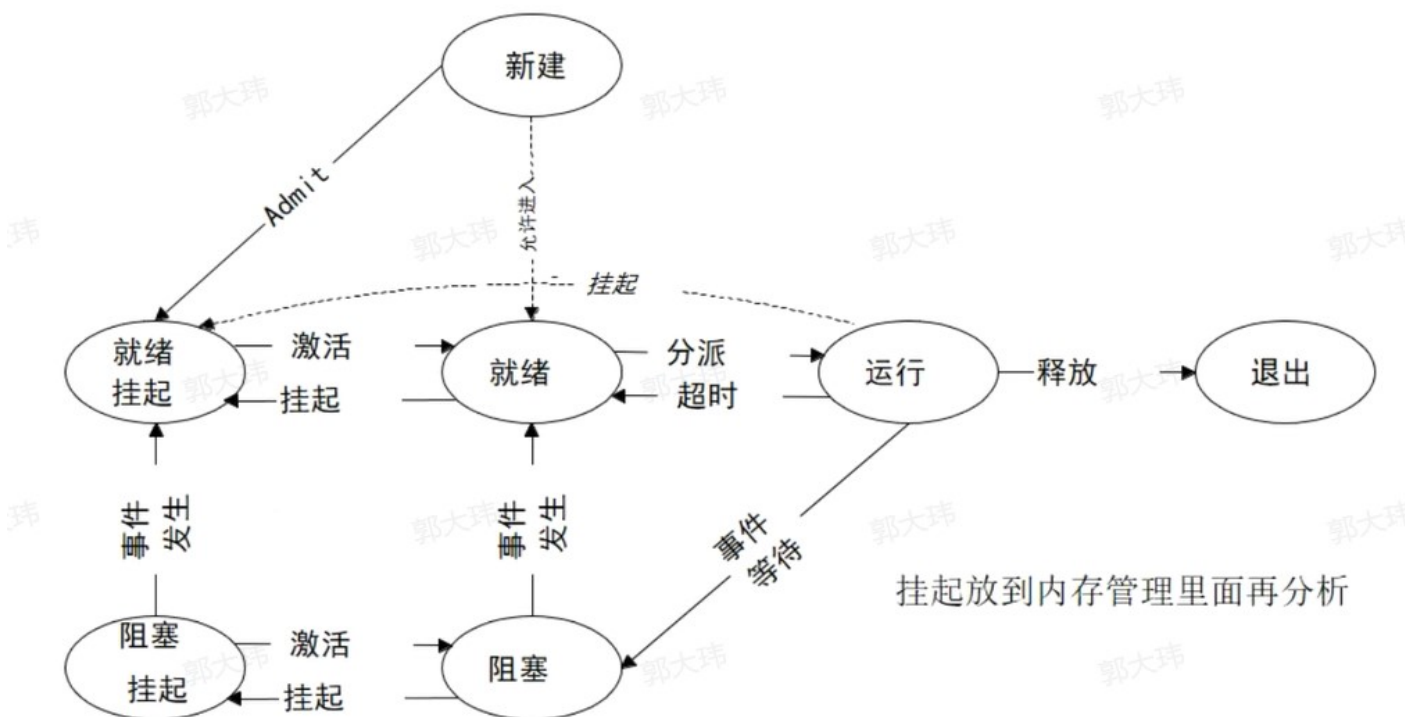
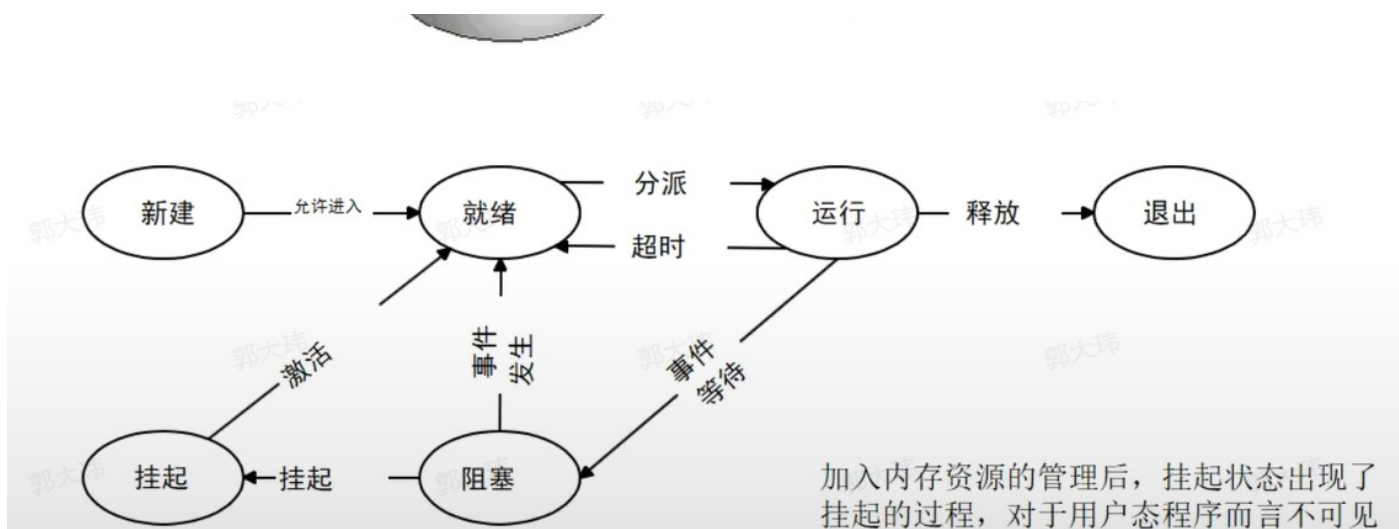


进程的特征

- 虚拟：每个进程有自己虚拟的地址空间、虚拟的设备，以为自己掌控了这台计算机。
- 并发：多个进程是在并发交替的执行以提升效率。
- 共享：资源（设备、内存）在进程之间共享，分时复用。
- 不确定：进程无法预知调度在什么时刻发生。

进程状态模型





进程调度算法

- 先来先服务
- 短任务优先，剩余时间短任务优先
- 时间片轮转
- 优先级队列

调度算法的设计原则：最优解，可以获取的参数，获取和推理的代价。

调度的驱动力：进程主动调度（进程主动让出，或者进程发起 syscall 而 os 认为有必要调度），或者中断抢占调度。

比较调度算法的几个非要重要的指标体系

CPU 使用率：CPU 处于忙状态的时间百分比

吞吐量：单位时间内完成的进程数量

周转时间：进程从初始化到结束（包括等待）的总时间

等待时间：进程在就绪队列中的总时间

响应时间：从提交请求到产生响应所花费的总时间

进程切换

- 进程切换（上下文切换）
 - 暂停当前运行进程，从运行状态变成其他状态
 - 调度另一个进程从就绪状态变成运行状态
- 进程切换的要求
 - 切换前，保存进程上下文
 - 切换后，恢复进程上下文
 - 快速切换
- 进程生命周期的信息
 - 寄存器 (PC, SP, ...)
 - CPU 状态
 - 内存地址空间

```
.text
.globl switch_to
switch_to:
    # switch_to(from, to)

    # save from's registers
    movl 4(%esp), %eax    # eax points to from
    popl 0(%eax)          # save eip !popl
    movl %esp, 4(%eax)
    movl %ebx, 8(%eax)
    movl %ecx, 12(%eax)
    movl %edx, 16(%eax)
    movl %esi, 20(%eax)
    movl %edi, 24(%eax)
    movl %ebp, 28(%eax)

    # restore to's registers
    movl 4(%esp), %eax    # not 8(%esp): popped return address already
                          # eax now points to to
    movl 28(%eax), %ebp
    movl 24(%eax), %edi
    movl 20(%eax), %esi
    movl 16(%eax), %edx
    movl 12(%eax), %ecx
    movl 8(%eax), %ebx
    movl 4(%eax), %esp

    pushl 0(%eax)          # push eip
    ret
```

进程的切换代价大于线程的切换代价，因为进程的切换需要切换内存地址空间（切换 CR3, 刷新 TLB），而线程的切换不需要切换内存地址空间。

进程间通信

- 信号
- 管道
- 消息队列
- 共享内存

线程

线程的实现方案

用户级线程（ULT）

线程的管理全部由用户程序完成，核心部分只对进程管理，但增加“线程库”概念。

特点：

- 进程表在核心区
- 线程表在用户区
- 线程执行需要一个支持系统（线程库）

优点：

- 线程切换不需要内核模式特权。
- 线程调用可以是应用程序级的，根据需要可改变调度算法，但不会影响底层的操作系统调度程序。
- ULT管理模式可以在任何操作系统中运行，不需要修改系统内核，线程库是提供应用的实用程序。
-

缺点：

- 系统调用会引起进程阻塞
- 这种线程不利于使用多处理器并行

内核级线程（KLT）

线程由OS内核进行管理，内核给应用程序级提供系统调用，实现对线程的使用。

特点：

- 线程和进程都在用户空间完成
- 进程表和线程表都放在内核空间
- 线程在内核中有保存的信息，系统调度是基于线程完成的。

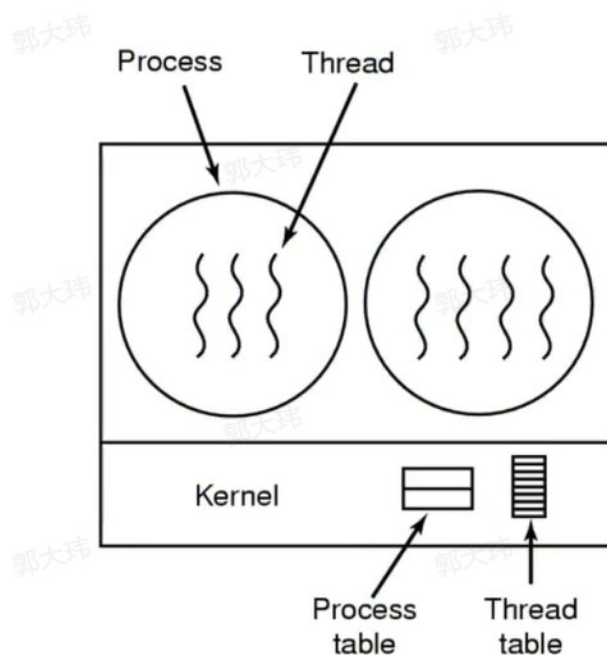
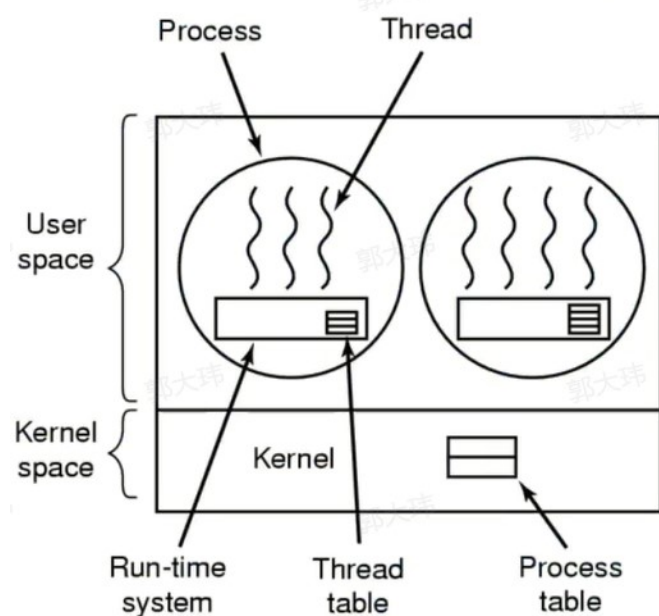
优点：

- 可克服ULT的两个缺点，且内核程序本身也可以是多线程结构的。

缺点：

- 线程间的控制转换需要转换到内核模式。

• 用户态线程 vs 内核态线程



组合式

略

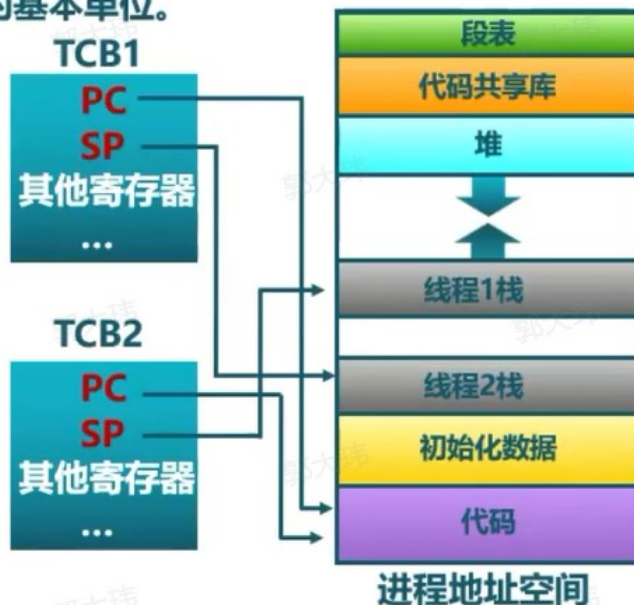
纤程 (Fiber, ucontext)

用户态线程的一种实现方式，用户态线程的切换不需要内核态的切换，只需要用户态的切换，因此切换代价小。

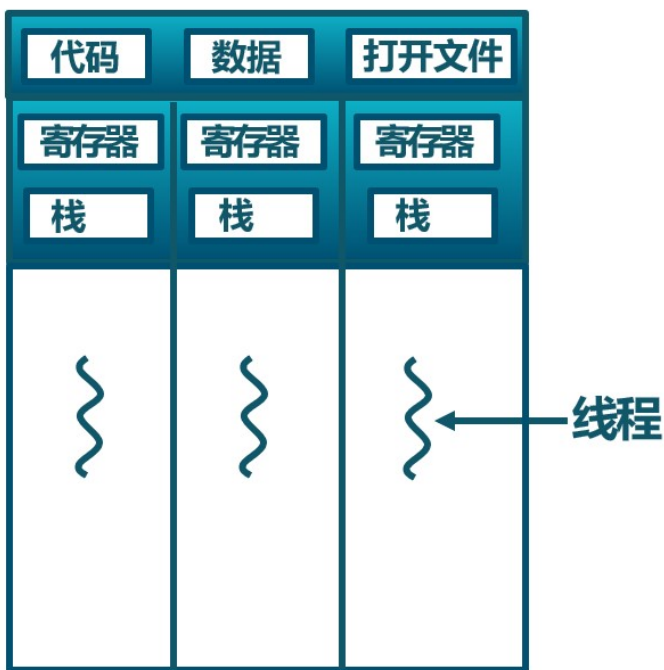
- 发挥ULT快速切换的优势
- 在编程时提出对程序员的要求，要求他们妥善的设计代码（编译器不允许内核态的切换）

线程是进程的一部分，描述指令流执行状态。它是进程中的**指令执行流**的最小单元，是CPU调度的基本单位。

- ▣ 进程的资源分配角色：
进程由一组相关资源构成，包括地址空间（代码段、数据段）、打开的文件等各种资源
- ▣ 线程的处理机调度角色：
线程描述在进程资源环境中的指令流执行状态



单线程进程



多线程进程

为了保持线程执行的独立性，每个线程有自己的堆栈
OS 记录一个进程的执行状态所需的数据：

- 上下文（寄存器组）保存当前状态
- 栈（用于保存函数调用的序列和局部变量）

线程与进程的比较

- 进程是资源分配单位，线程是 CPU 调度单位
- 进程拥有一个完整的资源平台，而线程只独享指令流执行的必要资源，如寄存器和栈
- 线程具有就绪、等待和运行三种基本状态和状态间的转换关系
- 线程能减少并发执行的时间和空间开销
 - 线程的创建时间比进程短
 - 线程的终止时间比进程短
 - 同一进程内的线程切换时间比进程短
 - 由于同一进程的各线程间共享内存和文件资源，可不通过内核进行直接通信

系统调用

系统调用和保护机制

- 如何实现内存的保护？设置不同的权限
- 系统调用：切换系统的权限状态
 - 系统调用的本质是一个函数，一个由 os 提供的函数
 - 提升权限的过程是一个中断，一个用户程序可以主动发起的中断
 - 系统调用的意义是什么？
 - 系统调用的代价是什么？
 - 系统调用（权限转换）有没有高性能的实现方式？（另一个有趣的科研话题，比如 Intel 的 Sysenter & sysexit）

sfs_filetest1.c: ret=read(fd,data,len);

```
.....
8029a1: 8b 45 10          mov    0x10(%ebp),%eax
8029a4: 89 44 24 08       mov    %eax,0x8(%esp)
8029a8: 8b 45 0c          mov    0xc(%ebp),%eax
8029ab: 89 44 24 04       mov    %eax,0x4(%esp)
8029af: 8b 45 08          mov    0x8(%ebp),%eax
8029b2: 89 04 24          mov    %eax,(%esp) ; 以上代码在填充栈
8029b5: e8 33 d8 ff ff    call  8001ed <read> ; 这里的read是宏==6
```

syscall(int num, ...) {

```
...
    asm volatile (
        "int %1; //这一句会个产生一个中断\n"
        : "=a" (ret)
        : "i" (T_SYSCALL), //这句指定中断号
          "a" (num), //这一句的作用是把6放进了eax
          "d" (a[0]),
          "c" (a[1]),
          "b" (a[2]),
          "D" (a[3]),
          "S" (a[4])
        : "cc", "memory");
    return ret;
```

进程同步

临界区

临界区的访问规则

- 空闲则入：没有进程在临界区时，任何进程可进入
- 忙则等待：如果有进程在临界区，其他进程均不能进入临界区
- 有限等待：等待进入临界区的进程不能无限期等待
- 让权等待（可选）：不能进入临界区的进程，应释放CPU（如转换到阻塞状态）

关中断

不安全

Peterson算法

```
// flag initialized to [false, false]
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
    // 临界区
    flag[i] = false;
    // 余下区
} while (true);
```

Dekker算法

```
// flag initialized to [false, false]
do {
    flag[i] = true;
    while (flag[j]) {
        if (turn != i) {
            flag[i] = false;
            while (turn != j);
            flag[i] = true;
        }
    }
    // 临界区
    turn = j;
    flag[i] = false;
    // 余下区
} while (true);
```

Test and set (TAS) 指令

TSL reg, lock

- 将 lock 的值存入 reg
- 将 lock 的值设为 1
- 返回 reg 的旧值

```
enter_cr:
TSL reg, lock
CMP reg, #0
JNE enter_cr
RET
```

```
leave_cr:
MOV lock, #0
RET
```

锁

TSL 实现自旋锁（忙等待）

```

Lock::Acquire() {
    while (test_and_set(&lock));
}
Lock::Release() {
    lock = 0;
}

```

线程在等待的时候消耗CPU时间

TSL 实现自旋锁（无忙等待）

```

Lock::Acquire() {
    while (test_and_set(&lock)) {
        add_to_queue();
        sleep();
    }
}

Lock::Release() {
    lock = 0;
    remove_from_queue();
    wakeup();
}

```

Sleep-Wakeup

Primitive: atomic action supported by OS

Sleep(): block the caller process

Wakeup(PID):wake the process whose ID is PID

```

// Producer
while (true) {
    produce();
    if (count == N) sleep();
    append();
    if (count == 1) wakeup(consumer);
}

```

```

// Consumer
while (true) {
    if (count == 0) sleep();
    take();
    if (count == N - 1) wakeup(producer);
    consume();
}

```

但是如果出现 wakeup 丢失的情况，就会出现死锁。

信号量


```

class Semaphore {
    int value;
    QueueType queue;
}
Semaphore::P() {
    value--;
    if (value < 0) {
        add_to_queue();
        sleep();
    }
}
Semaphore::V() {
    value++;
    if (value <= 0) {
        remove_from_queue();
        wakeup();
    }
}
}

```

信号量由操作系统实现。

信号量实现互斥锁

```

mutex = new Semaphore(1);
mutex->P();
// 临界区
mutex->V();

```

信号量实现同步

```

cond = new Semaphore(0);
{
    // Thread 1
    ...
    cond->P();
    ...
}
{
    // Thread 2
    ...
    cond->V();
    ...
}

```

信号量解决生产者消费者问题

```

mutex = new Semaphore(1);
empty = new Semaphore(N);
full = new Semaphore(0);

```

```

// Producer
while (true) {
    empty->P();
    mutex->P();
    produce();
    mutex->V();
    full->V();
}

```

```
// Consumer
while (true) {
    full->P();
    mutex->P();
    consume();
    mutex->V();
    empty->V();
}
```

管程

管程是一种高级的同步工具，它是一种数据结构，由数据和对数据的操作组成，数据及对数据的操作被封装在一个模块中，只有通过管程中的操作才能访问数据。

条件变量

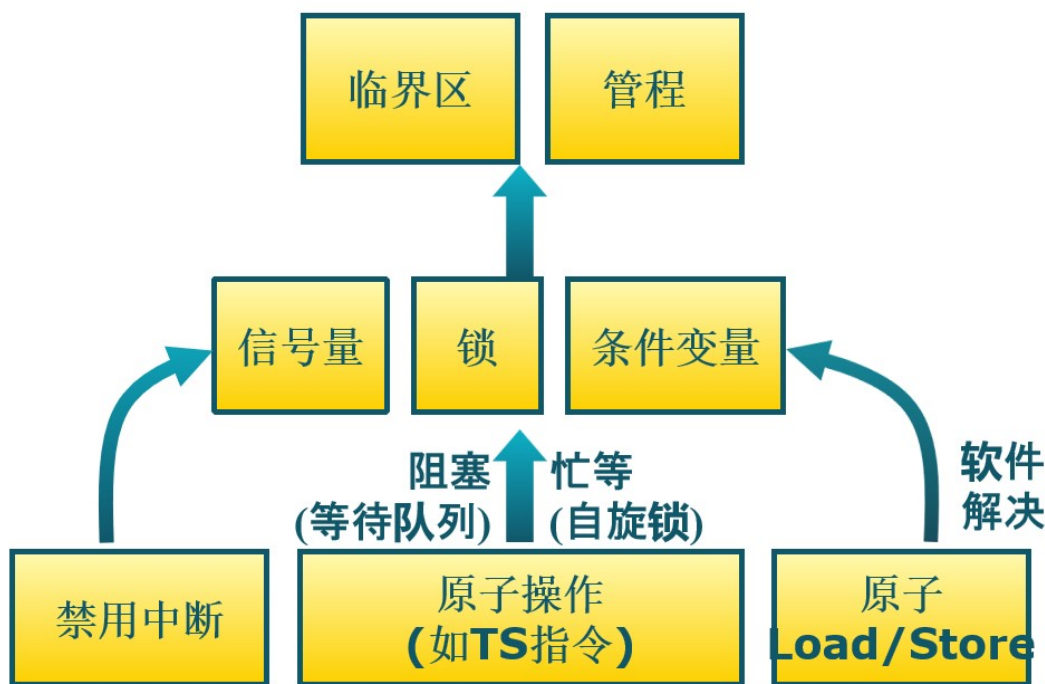
条件变量是管程内的等待机制

基本同步方法

并发编程

高层抽象

硬件支持



内存管理

几个地址概念

- 逻辑地址：逻辑地址是段内偏移地址。它是程序员在编程时使用的地址。
- 虚拟地址：虚拟地址是在虚拟内存系统中使用的地址。
- 线性地址：在分段内存管理系统中（如早期的 x86 架构的保护模式），线性地址是虚拟地址的一种形式，它是通过分段机制转换后的地址。
- 物理地址：物理地址是最终用于访问计算机系统中实际物理内存的地址。它是存储单元在物理内存中的实际位置。

他们之间的转换关系：

x86: 逻辑地址 --> (段转换) --> 线性地址 --> (页转换) --> 物理地址

rv: 虚拟地址 --> (页表转换) --> 物理地址

连续内存分配

动态分区分配

连续内存分配：动态分区分配

■ 动态分区分配

- 当程序被加载执行时，分配一个进程指定大小可变的分区(块、内存块)
- 分区的地址是连续的

■ 操作系统需要维护的数据结构

- 所有进程的已分配分区
- 空闲分区(Empty-blocks)



First-fit

- 分配和释放的时间性能较好，较大的空闲分区可以被保留在内存高端。
- 但随着低端分区不断划分而产生较多小分区，每次分配时查找时间开销会增大。

Next-fit

- 该算法的分配和释放的时间性能较好，使空闲分分布得更均匀，但较大的空闲分区不易保留。

Best-fit (最小空间)

- 个别来看，外碎片较小，整体来看，会形成较多外碎片。但较大的空闲分区可以被保留。

Worst-fit (最大空间) (最常用)

- 基本不留下小空闲分区，但较大的空闲分区不会被保留。

碎片整理：紧凑(compaction)

■ 碎片整理

- 通过调整进程占用的分区位置来减少或避免分区碎片

■ 碎片紧凑(紧缩)

- 通过移动分配给进程的内存分区，以合并外部碎片
- 碎片紧凑的条件



- 所有的应用程序可动态重定位

郭大玮

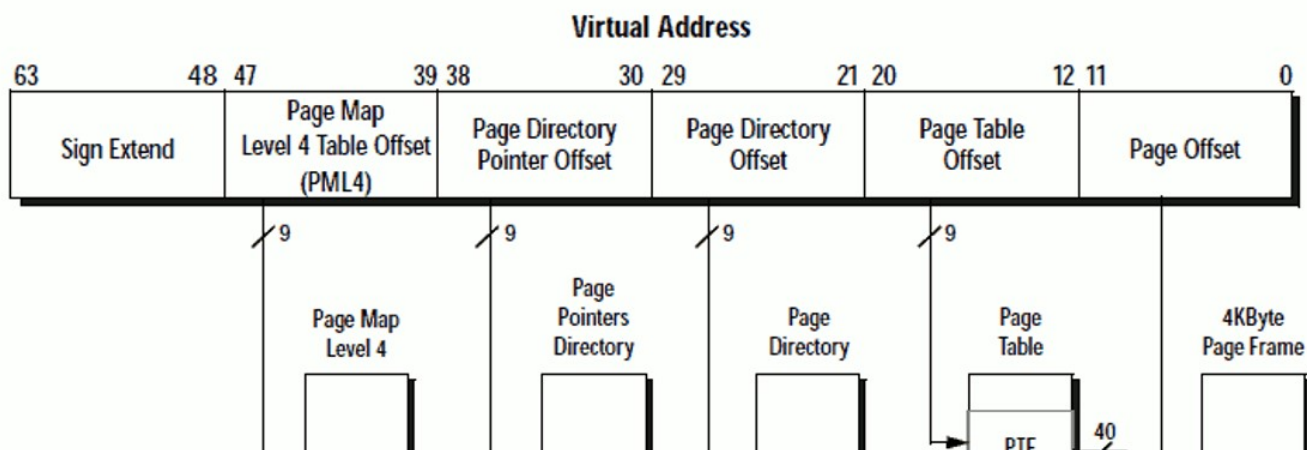
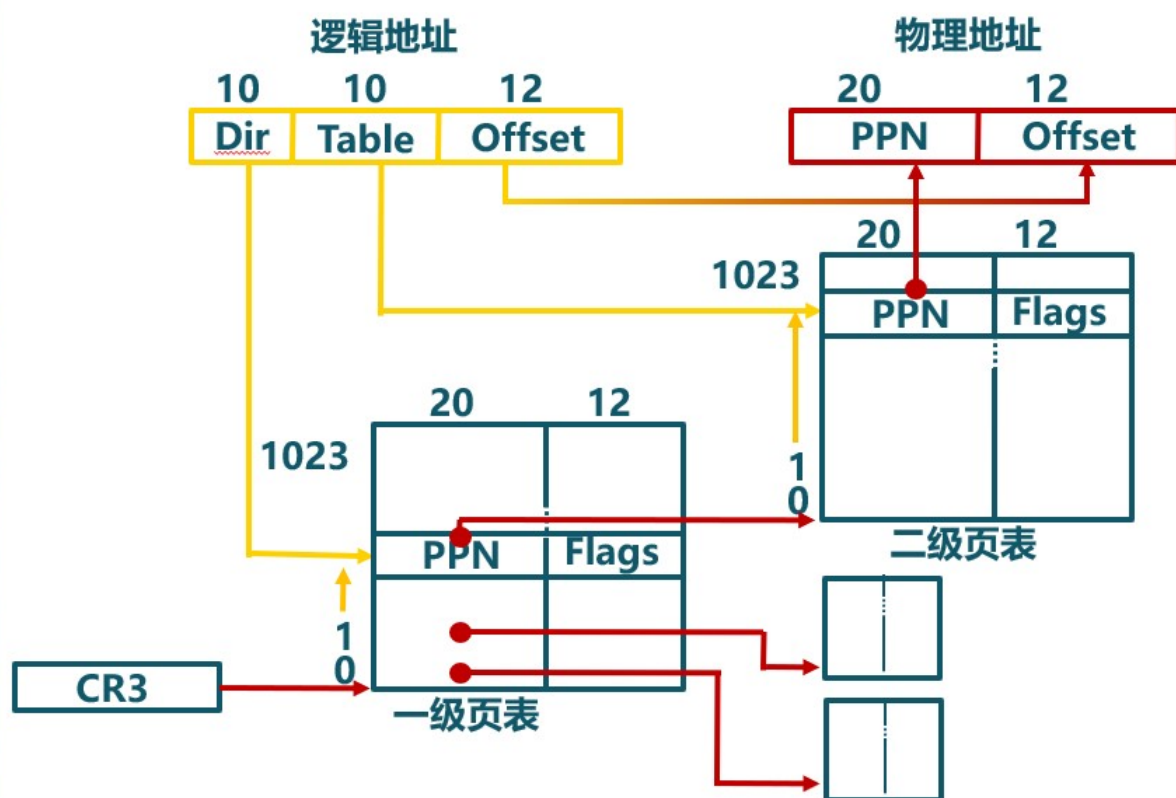
0

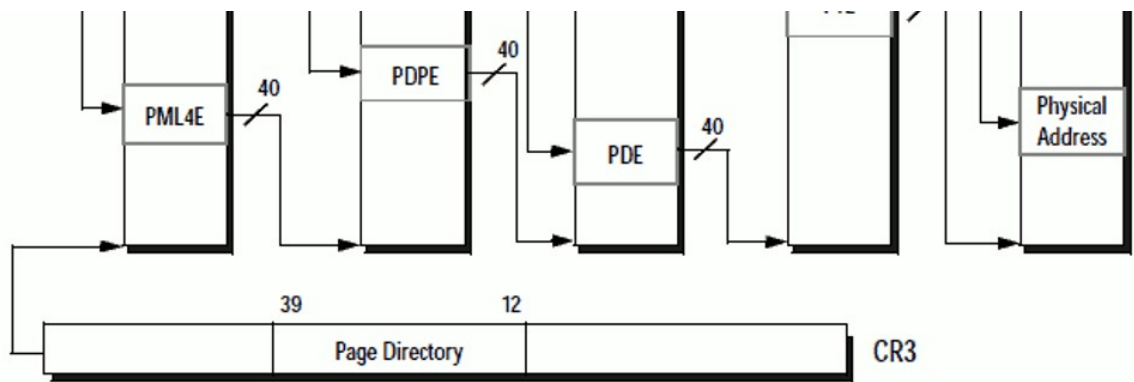


虚拟内存

- 覆盖式
- 分页式
- 分段式
- 段页式

Case Study: X86页表结构

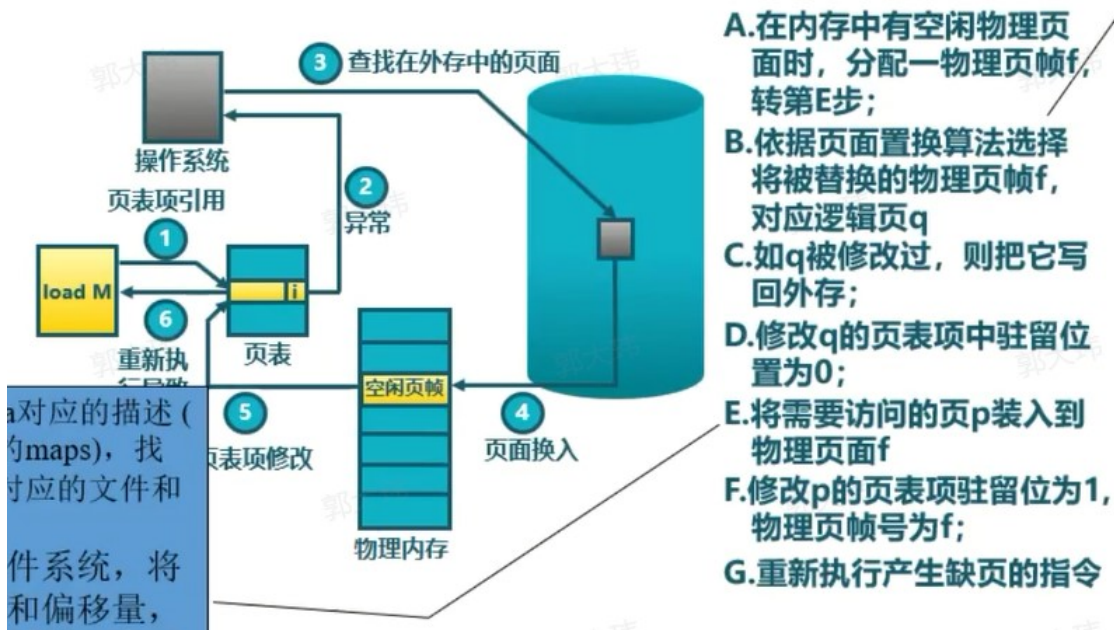




x64 的页表是什么样的

- 每个页表里面有 512 项
- $4k / 8 \text{ 字节} (64\text{bit}) = 512$
- 因此用于页表项的寻址部分仅有 9 位
- 类似的，页目录项也只有 512 项
- 页目录项也只有 9 位
- 页表扩展为 4 级，每级都是 9 位地址
- 寻址空间最大为 256T
- 最高的 16 位目前没有使用

缺页异常（缺页中断）的处理流程



页面置换算法

最佳置换算法 (OPT)

选择在未来最长时间内不再被访问的页面予以置换。

先进先出置换算法 (FIFO)

选择在内存中驻留时间最长的页面予以置换。

- 有 Belady 现象

最近最久未使用置换算法 (LRU)

选择最近一段时间内最久未使用的页面予以置换。

- 没有 Belady 现象

最不常用置换算法 (LFU)

选择在最近一段时间内访问次数最少的页面予以置换。

时钟置换算法 (Clock)

页面装入内存时，访问位初始化为 0

访问页面（读/写）时，访问位置 1

缺页时，从指针当前位置顺序检查环形链表

- 访问位为 0，则置换该页
- 访问位为 1，则访问位置 0，指针移动到下一个页面，直到找到可置换的页面

改进型时钟置换算法 (Enhanced Clock)

在时钟置换算法的基础上，增加了修改位的判断

- 访问位为 0，修改位为 0，则置换该页
- 访问位为 0，修改位为 1，则修改位置 0，继续检查下一个页面
- 访问位为 1，修改位为 0，则访问位置 0，继续检查下一个页面
- 访问位为 1，修改位为 1，则访问位置 0，继续检查下一个页面








































工作集

当前时刻前 τ 个内存访问的页引用是工作集， τ 被称为窗口大小

- 访存链表：维护窗口内的访存页面链表
- 访存时，换出不在工作集的页面；更新访存链表
- 缺页时，换入页面；更新访存链表

工作集置换算法

$\tau = 4$

时间		0	1	2	3	4	5	6	7	8	9	10
访问页面			c	c	d	b	c	e	c	e	a	d
逻辑 页面 状态	页面a											
	页面b	$t=0$										
	页面c											
	页面d											
页状态	页面e	$t=-1$										
		$t=-2$										
缺页状态												

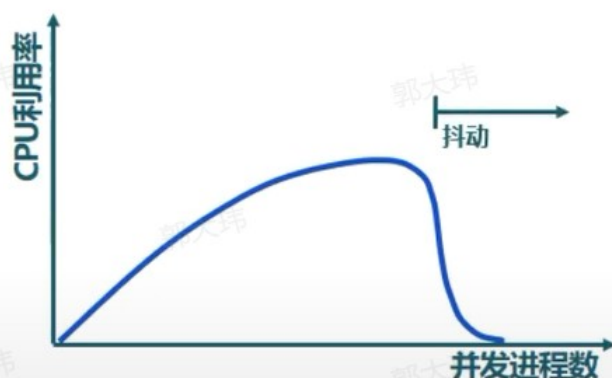
缺页率算法

- 缺页率 = 缺页次数 / 总访问次数
- 通过调节常驻集大小，使每个进程的缺页率保持在一个合理的范围内
 - 若进程缺页率过高，则增加常驻集以分配更多的物理页面
 - 若进程缺页率过低，则减少常驻集以减少它的物理页面数
- 访存时，设置引用位标志
- 缺页时，计算从上次缺页时间 t_{last} 到现在 $t_{current}$ 的时间间隔
 - 如果 $t_{current} - t_{last} > T$ ，则置换所有在 $[t_{last}, t_{current}]$ 时间内没有被引用的页
 - 如果 $t_{current} - t_{last} \leq T$ ，则增加缺失页到工作集中

	Principle	Performance
最优方法	预知未来情况的上限	永远不可能达到
FIFO	按时间排序，最直接最简单	来得最早，和最不可能用到之间没有关系
LRU & LFU	利用过去使用的频度和次数预测未来	目前可达的最佳效果，代价略高
CLOCK	FIFO 与 LFU 的折衷	足够简单，也有一定的合理性，在 OS 中大量使用
Working set	只给进程需要的内存	不现实
Page Fault Frequency	根据缺页事情的频度预测内存的需求	相对现实

抖动

CPU利用率与并发进程数的关系



■ CPU利用率与并发进程数存在相互促进和制约的关系

- ▣ 进程数少时，提高并发进程数，可提高CPU利用率
- ▣ 并发进程导致内存访问增加
- ▣ 并发进程的内存访问会降低访存的局部性特征

Belady现象

Belady现象是指在采用一些算法（如 FIFO）时，当分配给进程的物理块数增加时，反而出现缺页次数增加的现象。

原因：FIFO算法的置换特征与进程访问内存的动态特征矛盾，被它置换出去的页面并不一定是进程近期不会访问的，导致Belady的原因是，页框数量增加后，留在内存当中的内容与增加之前无关，从而导致缺页事件的不可预测（增加页框，得到的留在内存中的页面，不一定是增加之前页面集合的超集）

文件系统

存储方式

Summary of file physical structure

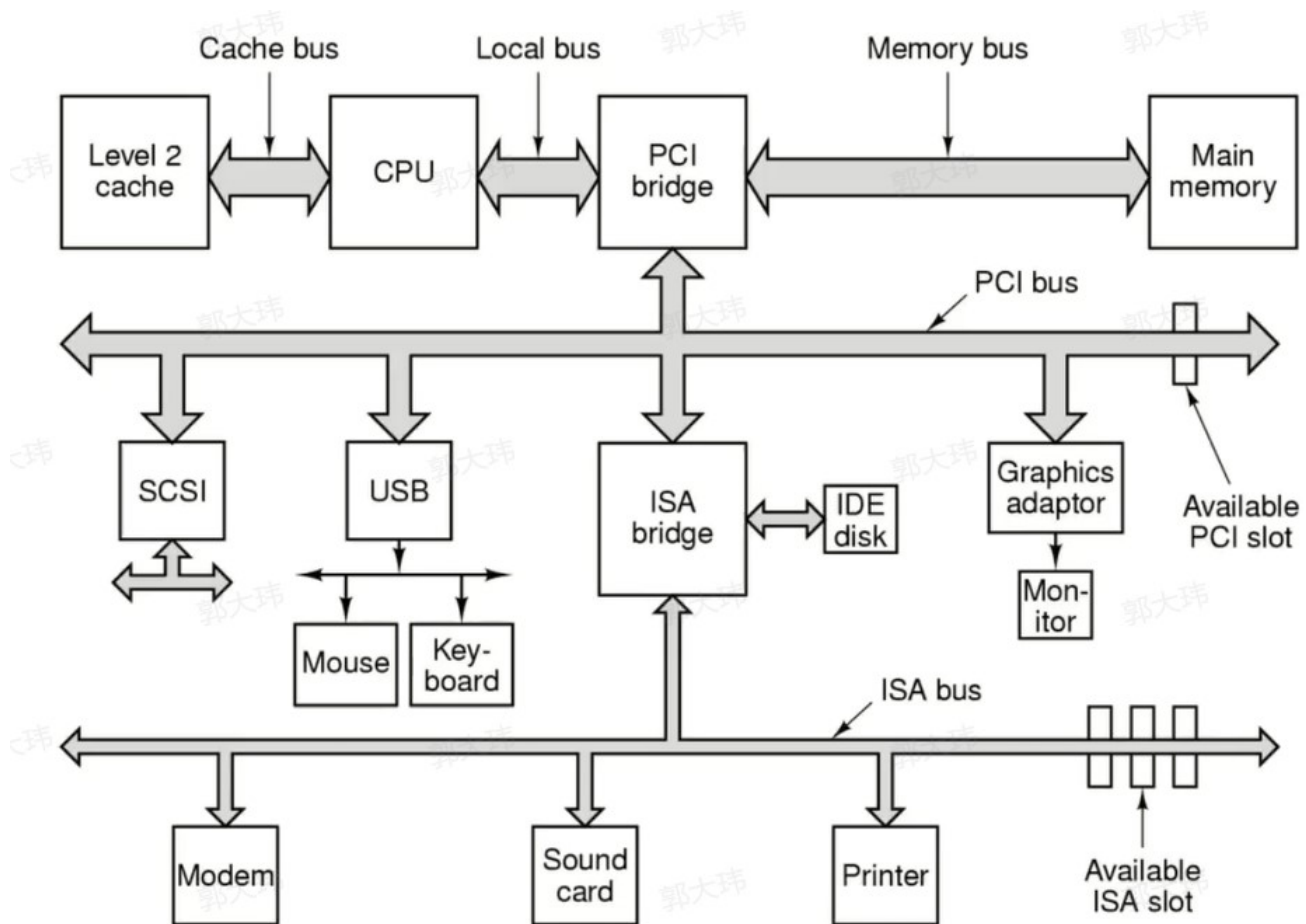
		Continuous	Link table	Index table
media	tape	Supported	Unsupported	unsupported
	disk	Supported	supported	supported
Access mode		Sequential & random	sequential	Sequential & random
Efficiency		Low	Middle	High
Application		To simple to be used	Not popular	Widely used

设备管理

- IO资源的管理
 - 独立编址、共享编址、混合式
- 三种设备的通信模式
 - 忙等待、中断、DMA
- 驱动程序设计中的典型问题
 - 异步、缓冲、spooling
- 磁盘管理中的技术问题

收益管理中的若干问题

- RAID
- 磁臂管理



方法	资源分配策略	各种可能模式	主要优点	主要缺点
预防 Prevention	保守的; 宁可资源闲置(从机制上使死锁条件不成立)	一次请求所有资源<条件 1>	适用于作突发式处理的进程; 不必剥夺	效率低; 进程初始化时间延长
		资源剥夺<条件 3>	适用于状态可以保存和恢复的资源	剥夺次数过多; 多次对资源重新启动
		资源按序申请<条件 4>	可以在编译时(而不必在运行时)就进行检查	不便灵活申请新资源
避免 Avoidance	是“预防”和“检测”的折衷(在运行时判断是否可能死锁)	寻找可能的安全的运行顺序	不必进行剥夺	使用条件: 必须知道将来的资源需求; 进程可能会长时间阻塞
检测 Detection	宽松的; 只要允许, 就分配资源	定期检查死锁是否已经发生	不延长进程初始化时间; 允许对死锁进行现场处理	通过剥夺解除死锁, 造成损失
忽略 ignore	不理睬死锁问题, 认为它不是主要问题(鸵鸟法)			

综合案例分析

- 系统启动过程分析
- 页面置换过程分析
- 进程切换过程分析
- 权限切换过程分析
- 数据读写过程分析