课堂笔记

2020年9月20日

9:47

**W1**

操作系统：对硬件资源进行管理，利用资源提供服务

对象：资源+进程

进程：主体，PCB（进程控制），状态，变迁，线程并发性

解决方法：死锁，饥饿

资源：种类多=CPU+内存+IO

内存：所有程序载入内存形成进程，程序（静态），进程（动态）

物理内存+虚拟内存 部分加载

管理方法：页式管理，段式管理，段页式管理

CPU：最重要资源 OS目标：如何提升利用率

CPU状态=运行+空闲=运行（系统指令+用户指令）+空闲

利用率=执行用户指令时间/总时间，默认单CPU

调度->多进程服务

IO：速度慢，种类多

缓冲，磁盘，文件

计组：CPU，内存，IO，系统总线，数据交换通路->地址，内容

地址寄存器，内容寄存器

CPU三对六个寄存器：MAR，MBR：CPU，内存；IOAR，IOBR：内存，IO；PC，IR：指令，执行

PC（next），执行：取指+执行（原子性）

串行->中断->并行（逻辑，物理）

多道程序设计->单道程序设计

多进程：进程数量=道数（CPU利用率）

指令执行：2阶段->3阶段

中断：暂停/中断现行 多中断/嵌套中断

CPU快，IO慢，CPU等IO

3阶段执行：开中断，中断源，优先级高

三步骤：处理中断：保护现场（软硬件结合），转中断服务程序，恢复现场

进程切换 CPU的利用率计算（三状态：系统，用户，空闲） 串行->并行

存储器三要素：容量，价格，存取时间

金字塔结构：CPU-内部（寄存器，cache，内存）-外部（磁盘，光盘）-离线（磁带）

平均访问时间：不同时间\*次数/总次数

执行频率低，整体性能有关

Cache：目标：低成本的设备，达到高成本设备效果

有效性能、有效平均时间：要素->命中率

局部性原理：时间局部性：正在执行的指令，不久将来有很大概率又会被执行

空间局部性：正在执行的指令存储区域的周围，其所存储的指令在不久将来有很大概率被执行到->短时间内，所执行指令在存储器上具有集簇性倾向，使用块进行传递 块写缓读

cache大小，块大小，映射函数（边界法，极限法），替换，级联

IO：CPU从IO工作中解脱出来的解放历史，CPU减少参与IO工作

可编程IO：CPU不停轮询IO设备，CPU忙闲，利用率低

中断IO：变IO设备状态从被动到主动，IO主动汇报（中断），IO设备不就绪可做其他，基于字中断，次数频率高

DMA：基于块中断，次数和频率降低，CPU减少参与中断，CPU利用率上升

DMA传数据->系统总线

CPU执行其他事务->指令或数据->系统总线

争抢：CPU权限高，争抢时：总线空，CPU使用；总线忙，CPU等待一周期

PCB：进程，终止：正常退出，异常退出，被迫退出

**W2**

进程控制与描述：进程定义：进程=代码+数据的可执行实体

进程控制块PCB：描述进程，存储进程信息，现场恢复和保护

PCB信息：进程ID（唯一标识），状态：就绪、运行，阻塞

优先级：基于优先级<->优先数

PCB由OS创建，存储进程信息的数据结构，OS用此对进程进行管理和控制

通过进程组合的的运行轨迹，CPU行为交替执行

两状态=运行态，非运行态

进程状态变迁图：圆圈（状态），箭头（有向）（状态变迁），线上的字（变迁原因）

变迁：运行<->非运行/暂停（运行->就绪/阻塞），调度（分派），给予CPU的使用权

两个问题：创建（新…)，退出（正常，异常，被迫）

缺陷：非运行状态下，有一类即使给予CPU控制权也无法运行，效率浪费：非运行->就绪/阻塞；两状态->五状态（新建，就绪，运行，阻塞，退出）

变迁：

* 空->新建
* 新建->就绪：分配
* 就绪->运行：调度
* 就绪->退出：被迫
* 运行->就绪：时间片到，抢占（系统执行基于优先级的抢占式调度），主动释放
* 运行->阻塞：等待某个时间（IO，资源分配，进程同步，系统服务，远程通讯）
* 运行->退出：正常，异常，被迫
* 阻塞->就绪：等待事件发生，运行与阻塞事件一一对应
* 阻塞->退出：被迫

实现：就绪队列（FIFO），阻塞队列（事件分队）

问题：无优先级：扫描的方式获得，耗时，频率高效率低；

多优先级，多阻塞（优先级分类，事件分队），优先级/事件多->判空多->，效率低

平衡方法：分组，组内FIFO，组间优先调度

挂起：（IO操作），内存->磁盘

挂起进程：不能立即执行，通过代理

原因：OS空间/交换/审计/用户请求

七状态模型：挂起态（就绪/挂起，阻塞/挂起）

挂起动作：程序（硬盘）<-进程（内存）IO操作

无虚拟内存，不部分加载

唤醒：硬盘->内存

原因：CPU与IO速度差异；内存容量有限->进程有限->进程均阻塞->CPU空闲->利用率低

提升利用率->更多可执行进程->加载进程->内存满->挂起腾出内存->新进程

挂起进程：硬盘中无法立即执行，2个挂起状态，代理显示唤醒，父进程

原因（目的）：释放内存，暂停进程（检查可疑程序调试）

六状态：阻塞/挂起->就绪：唤醒，等待事件发生

七状态：

* 新建->就绪/挂起：准备执行（接纳）
* 就绪->就绪/挂起：调试，审计，修改
* 运行->就绪/挂起：
* 阻塞->阻塞/挂起：
* 阻塞/挂起->阻塞：某些执行优先级调度的系统中，预测到高优先级进程所等待的事件不久将发生，可将其预先唤醒，使其可尽快获得CPU使用权
* 阻塞/挂起->就绪/挂起：等待事件发生了，类似阻塞->就绪，无IO
* 就绪/挂起->就绪：有空闲

进程资源分配图：圆圈（进程），方框（资源），箭头（进程->资源），实线（占用，已分配），虚线（申请，未分配）

OS管理：需要信息，内存中数据结构：IO表，内存表，文件表

资源本身信息，使用分配信息，管理控制信息

进程表：进程位置+映像=进程+执行 映像=代码+数据+PCB+堆栈

PCB：进程身份，上下文，管理控制

进程身份=自身ID+父ID+用户ID 进程权限和执行用户权限一样

进程无法访问很多系统例程有权访问，访问不当破坏，进程管理和执行受到很大影响

PCB保护->唯一仲裁程序

**W3**

进程执行模式：用户模式，系统模式（内核模式，特权模式：高权限，系统例程）

模式识别：PSW中的一个位

进程创建：档案（身份）+资源

ID，PCB，资源（包括空间），PCB->正确队列中，其他

进程切换：保护现场，选择进程，恢复现场

保护：软硬件结合，两步

选择：CPU调度（给哪个进程CPU控制权）

恢复：软硬件结合、两步（和保护对应）

操作系统执行模式->和进程的相关程度

无关，操作系统以大内核模式运行，进程概念仅存在于用户和OS无关；有点相关，把OS例程依附于用户进程中执行，当作用户进程的一部分（模式仍不同）；本身就是进程，系统例程=进程，适合多处理器调度

九状态模型图：UNIX，SVR4

5->7：两个挂起态

7->9：两个裂变：运行态->用户运行+内核运行；就绪->就绪+被抢占（同一个队列）

PS.术语的叫法不同

区别：

* 用户运行->被抢占：被抢占
* 用户运行->内核运行：调用系统例程
* 内核运行->用户运行：返回，调用结束
* 内核运行->内核运行：系统例程可以调用系统例程
* 被抢占和就绪是同一个队列

fork指令：创建子进程->父进程用来创建具有和自己上下文完全相同逻辑副本的子进程（进程ID和fork返回值除外）

fork指令完成后：

* 父进程获得CPU控制权
* 子进程获得CPU控制权
* 其他进程获得CPU控制权

->父子进程重新参与CPU控制权的调度

内存管理：没有虚拟内存，没有部分加载（全部加载）

内存在系统启动后->内存=系统内存+用户内存->内存管理：用户内存

用户进程合理利用内存，在固定内存大小的情况下，尽可能多的进程加载到内存->就绪进程多->CPU利用率高

需求：

* 重定位：

定位：找到逻辑地址对应的物理地址

重：进程在内存中的位置发生了变化（物理地址变了）<-挂起唤醒、压缩整理

实现：重定位寄存器：基地址寄存器，界限寄存器

地址类型：物理地址（绝对地址）->访问设备需要，逻辑地址（相对地址）

逻辑地址->+基地址寄存器中的地址->物理地址->界限寄存器中的地址比较->越界：中断，OS处理；不越界：正常访问内存

进程在内存中是连续存放的->两个寄存器中的地址唯一确定一个内存中的连续区域

* 保护：模块，授权
* 共享：模块，授权
* 逻辑组织：模块
* 物理组织：覆盖技术

针对同一个进程的不同模块

内存的管理技术：区，页，段

1. 固定分区（静态分区）

固定：分区（块）数目和大小：等大小和不等大小

如何放置：等大小->内存存放的位置与其存取速度无关->随机；不等大小->最小适配原则->浪费的空间最小（内部碎片->已分配、不可再利用）；队列方案：每个块一个队列，所有块一个队列（全局一个队列）

缺点：固定，小作业（小进程）浪费大量空间

1. 动态分区

可以给进程分配其所需要大小的内存

空洞->外部碎片->未分配、可再利用<-压缩整理技术

放置（分配）：

最佳适配：找到一个装得下进程的最小块

首次适配：从开始位置找到第一个可以装下的空闲块

临近适配：保存最近一次内存分配的位置，从该位置开始查找，找到第一个可以装得下的空闲块

1. 伙伴系统（动态分区特例）

申请分配：进程申请空间S；空闲块2^U，2^U-1<S<2^U，该空闲块装得下进程而该块的一半装不下，则分配，否则：->裂变，产生伙伴块，2^U=2\*2^U-1，重复

释放回收：进程释放空间K->查看内存K的伙伴，若该块空闲，则合并，形成新的空闲块，重复->伙伴块非空闲，则结束

1. 简单分页

进程分页，内存分帧，页帧等大小

如何去找到进程的页在内存中的哪个帧上？页帧的对应关系：页表：帧号、隐式存储的页号（页表下标）；一个进程一个页表

逻辑地址=页号+页内偏移量；物理地址=帧号+帧内偏移量；页内偏移量=帧内偏移量

1. 分页管理
2. 简单分段
3. 分段管理
4. 段页式管理

**W4**

地址映射

逻辑地址->物理地址

页：

页号+页内偏移量=》帧号+偏移量

页号=》页表项=》帧号

段：

段号+段内偏移量=》段在内存中的首地址+偏移量

段号=》段表项=》段在内存中的首地址

U8：虚拟内存

为什么：

1、提升CPU的利用率

物理内存有限=》扩充物理内存=》加载更多的进程=》就绪进程更多=》CPU利用率更高

2、部分加载

局部性原理=》在短时间内指令的执行具有集簇性的倾向=》短时间内的指令都在一块中=》只需要这一块在内存中，其他可以不在内存中

3、避免系统抖动

局部性原理+预测=》避免

4、地址映射是动态的（重定位）

内存中的位置不影响地址映射

如何使用：和无虚拟内存对比，程序（进程）指令和执行发生了什么变化？

（A）指令在执行时在内存中

（B）不在=》缺页（段）中断=》OS，运行->阻塞，启动换入的IO操作=》IO执行操作、CPU调度执行其他进程=》换入完成=》阻塞->就绪=》再调度运行=》同在内存

页式管理系统：页表项=》增加P（存在位）、M（修改位）

地址映射发生了变化：2条路经

一个进程一个页表，进程可能很大，页表也可能很大，占用大量内存空间=》对页表进行页式管理=》二级页表（页表的页表）

倒排页表（反向页表）和物理内存大小以及帧的大小有关，和进程数量和大小无关=》全局唯一一个页表

页表项：页的信息

页号，进程ID，控制位，链指针

冲突（多个页表映射到同一个帧号）

线性散列，溢出链表

**W5**

内存管理：页式管理、段式管理、段页式管理

段式管理：动态分区=》分配给进程多需要的内存空间

无内部碎片

地址映射：逻辑地址=》物理地址

逻辑地址=段号+偏移量

物理地址=没有划分=内存中块的首地址+偏移量

段号=》内存中的块首地址

段表：P，M

一个进程一个段表

段页式管理：进程分段，段再分页（对段采用页式管理），内存分帧（对内存采用页式管理）

一个进程一个段表，若干个页表（一个段一个页表）

地址映射：

逻辑地址=段号+页号+偏移量

物理地址=帧号+偏移量

段表：没有P和M，原因：段表项种的地址指向该段所对应页表的首地址，页表在内存中不存在去判断是否在内存中和是否需要回写=》不需要P和M

页表：P、M（和页式管理相同）

策略：

读取策略：硬盘=》内存

请求式读取：用时才读，用：访问

预约时读取：一次多个页一起读：前提条件：一起读多个页比单个页分别读的时间要很多。缺点：读进来的页有可能没有用都就被替换出去（浪费）

页在硬盘上要连续存放+进程要有足够的空闲页（进程的开始阶段）

放置策略：内存中具体位置

内存的读取速度和位置的无关系

页式、段页式管理：随机放置

段式管理：动态分区

考虑：不是速度，考虑空洞=》最佳适配，首次适配，临近适配

替换策略：哪个页，内存=》硬盘（选择）

选择标准：替换出去的页在最久的将来最不可能被用到

帧锁定：lock位来指定哪些帧不能被替换

常住集，控制结构，IO缓冲区、OS数据结构=》重要，高频

OPT：未来事件，不可能实现，衡量其他算法的标准

LRU：历史事件，历史预测未来《==》局部性原理==》避免系统抖动

标记时间：时间戳，访问栈=》代价大

FIFO：FCFS，最简单，指针

CLOCK：用一个使用位来模拟时间标记（u==0，u==1）循环扫描找到第一个u==0的页用于替换，对u如何赋值：u=1页被访问（从硬盘加载到内存时=》请求式读取=》要访问）u=0在选择替换页扫描的时候，被扫描的u==1的页，u被赋值为0

MIN：1；MAX：N+1

希望在同等条件下，替换出不需要回写的页=》无IO操作=》用m位来标志是否回写

CLOCK++：um位，可被替换页的顺序（找00》找01，遇到u==1把u赋值0

MIN：1；MAX：3N+1

优化：尽量避免IO操作=》页缓冲=》被替换掉的页还是在内存中（系统）=》自由页表+修改页表

算法比较：没有好坏之分，只有使用场合；简单算法一般效果差，复杂算法效果好；os：简单算法的代价达到、接近复杂算法的效果

驻留集：给进程多少的内存空间

清除策略：如何：内存=》硬盘

**W6**

单处理器调度：

调度类型：长、中、短、IO

IO调度：IO请求和IO设备如何使用的问题？跟进程和资源的使用问题类似。

IO请求=》启动IO服务例程=》进程=》行为主体

IO设备=》资源

距离CPU的远近：

长程：系统外=》系统内，接纳进程进系统，道数。磁盘，磁鼓=》内存（物理、虚拟） （就绪，就绪挂起）

中程：交换，虚拟内存=》物理内存，进程如何加载到物理内存中

短程：物理内存（就绪进程）=》CPU

从就绪进程选择一个进程，分配给其CPU的控制权

短程调度准则：两个角度：系统、用户

用户：

性能相关：响应时间，周转时间，执行的最后的期限，周转时间…

性能无关：预测性

系统：

性能相关：吞吐量，吞吐率，CPU利用率（用户进程）

其他：公平性，优先级（动态、静态），平衡性（资源使用）…

饥饿：就绪进程，无限忽视（无CPU使用）

每一个算法要考虑是否会存在饥饿现象，有如何解决？

动态优先级=》奖励等待时间，进程等待一定时间，该进程优先级提升一级=》能够解决基于优先级调度的饥饿问题=》？

系统中，任意进程的执行时间是有限的

优先级的级数是有限的，也就是系统中，优先级的排队队列数量是有限的

每一个优先级排队队列中排队的进程数量是有限的

一个低优先级的进程=》等待一定时间（有限时间）能够提升一个优先级=》所有优先级是有限的=》在有限时间内，该进程可以提升到最高优先级队列中排队=》排在该进程前面的进程数量是有限的=》在有限时间内，这些进程可以执行完成并调度到该进程=》有限时间内，该进程获得了CPU的使用权、控制权=》无饥饿

选择函数：从就绪队列中选择一个进程执行

W（等待时间），E（花费的执行时间），S（总服务时间，包括E）

决策模式：放弃CPU是因为进程本身的原因（内因），还是因为外部原因（外因）

非抢占模式：内因

抢占模式：外因，时间片轮转

调度算法：

先来先服务FCFS：FIFO，简单算法，无饥饿

轮转RR：基于时间片轮转，时间片长短=》略大于一个典型的交互=》要让一个典型的交互在一个时间片内完成

每一个进程每一次排队，可以用到一个时间片

HeavyIO的进程，经常使用不到一个时间片就会发生IO操作，等待IO，变阻塞=》每次排队，没用完一个时间片=》对HeavyIO是不公平的

改进或补偿：没有用完的时间片补还给进程=》

VRR：虚拟RR

辅助队列：就绪队列，比原来的就绪队列优先级高

阻塞状态=》就绪状态（不进入原来的就绪队列），进出入辅助队列=》优先调度=》补偿未用完的时间

最短进程优先SPN：短进程优先，执行时间最短的先执行

执行时间=》未来事件=》用历史来预测、估算

简单平均，指数平滑

最短剩余时间SRT：剩余时间最短，SPN的抢占式版本

剩余时间=执行时间-已执行时间（已知）

最高相应比优先HRRN：最高相应比优先：相应比：R=（W+S）/S=1+W/S（W等待处理器时间，S期待服务时间）

反馈算法：=》惩罚执行时间，每经过一次调度（一个时间片，不一定）优先级降低，最后一个优先级用RR=》饥饿问题，长进程的周转时间大幅增加

不等时间片策略，2^i=》动态优先级

基于组的调度策略=》基于用户的调度

组间的使用调度策略，组内的策略可以独立=》组内的各个进程会互相影响

基于彩票的调度

准确度量优先级，容易实现进程的协作与共享

**W7**

IO设备和磁盘调度：

IO设备的类型：

1.人可读：人机交互

2.机器可读：

3.通信

IO设备的特性：=》差异性

1.速度

2.传送的方式：流，块

IO技术的发展=》CPU越来越少参与IO的操作

直接访问

可编程IO

中断IO

DMA

IO通道（+CPU）

IO处理器（+CPU+指令集=》计算机）

DMA：

构成：CPU（六个三对寄存器）

数据寄存器+地址寄存器=》+控制=》DMA数据传送

集成方式：

1.单总线：分离式，集成式

2.IO总线：专用

进程执行和IO操作的顺序关系：进程需要等待IO操作完成才能继续执行

进程：运行态=》阻塞态=》内存管理=》阻塞挂起态

IO操作=》数据交换（传送）=》源地址和目的地址=》IO设备和物理内存（属于进程的）

死锁：两个或者两个以上的进程，由于互相等待而无法继续执行的现象

单进程死锁：

1.进程发出IO指令（请求）后被挂起=》该进程没有物理内存

2.OS收到IO请求后，启动一个IO进程用来服务该IO请求

3.IO进程无法运行（阻塞态），等物理内存《=该进程的唤醒（换入）（虚拟内存=》物理内存）

4.该进程无法运行（阻塞挂起），等IO操作完成

=》该进程和由该进程IO请求启动的IO进程互相等待=》死锁=》单进程死锁（和死锁定义不矛盾）

根本原因：没有物理内存

缓冲区=》物理内存快=》系统区域=》和用户进程无关

作用：平滑IO需求峰值

用时间换取速度=》笨鸟先飞

单缓冲区

双缓冲区

循环缓冲区

=》容量，并发度

磁盘调度

磁盘：

磁：使用NS不同指向来存储0和1

盘：环状的磁道来划分存储区域

磁道上还可以进行扇区的划分

=》磁道和磁道间，扇区和扇区间：间隔

=》不同的磁道，面积可能不同，但存储量一样

=》不同扇区，面积可能不同，但存储量一样

=》扇片：单面存储、双面存储

=》磁盘：包含多片

磁头和磁力臂=》径向运动=》磁道定位=》寻道

旋转轴=》带动磁盘旋转=》扇区定位=》随机性，单向旋转=》旋转扇区定位的平均时间=转半圈的时间=》旋转延迟=1/2r

数据的传送时间=》真正的数据读写操作=》数据存储区域经过（旋转经过）磁头的时间

T=寻道时间+旋转延迟+传送时间=》存取时间

磁盘读写=》IO操作=》占用（拥有）IO通道=》其他的进程无法使用该通道（独占性、排他性）=》当进程占用该通道的时候，IO进程还没有找到数据所在区域=》为了提高Io设备的利用率=》RPS：旋转定位感知

1.占有通道

2.在寻道的时候，先释放通道（给别的进程使用），在数据存储区域接近（完成寻道，在做旋转延迟）磁头时，尝试连接通道

3.成功：数据传送，失败：在转一圈=》饥饿

例子=》文件或数据在辅存中的存储方式对整个的性能影响很大（指数级别）

预约式请求读取策略的时候，有一个前提条件：进程的页在辅存中要连续存放，为什么？

磁盘调度

地盘性能=寻道时间+旋转延迟+传送时间（数据读写时间、固定）=》寻道调度=》提高磁盘性能=》降低寻道时间

寻道时间=距离/速度=磁道间隔数量\*磁道间隔/速度=》寻道时间 正比 磁道间隔数量

=》寻道长度、平均寻道长度

寻道=》一个磁盘、多个IO请求

多个磁盘=》RAID

具体算法：

先进先出FIFO：队列

优先级PRI：非抢占式

后进先出LIFO：堆栈

最短服务时间优先SSTF：最短服务时间优先，服务时间=磁头从当前位置移动到IO请求（数据）所在磁道的时间

IO服务不停进行，磁头的位置不停发生变化，服务时间=》动态

SCAN：边移动，边服务=》双向服务（从里到外+从外到里）=》折返（返回）时机：磁盘边界（最里和最外的磁道），当前时刻所有IO请求的最大最小边界，

C-SCAN：同SCAN，除了：单向服务

问题：上面的几种算法，哪些会产生饥饿？：非FIFO

如何解决SCAN系列算法的饥饿现象？

=》分组

1、定队列长度：每个队列N个请求，队列数量不定

2、定队列数量：只有两个队列，=》队列中请求数量不定

在系统开始进行队列中IO请求服务的时候，没有新的IO请求进入该队列=》该队列中的IO请求数量是有限的，每个IO请求的服务时间是有限的=》对该队列而言，无饥饿现象

队列间，使用FIFO算法=》对队列而言，无饥饿现象

定长：N-STEP-SCAN

定量：FSCAN

如何提升磁盘的性能：

1、RAID：独立磁盘冗余阵列

独立=》物理磁盘

冗余=》空间换时间=》数据冗余、校验冗余

阵列=》多个磁盘=》并行

条带化

7 Levels

RAID 0：无冗余，100%，可靠性无提升

1-6：可靠性提升

1：镜像，50%，数据冗余

2：海明码

3-6：奇偶校验

3：基于位的奇偶校验

4：基于块的奇偶校验

5：分布式奇偶校验（基于块）=》最常用

6：双奇偶校验（基于块，分布式）

Cache Buffering

Cache关注命中率，置换算法，局部性原理

最近最少使用LRU：访问栈，栈底呗替换

最不常用LFU：访问计数器，每次访问+1

初始化=1（新加载）

=》问题：短时高频访问=》计数器暴增、很大=》后面如果没用，很难被替换=》基于频率的LFU=》区=新区+旧区

新区：访问计数器没有变化

旧区：访问计数器+1，只有在该区的块会被替换

=》问题：短时高频在进入旧区后很容易被替换（计数器很小）

=》区=新区+中间区域+旧区

中间区域=》缓冲=》不会被替换

**W8**

文件管理：两个图：功能结构层次图+读写过程图

=》用户如何使用磁盘上的文件（读写）

层次结构图：

用户程序

访问层：访问方法，文件逻辑组织形式：堆，顺序文件，索引顺序文件，索引文件，散列（哈希）文件

逻辑IO：记录，内容=》不关注存储：位置

监控：选择、调度、寻优、缓冲区

物理IO：块=》文件的存储形式=》关注的是位置（Cache、磁盘）=》不关注内容

驱动层：具体IO操作管理，具体实现由厂商提供=》细节

硬件（磁盘，磁鼓等）

如何读？

1、发送IO指令（操作类型，操作对象）=》读、文件名

2、权限判断（ACL）

EXIT：无权限

3、映射管理（找到操作对象具体的物理位置）

4、读数据：从Cache/Buffer中读，从磁盘=》Cache/Buffer=》再读=》块

5、解块：块=》记录=》使用文件结构（逻辑组织）进行组合

EXIT：读错误

6、返回用户

如何写？

1、发送IO指令（操作类型，操作对象，操作内容）=》写、文件名、数据

2、权限判断（ACL）

EXIT：无权限

3、映射管理（找到操作对象具体的物理位置）

4、组块：记录=》块

5、写数据：缓写：Cache/Buffering，直接写：Disk

EXIT：写失败

不同类型：修改写（编辑写）=》找到原来的块，新增写=》分配空闲块

EXIT：无空闲块/空闲空间不足

6、返回用户

用户程序=》文件结构=》记录=》块=》磁盘=》空闲块

文件逻辑结构：（5种）

1、堆：按照到达的时间顺序进行组织

自描述的，无固定结构的

访问：顺序访问、穷举搜索

=》数据搜集

2、顺序文件：KEY，按照KEY的顺序排列

每次新增都需要进行排序=》效率低（硬盘）

日志、事务文件=》批量更新

溢出文件

访问方式：顺序访问

查找：顺序查找 vs. 堆的穷举搜索

中间终止

3、索引顺序文件

对顺序文件构建索引=》分块查找=》主文件（顺序文件）分块，每个块的首位值提取出来，放到一个索引文件中=》文件索引（顺序文件=》多级索引

4、索引文件

关键字域，构建索引（全索引、部分索引）=》所有记录都必须有至少一个索引指向=》必须通过索引才能访问记录

5、散列、Hash文件

线性散列、溢出链表

直接访问的地址（物理地址=》硬盘位置信息）

B树 vs 索引顺序文件

度为d的B树【半满，全满】（除根节点外）

小于半满=》合并

大于全满=》分裂（分解）

合并、分解=》递归迭代过程

**W9**

文件上完

**W10**

并发性：同步和互斥

概念：

（1）临界区：代码段、执行具有互斥性（排他性）

（2）互斥：任何时刻，最多只有一个进程正在执行。当没有其他进程正在执行的时候，一个想进入的进程可以立即进入（公平性），如果有其它进程正在进行，想进入的进程需要排队等候

（3）死锁：一组进程（2个或两个以上）、互相等待（组内互相等待）、无法继续执行（整组）=》单进程死锁

（4）活锁：某些进程，为了组内其他进程能继续执行而做出的牺牲（不做无用功=不执行进程指令）

（5）竞争条件：读写过程、赢得先操作、失败者决定了最后的值

（6）饥饿“无限期忽视（CPU）、无法执行=》IO请求

哪里有并发性：进程间、进程内（多线程）、OS服务历程

引发问题：结果不可预测、和预期不一致

进程间三种关系：

（1）互相不知道：资源竞争

（2）间接之道：资源共享

（3）直接知道：直接通讯

如何解决：增加规则

针对访问变量=》一次只有一个进程执行

=》互斥支持=》实现临界区（互斥执行）

=》单、多 同一个规则

对互持支持的条件：

（1）进得去：没有正在执行的进程，一个想进入的进程可以立即进入

（2）进不去：如果有正在执行的进程，任何想进入的进程都无法进入，需要等待

（3）出的来：有限时间（进程属性）=》关注进程离开时对于条件的变化或设置

（4）再进去：正在执行的进程离开了，正在等待的进程可以进入（某一个）

六种对互斥支持的方法：

（1）中断禁用：代价大，单、多需要不同考虑（唯一）

（2）专有机器指令：比较与交换

函数，三个参数（一个形参，两个值参)

（3）专有机器指令：交换

过程，两个形参

（2）+（3）=》原子过程：一旦开始执行，不会被其他进程中断=》效率低、忙等待=》实现信号量

（4）信号量

Struct semaphore

{

Int count；//信号计数器，用于存储信号，具备无限的信号存储能力；大于0：表示可以执行semWait而不需要等待进程的数量；小于等于0：其绝对值表示正在排队的进程的数量

queueType queue；//排队队列（阻塞队列）

}

两个操作原语：senWait，semSignal=》原子过程

先加减后判断

Void semWait（semaphore s）

{

s.count--；

if（s.count<0）//是否有可用的信号

{1.把自己放在该信号排队队列中-阻塞；2.改变状态为阻塞状态}

}

Void semSignal（semaphore s）

{

s.count++；

if（s.count>0）//是否有排队的进程

{1.把队首进程（强信号量）移出排队队列；2.把该进程放到就绪队列中}

}

其他资料：PV操作：P（s）=semWait（s），V（s）=semSignal（s）

semWait（s）；

临界区；

semSignal（s）；

无限缓冲区的生产者与消费者问题：规则=先生产后消费

二元信号量，信号量的一种特例

Struct binary\_semaphore

{

Enum {0,1} value;//信号量的值=》最多存储一个信号量，多余信号会丢失=》semSignalB

queueType queue；//排队队列

}

Void semWaitB（binary\_semaphore bs）

{

if（s.value==1）s.value=0；

Else 1、放到队列中 2、更改状态

}

Void semSignalB（binary\_semaphore bs）

{

if（s.queue isEmpty（））s.value=1；

Else 1、唤醒队首进程（移出） 2、放入就绪队列

}

（5）管程

（6）消息

**W11**

生产者与消费者问题（无限，有限）

并发性应用问题思考：

1、行为主体：生产者、消费者

2、临界资源：缓冲区；=》一种临界资源用一个信号量

3、同步规则：（1）先生产后消费（2）有空位才能放

4、行为过程：

（1）生产者

生产产品；

放置产品；=》临界资源、规则（2）

（2）消费者

获得产品；=》临界资源、规则（1）

消费产品

无限缓冲区的生产者和消费者问题：

二元信号量的解决方案=》注意全局变量的使用（控制）

信号量的解决方案：

semaphore s=1；//临界资源（缓冲区）的使用权限

semaphore n=0；//可用产品的数量=》规则（1）

void producer（）

{

while（true）

{

生产产品；

semWait（s）；//获得缓冲区的使用权

放置产品

semSignal（s）；//释放缓冲区的使用权

semSignal（n）；//通知消费者多了一个可用的产品=》先生产后消费

}

}

void producer（）

{

while（true）

{

semWait（n）；//缓冲区中是否有可用的产品=》先生产后消费

semWait（s）；//获得缓冲区的使用权

获得产品；

semSignal（s）；//释放缓冲区的使用权

消费产品；

}

}

void main（）

{

parbegin（producer，consmer）；

}

有限缓冲区的生产者和消费者问题：

信号量的解决方案：

const int sizeOfBuffer=10；

semaphore s=1；//临界资源（缓冲区）的使用权限

semaphore n=0；//可用产品的数量=》规则（1）

semaphore e=sizeOfBuffer；//缓冲区中可用空闲数量=》规则（2）

void producer（）

{

while（true）

{

生产产品；

semWait（e）；//缓冲区中是否有可用的空闲空间

semWait（s）；//获得缓冲区的使用权

放置产品

semSignal（s）；//释放缓冲区的使用权

semSignal（n）；//通知消费者多了一个可用的产品=》先生产后消费

}

}

void producer（）

{

while（true）

{

semWait（n）；//缓冲区中是否有可用的产品=》先生产后消费

semWait（s）；//获得缓冲区的使用权

获得产品；

semSignal（s）；//释放缓冲区的使用权

semSignal（e）；//通知生产者多了一个可用的空闲空间

消费产品；

}

}

void main（）

{

parbegin（producer，consmer）；

}

简单的水果问题：

=》缓冲区数量为1的生产者与消费者问题

=》生产者=父亲，消费者=女儿、儿子=》行为一致=》合并为孩子的行为

管程：

并发性问题=》同步+互斥

管程机制=》任何时候，最多只有一个进程在莞城中运行=》解决互斥问题=》程序员：解决同步

整体：

组成：

（1）整体，monitor name；=》定义管程

（2）对外的通讯=》出口、入口=》入口：排队队列（等待管程变成可用）

（3）局部数据：临界资源、正常的业务逻辑

（4）初始化代码=》初始化局部数据

（5）使用数据的过程=》管程外的进程通过调用这些过程进入管程

（6）同步=》条件变量：特殊变量，无值，不用初始化，只用于排队

（7）因cwait而排队的队列（多个，每个条件变量一个）

（8）因csignal而排队的队列（1个，紧急队列，优先级较高）

被唤醒的进程需要立即执行=》唤醒条件会发生变化（如果别的进程执行）

调度的，不能让在入口排队的进程抢了变为可用的管程=》调度要高度可靠

正在执行csignal的进程要让出管程（如果该进程的csignal语句不是最后一个语句）

如何使用管程来解决有限缓冲区的生产者和消费者问题？

---

monitor name；=》定义管程

。。。

临界资源、局部数据定义

条件变量定义

void append（char x）

{

//规则：同空位才能放

if（count==N） cwait（notfull）；=》semWait（e）；

。。。

csignal（notempty）；=》semSignal（n）；

}

Char take（）

{

//先生产后消费

if（count==0） cwait（notempty）；=》semWait（n）；

。。。

csignal（notfull）；

}

使用信号的管程的问题

（1）调度必须高度可靠

（2）执行csignal进程需要阻塞=》就绪=》调度才能继续执行=》效率不高

=》根本原因：怕唤醒条件被改变

=》使用消息和通知的管程

=》特点：被唤醒的进程进入管程进行执行时要重新检查被唤醒的条件。符合（满足）=》继续执行；不满足=》继续等待

解决方法的不同：

（1）if=》while

（2）csignal=》cnotify

**W12**

消息机制

消息=header+body=信号+内存（数据）

空消息null=》body为空的消息=》信号=》无内容

两个原语：

send（destination，message）；

receive（source，message）；

执行方式=》阻塞、非阻塞

组合方式又4种，默认：send不阻塞，receive阻塞

消息通讯方式=》直接+间接（1：1，1：M，M：1，M：N）

消息=》互斥

1、定义与初始化

create\_mailbox（mutex）；

send（mutex，null）；

2、临界区

message msg；

receive（mutex，msg）；

临界区

send（mutex，null）；

消息实现有限缓冲区的生产者与消费者问题：

（1）消息具有携带/存储数据的能力=》无缓冲区

（2）一个信号量=》一个信箱

Mailbox mayproduce，mayconsume；=》e，n

create\_mailbox（mayproduce）；

create\_mailbox（mayconsume）；

for(int i=0；i++；i<N）send（mayproduce，null）；

生产者：

//有空位才能放（生产）

receive（mayproduce，msg）；

msg=生产；

send（mayconsume，msg）；

消费者：

//先生产后消费

receive（mayconsume，msg）；

send（mayproduce，null）；

消费（msg）；

读者和写者：

读读=》并行=》跟读=》成组思维=》（1）一个计数器和一个用于控制计数器访问的信号量（2）组的思想：第一个进程负责获取权限，最后一个进程负责释放权限

读写、写写=》互斥

生产者=读者+写者（综合）

消费者=读者+写者（综合）

读进程优先的读者与写者问题

写进程优先的读者与写者问题

优先=》读进程优先=》优先跟读（即使有写进程在等待）

写进程优先=》阻止跟读（有写进程在等待），正在读的进程让其完成读=》排队信号量

读进程优先：

int readcount=0；

semaphore x=1，wsem=1；

void reader（）

{

while （true）

{

semWait（x）；//获得权限

readcount+；

if（readcount==1）semWait（wsem）；//组中第一个进程负责获取缓冲区使用权限

semSignal（x）；//释放权限

readUnit（）；

semWait（x）；

readcount--；

if（readcount==0）semSignal（wsem）；//组中最后一个进程负责释放缓冲区的使用权

semSignal（x）；//释放权限

}

}

void writer（）

{

while （true）

{

semWait（wsem）；//获得权限

writeUnit（）；

semSignal（wsem）；//释放权限

}

}

写进程优先：

int readcount=0，writecount=0；

semaphore x=1，y=1，rsem=1，wsem=1；

semaphore z=1；

void reader（）

{

while （true）

{

semWait（z）；//读进程排队信号量

semWait（rsem）；//在rsem上排队的读进程最多只有一个

semWait（x）；//获得权限

readcount+；

if（readcount==1）semWait（wsem）；//组中第一个进程负责获取缓冲区使用权限

semSignal（x）；//释放权限

semSignal（rsem）；

semSignal（z）；

readUnit（）；

semWait（x）；

readcount--；

if（readcount==0）semSignal（wsem）；//组中最后一个进程负责释放缓冲区的使用权

semSignal（x）；//释放权限

}

}

void writer（）

{

while （true）

{

semWait（y）；//获得权限

writecount+；

if（writecount==1）semWait（rsem）；//第一个写进程负责获取读权限=》写者组和读者在rsem上竞争

semSignal（y）；//释放权限

semWait（wsem）；//获得权限

writeUnit（）；

semSignal（wsem）；//释放权限

semWait（y）；

writecount--；

if（writecount==0）semSignal（rsem）；//最后一个写进程负责释放读权限

}

}

进程趋势图

Void s1()

{

…

semSignal(s1);//通知其他进程s1执行结束

semSignal(s1);//通知其他进程s1执行结束

}

Void s3()

{

semWait(s1);

…true

semSignal(s2);//通知其他进程s2执行结束

semSignal(s2);//通知其他进程21执行结束

W13

利用消息机制来解决写者优先的读者-写着问题：

（1）行为主体=》读者（M个）、写者（N个）、控制器（1个）

（2）消息=》直接通信（进程ID）、间接通信（信箱）

（3）信箱：readrequest，writerequest，finished=》读者写者与控制器的交互

（4）控制器：count变量=》初始化值要大于系统中最大读进程的并发数=》不同取值范围来表征读者写者的不同状态

读者：

（1）发送读请求=》readrequest信箱

（2）等待请求批准=》直接通信=》获得批准

（3）执行具体的读操作

（4）发送完成的信息=》finished信箱（和写者共用）

写者：

（1）发送写请求=》writerequest信箱

（2）等待写请求批准=》直接通信=》获得批准

（3）执行具体的写操作

（4）发送完成的信息=》finished信箱（和读者共用）

控制器：

void controller（）

{

while（true）

{

if（count>0）//只有0-N个读者，没有写者

{

（1）处理完成信息（finished信箱=》来自于读进程）

（2）处理写请求信息（writerequest信箱=》先处理是写者优先的体现=》不立即批准，需要记录消息来源进程ID（批准需要直接通信））

（3）count=count-1

（4）处理读请求信息（readrequest信箱）

}

if（count==0）//没有读进程，只有一个写进程已请求写

{

（1）批准写请求

（2）等写结束（写进程和任何其他都写进程都是互斥）

（3）count=count+初始化值

}

while（count<0）//一个写进程请求，1-N个读进程正在读

{

等待正在读的读进程结束

每个读进程结束count=count+1

}

}

}

死锁和饥饿：

资源分类=可重用资源+可消费资源

资源分配图=》系统当前时刻资源的分配与申请的情况

（1）进程：圆圈

（2）资源：方框；资源实例：方框中的黑点

（3）申请：进程申请资源分配=》从圆圈指向方框的有向线段

（4）占有：资源分配给进程=》从黑点到圆圈的有向线段

死锁的四个充分必要（充要条件）：

（1）互斥：资源本身的属性=》临界资源

（2）占有且等待：申请资源分配的时候已占有一定资源（已分配）

（3）非抢占：（A）不抢其他进程已分配的资源（B）主动释放自己的已分配资源

（4）循环等待：组内进程、互相等待

如何去解决死锁问题？

三类方法：

（1）死锁预防：破坏四个充要条件之一

（2）死锁避免：预分配检测

（3）死锁检测：不管分配，周期性检测=》还原点

有死锁=》死锁恢复（活锁）

没有死锁=》继续执行

死锁预防=》4个条件

（1）互斥=》一般不改变

（2）占有且等待

（A）占有不等待

（B）等待不占有

（C）不等待不占有=》不参与资源竞争 X

方案：整体申请与分配（一次性申请与分配）

分配=》占有不等待

不分配=》等待不占有

（3）非抢占

（A）抢占其他进程的已分配资源

（B）主动释放自己已分配的资源

（4）循环等待

方案：资源的线性化=》给资源编号=》进程所申请的资源编号要大于已分配的资源编号

如何解决死锁问题？

三类：

1、死锁预防：破坏2-4条件

2、死锁避免

1-3条件不做限定，动态选择来避免死锁

（1）进程启动拒绝

死锁=》多个的进程参与了资源竞争

启动拒绝=》避免/消除资源竞争

对于任意的资源=》系统中所有的进程并发（物理并行）地运行，资源也足够用

足够=》资源的数量大于等于所有进程对该类资源所需的总量=》进程间无需竞争

启动一个进程，需要假定该进程启动后满足以上条件

（2）资源分配拒绝

系统状态=》两向量，两矩阵

R：系统总资源向量

V：系统可用资源向量

C：需求矩阵

A：已分配矩阵

=》C-A未分配矩阵

系统状态=安全状态+不安全状态

能否去找到一个进程执行的序列=》所有的进程都能执行完成

进程执行完成=》满足该进程所有未分配资源的需求

如何去找到这样一个执行序列？（如何判断系统的安全性）

两个的进程列表=未执行完进程列表A+已执行完进程列表B

找一个可以执行完的进程=》集全系统可用资源给该进程，如果满足该进程未分配资源要求（满足：各个分量都大于等于），则该进程可以执行完成=》 A=》B 释放该进程的所有已分配的资源（加入到系统可用资源中）

A：空=》安全状态

A：不空=》不安全状态

银行家算法=》每次分配、次次安全=》预分配后形成新状态的安全性

算法步骤：

（1）合法性判断：所申请的+已分配的是否大于所声明的（任何一个分量都不能大于）

有=》ERROR

（2）可用性判断：所申请的是否小于可使用的

否=》suspend（非ERROR）

（3）预分配=》新状态（判断安全性？）

（4）安全性判断

安全=》分配（执行预分配）

不安全=》不分配（回收预分配）=》suspend

3、死锁检测

系统不干预资源的申请与分配

=》公平性原则：如果系统中有可用的资源，任何进程申请资源，要立即分配该资源

周期性检测=》有死锁、没有死锁

没有死锁=》继续指定、记录当前状态（还原点）

有死锁=》恢复死锁（活锁）

（1）所有进程回滚到0：杀死所有的进程

（2）所有进程回滚到还原点：所有进程回到上一个没有死锁的状态

=》再次死锁

（3）逐个回滚到原点0：逐个杀死死锁进程，直到没有死锁

（4）逐个抢占其他进程已分配的资源，知道没有死锁

（3）（4）如何选择=》选择标准或原则

个性化定义=》根据应用场和选择合适的标准

cpu、优先级、产出、剩余时间。。。

=》综合方案

资源分组、组间线性化、组内个性化

哲学家问题：

（1）增加可用资源数量

（2）减少参与资源竞争的进程数量

（3）指定更合理的资源竞争/使用规则=》（2）

信号量解决方案=》死锁=》room排队信号量，限定同时进入房间的哲学家数量

管程解决方案=》信号量解决方案（同逻辑）

管程仅一个进程进入

第四章：线程

进程：

（1）资源所有者（资源分配对象）X=》线程

（2）执行调度单元=》线程=》动态、运行特性

线程=》TCB、堆栈（用户堆栈，系统堆栈）

进程=》PCB、用户空间（资源）、Threads（TCB、用户堆栈，系统堆栈）

TCB<PCB，进程作为线程的容器

=》线程的创建、终止、切换比进程快

=》线程的通信更有效：无内核参与

=》进程的终止、挂起：所有的进程内线程都终止和挂起

线程=用户级线程+内核级线程

（1）用户级线程

线程的概念仅在用户空间中，OS（内核）不知道线程的存在

=》线程的调度是由线程库来实现的，线程库是应用程序相关的

=》OS是基于进程进行调度的

=》进程和线程的状态不一致：线程库没有CPU时间来实现线程的状态改变

=》线程在进行系统调用、IO请求、线程同步、线程调度、进程调度等情况下，线程和进程的状态变化

缺点：

=》一个线程的阻塞会引起所有线程的阻塞（同一个进程内）

=》无法用到多处理器技术

解决方案：

=》无阻碍的系统调用

=》多进程

（2）内核级线程

线程在用户空间和内核空间中都存在，1：1

=》基于线程调度（OS）

=》解决：阻塞问题、多处理器问题

用户级线程、内核级线程、进程

=》巨大的速度差异：数量级的

（3）综合方案=用户级线程+内核级线程

用户级线程进行分组（映射）=》内核级线程上

阻塞问题、多处理器问题=》部分解决+速度

=》线程库基于线程的调度、OS基于线程的调度

多核、并行

类似于开口向下的抛物线

=》调度、并行、同步、数据一致性。。。=》代价

微内核结构

核心功能=》内核

其他功能=》进程

分析：

缸=》缓冲区（s，n，e）

gang，gang\_n，gang\_e

信号量定义：

Semaphore jing=1；//井的互斥访问

Semaphore tong=3；//可用桶的数量，初始状态下有3个可用的桶

Semaphore gang=1；//缸的互斥访问，初始状态下缸是可访问的

Gang\_n=0;//缸中可用水的数量，初始状态下，缸中无水

Gang\_e=10；//缸中可再装水的数量（空闲空间数量），初始状态下，可再装10桶水（缸中无水）

Void old（）

{

while（true）

{

口渴；

semWait（gang\_n)；//港中是否有可以喝的水

semWait（tong）；//