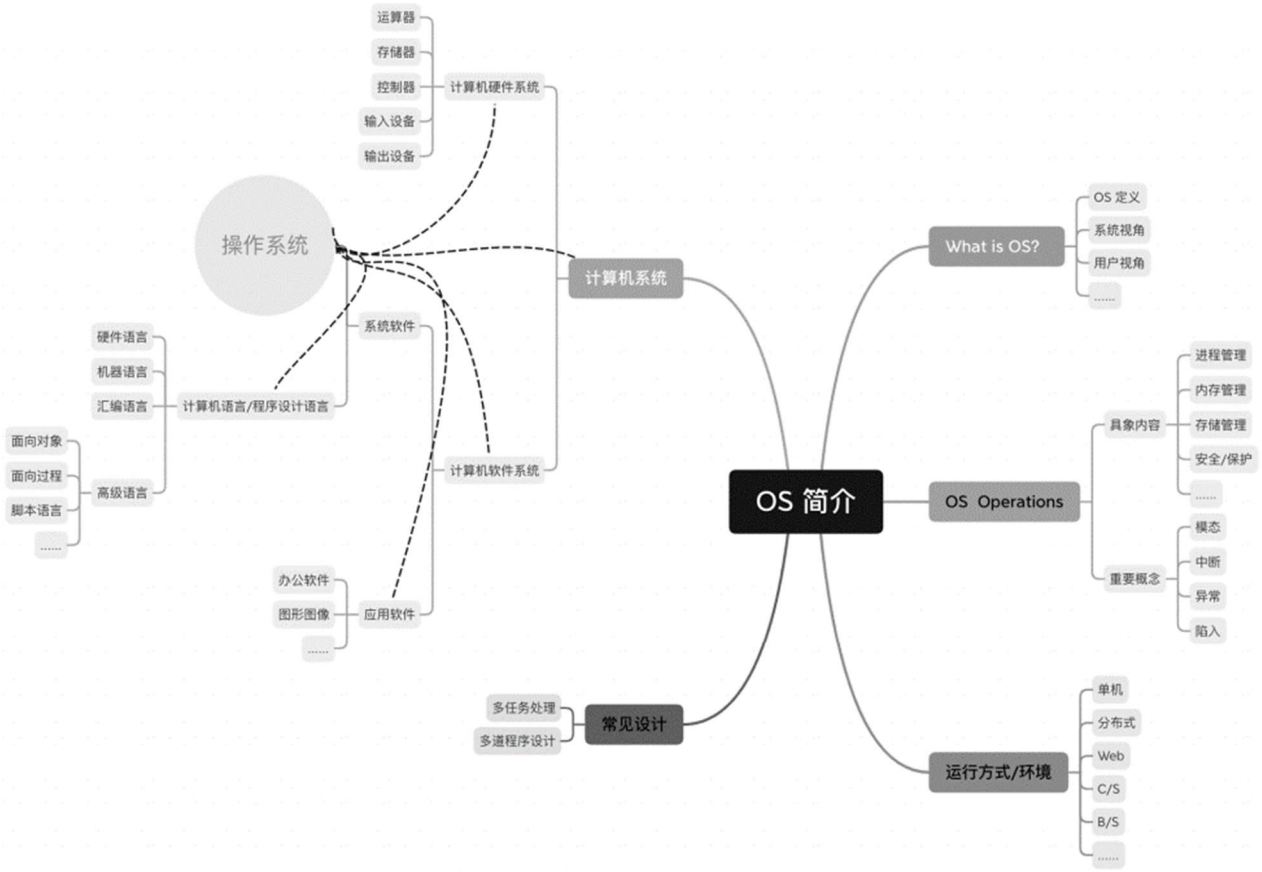


1 操作系统介绍

1.操作系统概念: a piece of software that manages a computer, making computer's hardware resources accessible to software through a consistent set of interfaces.

从硬件抽象出来，提供一系列接口，在整个生态上创造

2.批处理: 用户脱机，成批处理，任务周转时间长

-单道批处理 one application(MS-DOS): 自动、顺序、单道，资源利用率低，系统吞吐量小

-多道程序 multi-programming: 同时运行多个应用，互相隔离；防止程序 crash OS 和其他部分；地址转译，多模态；程序不被允许读写其他程序或系统的内存；多道、无序、调度；考虑平均周转时间，交互能力等

3.分时系统 time-sharing: CPU 同时处理多个在内存和硬盘上的任务，任务调入调出内存，用户和系统有在线交互，多路、独立、及时、交互

4.实时系统 real-time: 多道、独立、交互、及时、可靠，作为控制器；硬实时(次级存储有限或空缺，数据被存在短期内存或 ROM；不能用分时系统，不被通用操作系统支持)，软实时(工业控制中有限使用，可被分时系统兼容，在需要及时反应的 app 中较有用，如 VR,多媒体)

5.手持 handheld: personal digital assistant，手机，有限内存，低速处理器，小显示器

6.网络操作系统: Centralized，Client/Server，Peer2Peer；网络通信能力，网络服务；资源共享，独立自主

7.并行: 提高同一时间间隔内的操作数量，时间、空间、数据、任务并行

8.分布式: 网络连接在一起，各自局部存储器和外部设备；与网络的区别在软件而非硬件；独立，无主从关系，协作，数据/任务分布位置，健壮性；多机进程通信，分布资源共享，并行分布计算，分布式网络管理

9.嵌入式: 嵌入在各种设备、装置或系统中的软硬件系统，微型、可定制、实时、可靠、易移植

10.功能: CPU、存储、文件、设备、用户接口、网络与通信管理

-CPU: 进程/线程管理，同步互斥，进程通信死锁，处理器、作业、进程调度

-存储: 分配、共享、保护、地址转换、存储扩充

-文件: 目录、存取控制/保护、逻辑/物理组织、空间

-设备: 设备分配、设备驱动、缓冲管理

-用户接口: 命令、程序、图形接口

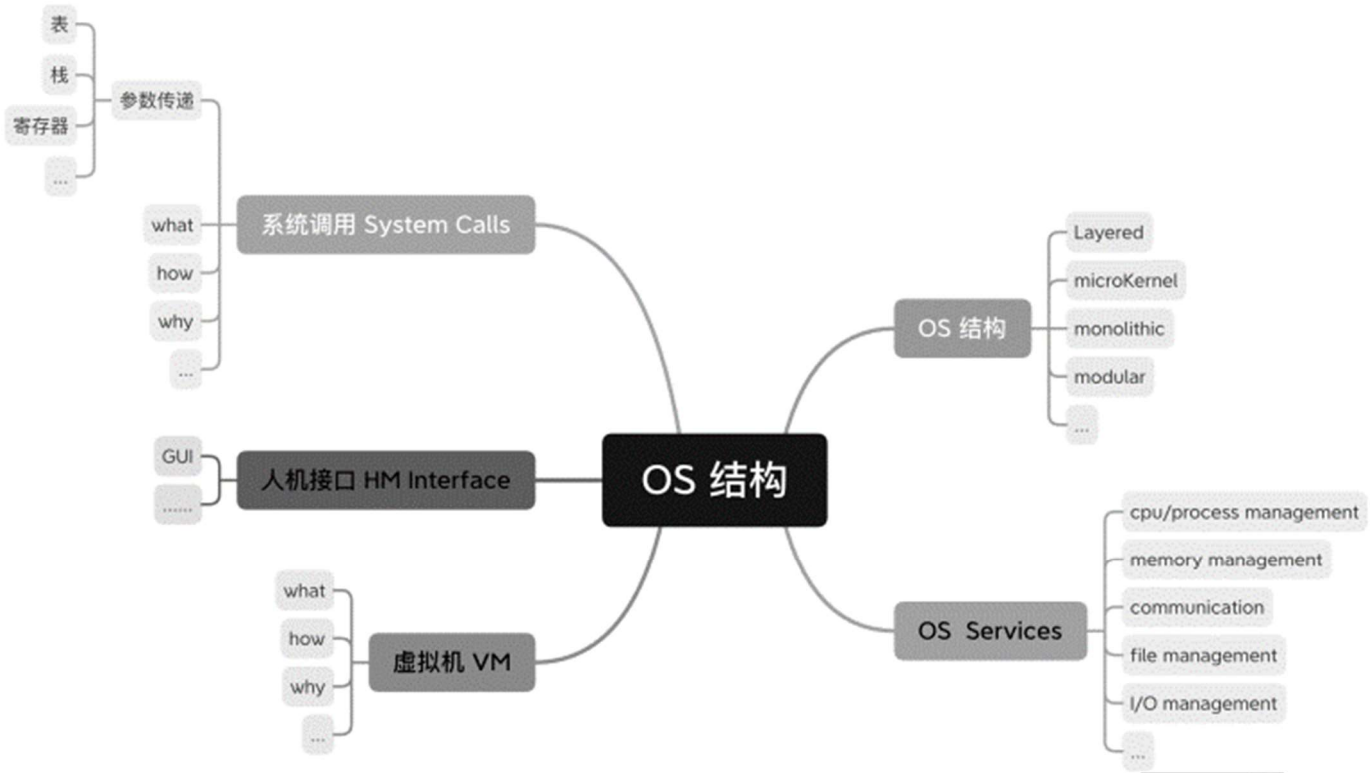
11.特征: **并发** Concurrence，切换互动，隔离，协作，协调竞争，保护资源，保证一致性；**共享** Sharing，互斥共享，同时访问；**虚拟** Virtual **异步** Asynchronism

12.性能指标: 可靠性、吞吐率、相应时间、资源利用率、可移植性

13. 问题: 冲突 conflict，协作 coordination，一致 coherence，数据存取控制 access control

14.用户行为难以预料，存在破坏性可能，硬件问题难以预见，健壮性、容错性

15.局部性



2 操作系统结构

1.抽象: CMOS→逻辑门→机器语言→汇编→程序语言

2.系统调用 system call: 提供程序和操作系统间的接口

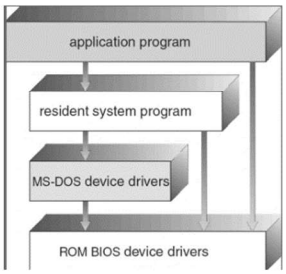
寄存器传参，表地址传参，栈传参

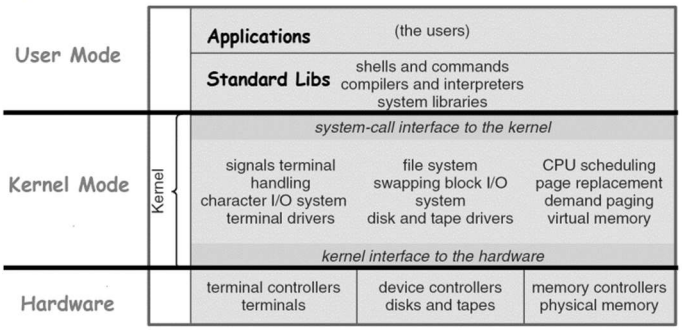
进程控制，文件管理，设备管理，信息保护，通信

3.简单结构

-MS-DOS 在最小的空间提供最多的功能，没有按模块拆分，接口和功能级别也没有被分开

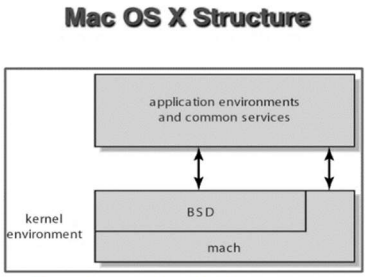
-UNIX 由硬件功能限制，内核(在系统调用接口下层和硬件上层的一切，文件系统、CPU 调度、内存管理和其他系统功能)+系统程序

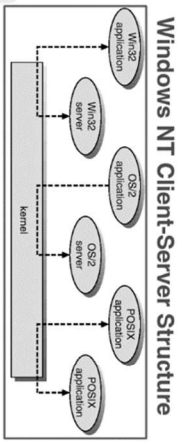




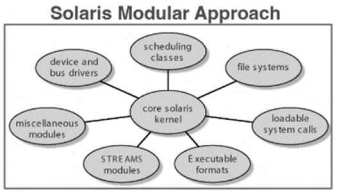
4.分层结构: 操作系统被分成很多层，最底层是硬件，最高层是用户接口。每层仅使用低层提供的函数和服务。模块化(易于 debug 和维护)，但并不总能实现(虚存层和进程调度层的关系)既有独立于机器的也有依赖于机器的分层系统(平台间迁移，平台迭代)

5.微内核 microkernel: 尽可能从内核向用户空间移动，使用信息传递来在用户模块间通信，便于扩展，便于迁移到新架构，更可靠(内核态代码少)，更安全，但性能会被用户与内核空间通信制约

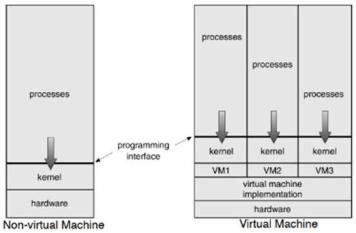




6.模块化: 现代操作系统实现内核模块，使用面向对象，内核组件分离，通信使用接口，可加载；和分层类似但更灵活



7.虚拟机: 把分层尝试进行到底，把硬件和操作系统内核都视作硬件，提供和硬件相同的接口，操作系统创造出多进程的假象，每个只在自己的处理器和自己的虚拟内存上运行

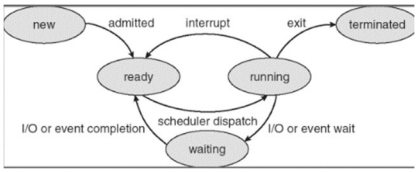


隔离，不干扰正常的系统操作；难以实现对物理机器的完全复制

8.policy: 要做什么； mechanism:怎么做

9.算法，模型，系统配置，硬件发展导致 OS 发展

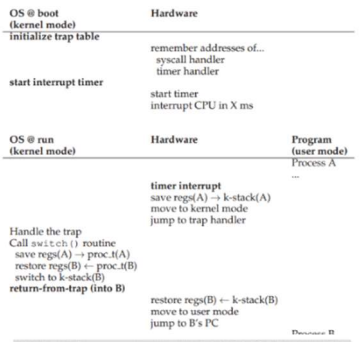
10.运行和管理任务

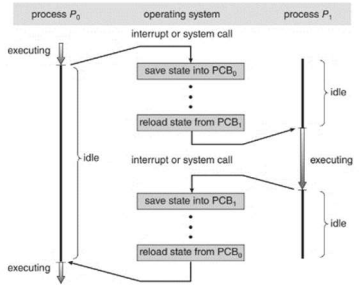


3-进程调度(如何切换，如何理解中断，中断如何工作)

1.**进程**：可并发执行的具有独立功能的程序关于某个数据集合的一次执行过程，是操作系统进行资源分配和保护的基本单位；三个方面：程序的一次运行活动，建立在某个数据集合上，在获得资源的基础上

2.上下文切换和中断：用户进程中止,OS 执行,用户进程恢复





3.算法(什么原则)

时机(何时分配 CPU)

进程切换(如何分配)

4.调度任务:控制协调进程对 CPU 的竞争,按一定的调度算法从就绪队列中选一个进程,把 CPU 使用权交给被选中的进程;原则：公平性,资源(CPU)利用率,交互式系统追求响应时间,批处理系统追求系统吞吐量,周转时间(进程提交到完成的时间间隔),等待时间(就绪队列中的时间之和)

5.进程是一次执行过程,程序是有序指令集合不涉及执行；不同进程可执行相同程序；进程有生命期

6.进程间有并发性,同时有多个进程,轮流占用 CPU 和各种资源;进程相互制约,资源共享和竞争

7.占用 CPU 方式:可剥夺式(preemptive),比正在运行的进程优先级高的进程就绪后系统可剥夺正在运行的进程 CPU 给高优先级进程;不可剥夺式(nonpreemptive)某一进程被调度运行后除非自身原因中止否则一直运行

①运行到等待②运行到就绪③等待到就绪④终止

非抢占式调度只发生在①④

8.经典进程调度算法:

-FIFO 先进先出：按照进程就绪的现后次序;实现简单;但没考虑进程优先级

-SJF 短作业优先：最少的平均等待时间

-PF 优先级调度：优先选择就绪队列中优先级最高的进程投入运行,优先级由优先数决定;静态/动态优先数

优先级因素:内部(时间、内存、打开文件数量、平均 IO 时间区间与平均 CPU 区间之比),外部(进程重要性、计算机支付的费用、赞助工作等)

问题:低优先级进程可能饥饿；可考虑 aging 来解决

-RR 时间片轮转:专为分时系统设计,类似 FIFO 但增加抢占来在进程间切换.

每个进程获得一小段 CPU 时间(时间片),时间片用完后 CPU 被抢占,进程排到就绪队列末尾.固定/可变时间片.

时间片大小与系统响应时间,就绪进程个数,CPU 能力有关.

性能很大程度上取决于时间片的大小:时间片很大就相当于 FIFO,时间片很小则响应时间短,但可能使得上下文切换开销过大

-多级队列反馈:允许进程在队列间移动以实现 aging.

队列数量,队列的调度算法,确定进程何时升级,合适降级,确定进程在需要服务时应进入哪个队列.

优先级高的为第一级队列,时间片最小,随队列级别降低,时间片加大.

各队列 FIFO;新进程就绪进入第一级队列;若进程等待则进入等待队列,等到后回到原就绪队列;优先级更高进程就绪可抢占 CPU,被抢占进程到原就绪队列末尾;第一级队列空就调度第二级队列;时间片到期后进程放弃 CPU 到下一级队列

-SRTF 最短剩余时间

-算法分析:SJF/SRTF 可用来最小化平均响应时间(SJF 对不可抢占 SRTF 对可抢占)且 SRTF 总不弱于 SJF.如果所有任务相同时间则 SRTF 和 FIFO 一样,如果任务不同时间则 SRTF(RR)保证短任务不会被长任务堵塞

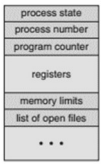
9.算法评估准则:CPU 利用率、响应时间、吞吐量

确定性建模,排队模型(到达率和服务率)

10.**可调度性**:存在一种调度策略使得可以在 ddl 前完成,条件：$U = \sum_i^n \frac{T_{cost}(i)}{T_{remains}(i)} \leq 1$

EDF(Earliest Deadline First)最优，只要可调度就能完成

11.进程状态:新建 new 运行 running 等待 waiting 就绪 ready 终止 terminated



进程控制块(process control block)对执行状态的快照,同时只有一个活动 PCB.

12.单处理器需要多路复用,每个虚拟 CPU 需要有 PC(program counter)和 SP(stack pointer)以及其他寄存器;在切换 CPU 时要保存当前的 PC,SP 等寄存器加载需要的 PC,SP 等寄存器;CPU 的切换由计时器,自发原因,IO 等触发

所有虚拟 CPU 共享非 CPU 资源(IO,内存),每个线程可以获取其余线程的数据,线程可以共享指令(对保护有挑战).这个模型在嵌入式应用很常见,yield 和计时器发起切换

13.程序地址空间：程序代码和数据都需要空间,各类资源需要被寻址

14.**中断**:程序执行中发生某个事件时中止 CPU 上现行程序的运行,引出处理事件的程序.充分发挥处理机的使用效率,提高系统实时处理能力.

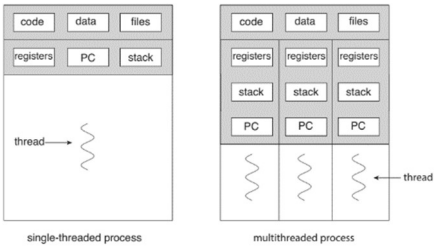
-分类:强迫(事故或外部请求信息)/自愿(对操作系统的需求),外中断(处理器和主存之外,可被屏蔽,由优先级决定)/内中断(异常,处理器和主存内部,不可被屏蔽),硬中断(硬件设施产生)/软中断(不必由硬件发信号,软件模拟,宏观异步)

-中断是由与现行指令无关的中断信号触发,终端发生与 CPU 处在用户模式或内核模式无关,两条机器指令间可响应中断.

-中断处理:保护现场,服务程序,恢复运行

-中断屏蔽:主机可允许或进制某类中断的响应,有些中断不能被进制.多重中断(中断嵌套)处理:中断处理期间CPU 又响应新的中断,关中断,定义中断优先级,开中断,响应并进行中断处理

4 线程



1.**线程**(轻量级进程):CPU 使用的基本单元,包括PC,寄存器集合和栈空间.线程和其他线程共享代码段、数据段、操作系统资源.这些线程被一起称为任务.进程和只有一个线程的任务等同.

在多线程任务中,一个线程被堵塞或在等待时,另一个线程可以运行.多线程的协作保证更高的吞吐量和更好的性能.需要共享相同缓冲区的应用依赖于线程的使用.

线程允许在实现并行同时保证顺序进程发起阻塞系统调用.

-应用程序通常实现得像一个具有多个控制线程的独立进程

-优点:响应度高(部分阻塞仍能执行),资源共享,创建和切换代价小,利用多处理器体系结构

-进程是资源的拥有者与调度单位(执行轨迹),线程是进程内一个相对独立的可调度的执行单元,进程中的实体

-线程状态:寄存器状态、堆栈、线程运行状态(执行、就绪、阻塞)、优先级、专有存储器、信号屏蔽

2.

-用户级线程(user level thread, ULT):应用程序通过线程库(创建、撤销、信息数据传递、调度执行、上下文保护和恢复)完成管理,内核不知道线程存在,线程切换不需要内核态特权,调度由应用程序决定,可以选择最好的算法,可运行在任何操作系统上,内核阻塞进程后所有线程将被阻塞,同一进程中的两个线程不能同时运行在两个处理器上

-内核级线程(kernel level thread, KLT):所有线程管理由内核完成,没有线程库但提供 API,内核维护上下文,线程切换由内核支持,以线程为基础调度,多处理器内核可以同时调度多个线程对多 CPU 体系结构支持更好,阻塞在线程一级完成相互独立,线程切换调用内核导致速度下降比用户线程慢

3.多线程模型

-多对一:多个用户线程映射到一个内核线程,线程管理在用户空间完成效率高,多用于不支持多内核线程的系统,但一个线程阻塞导致整个进程阻塞,不能并行在多处理器上

-一对一:每个用户线程映射到一个内核线程,提供了更好的并发性(某个线程阻塞不一定导致进程阻塞),支持多处理器,创建用户线程开销大

-多对多:多路复用了很多用户线程到同样或更小数量的内核线程上 m:n(m>=n),允许编程人员创建足够多的用户线程,避免线程阻塞引起进程阻塞,支持多处理器

4.线程取消:线程完成之前终止进程的任务.异步取消:线程立即终止目标线程.延迟取消:目标线程不断检查,允许在安全的点被取消.

5.同属一个进程的线程共享进程数据.线程特定数据:每个线程需要一定数据的拷贝.事务处理系统中每个事务有唯一标识符,线程特定数据可用于将每个线程与唯一标识符关联.

5 同步互斥

1.**并发** Concurrent:并发线程会在同时存取共享数据时带来问题(乱序, 一致性)需要同步来执行

2.**原子**操作 atomic:无法中断的操作

3.**竞争**:race condition:两个进程竞相访问同一数据,结果可能会被破坏或被错误解释.共享数据的值取决于哪个进程最后完成,为防止竞争,并发进程必须同步.

T_0:	*producer*	execute	$register_1 = count$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = count$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$count = register_1$	$\{count = 6\}$
T_5:	*consumer*	execute	$count = register_2$	$\{count = 4\}$

临界资源:进程间资源访问,共享变量修改,操作顺序冲突.

进程间制约:间接制约(竞争,互斥),直接制约(协作,同步)

4.**临界区问题**:设计一个协议以便协作进程.

没有两个进程可以在**临界区**(一段代码)内同时执行

进入区:请求许可的代码区段.

退出区:临界区之后.**剩余区**.

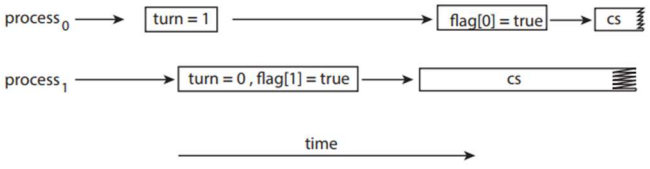
5.临界区问题需要满足**互斥** mutual exclusion 临界区内只能有一个进程、**进步** progress 若临界区无进程在执行且有进程要进入临界区则只有不在剩余区内的进程可以参与选择、**有限等待** bounded waiting 从一个进程做出进入临界区的请求到被允许为止,其他进程允许进入临界区的次数有上限

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

6.**Peterson 算法**: flag[i] 代表P_i申请进入临界区 turn 代表当前P_{turn}进入临界区.但在现代 CPU 体系结构上因为乱序多发射 Peterson 算法不一定可行.

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

    /* critical section */
```



7.**硬件同步**:memory barrier 保证先后顺序; 原子指令 test_and_set()和 compare_and_swap();原子变量.

8.**互斥锁** mutex lock 获取 acquire()释放 release().

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;
}

release() {
    available = true;
}
```

此处为**自旋锁**,忙等待(busy waiting)浪费 CPU 周期.等待锁时无上下文切换,适合使用锁事件较短情况,多处理器.

9.**信号量** semaphore S 是整型变量,除初始化外只能用原子操作 wait()(P,测试)和 signal()(V,增加)访问——作为 OS 核心代码执行,不受进程调度打断

```
wait(S) {
    while (S <= 0)
      ; // busy wait
    S--;
}
```

```
signal(S) {
    S++;
}
```

-二进制信号量类似互斥锁.

-计数信号量可用于控制多个实例的某种资源,初值为可用资源量.使用时 wait()释放时 signal(),计数为 0 代表资源都在使用中,进程会被阻塞知道计数大于 0.

保证S_1后执行S_2,但为忙等

```
S1;
signal(synch);
```

```
wait(synch);
S2;
```

-信号量的实现利用 sleep()和 wakeup(P)如下定义

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            add this process to S->list;
            sleep();
        }
}
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            remove a process P from S->list;
            wakeup(P);
        }
}
```

此时信号量可为负值(**绝对值**为等待临界区的进程数),但忙等时的非负.这样实现的信号量可以保证在等待时挂起,释放占用的 CPU 用于调度其他的任务(**让权等待**).

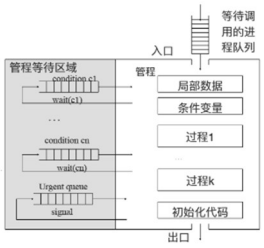
-And 型信号量: 将进程在整个运行过程中需要的所有资源一次性全部分配给进程,待使用完后一起释放.对若干临界资源的分配,要么全部分配,要么一个也不分配.在 wait()中添加了 AND 条件,变为 Swait()(simultaneous wait)

-优先级反转:高优先级需要资源被低优先级占用,低优先级被中优先级阻塞.优先级继承协议.

```
Swait(S1, S2, ..., Sn)
  if S1≥1 and ... and Sn≥1 then
    for i := 1 to n do
      Si := Si-1;
    endfor
  else
      place the process in the waiting queue associated with the first
    Si found with Si < 1, and set the program count of this process to
    the beginning of Swait operation
  endif
```

```
Ssignal(S1, S2, ..., Sn)
  for i: = 1 to n do
    Si=Si+1;
  Remove all the process waiting in the queue associated with Si into
the ready queue.
 endfor;
```

-信号量集:每个信号量S_i有临界值t_i以及每次申请d_i,Swait(S,d,d)在$S \geq 1$时允许多个进程进入临界区,而当$S = 0$时阻止进入临界区.



-wait signal 使用简单,但不够安全可能出现死锁且实现复杂

10.**管程** monitor:高级同步工具.管程类型属于 ADT 类型,提供一组由程序员定义的在管程内互斥的操作.管程类型包括用于定义类型的实例状态和操作变量的函数实现.只有管程内定义的函数才能访问管程内的局部变量.

管程确保每次只有一个进程在管程内处于活动状态.程序员不需要明确编写同步约束.

-因为信号量易读性差,不利于修改维护且正确性难以保证,引入管程.将分散的临界区集中管理.防止进程的违法同步操作.便于高级语言书写且便于正确性验证.

-共享性,安全性,互斥性.

-模块化:一个管程是一个基本程序单位,可以单独编译

-信息掩蔽:半透明,外部函数实现了某些功能,外部不可见其怎么实现.且外部不可见内部共享变量,只能调用外部函数间接访问共享变量

-管程互斥进入.有进程等待队列.

-定义附加的同步机制:条件(condition),条件变量 x 只可调用 x.wait()和 x.signal().与信号量不同的是如果没有挂起进程则 x.signal()无作用.

-一个进程唤起另一个进程可能**唤醒并等待**或**唤醒并继续**书 P190

-使用信号量的管程实现

-管程内的进程重启:条件等待 x.wait(c)其中 c 为优先值最小会被先启动.

-问题:无访问权限时访问资源,获得访问权限后不释放,未请求前释放,请求同一资源两次而不释放

-检查条件:用户进程总按正确顺序调用管程,不遵守协议的进程不能访问共享资源

-管程定义的是公用数据结构,进程定义的是私有数据结构;管程把共享变量同步操作集中,进程定义的是私有数据结构;管程是为管理共享资源而建立,进程是为占用系统资源和实现系统并发性引入.

11.有界缓冲区 bounded-buffer(**生产者-消费者**)问题:一个有限缓冲池(一/多个缓冲区),两种进程,生产者把产品放入缓冲区,消费者把产品取出缓冲区.如有 n 个缓冲区的缓冲池.所有进程异步运行,但生产者和消费者必须同步,不允许消费者从空缓冲区取,不允许生产者向满缓冲区放.生产者等待满缓冲区有空间,消费者等待空缓冲区有产品.如只使用一个 counter 变量,则会发生竞争.

-使用信号量: wait(mutex)和 signal(mutex)要成对

```
while (true) {
   . . .
  /* produce an item in next_produced */
   . . .
  wait(empty);
  wait(mutex);
   . . .
  /* add next_produced to the buffer */
   . . .
  signal(mutex);
  signal(full);
}
```

```
while (true) {
  wait(full);
  wait(mutex);
   . . .
  /* remove an item from buffer to next_consumed */
   . . .
  signal(mutex);
  signal(empty);
   . . .
  /* consume the item in next_consumed */
   . . .
}
```

对资源信号量的 wait()和 signal()出现在不同进程中每个进程的 wait 操作顺序不能颠倒,要先对资源信号量再对互斥信号量,否则可能引起死锁.

如果使用 And 型信号量则将原本的两行改为一行.

-管程.建立一个管程包括 put 和 get 两种过程.

```
producer:begin
      repeat
          produce an item in nextp;
          PC.put(item);
      until false;
      end

consumer:begin
      repeat
          PC.get(item);
          consume the item in nextc;
      until false;
      end
```

```
procedure entry put(item)
   begin
      if count≥n then notfull.wait;
      buffer(in) : = nextp;
      in : = (in+1) mod n;
      count : = count+1;
      if notempty.queue then notempty.signal;
   end
procedure entry get(item)
   begin
      if count≤0 then notempty.wait;
      nextc : = buffer(out);
      out : = (out+1) mod n;
      count : = count-1;
      if notfull.quene then notfull.signal;
   end
```

12.**哲学家进餐** dining-philosophers 问题:可能死锁,即都获取了一部分临界资源,且无法获得另一部分临界资源,可以用 And 型信号量解决.

13.**读者-写者** reader-writer 问题:共享数据库.

-互斥锁 rw_mutex 和读计数器 read_count(用 mutex 上锁)写者和第一个以及最后一个读者使用 rw_mutex 而读者在改变 read_count 时要用 mutex

-读写锁,可指定读/写访问.多个进程可并发申请读模式,但只有一个进程可获取写模式.可以用信号量集实现.Swait(L,1,1)与 Swait(mx,1,0)这对于易识别出哪些进程只读哪些进程只写的程序以及读者进程比写着进程多的程序有用.虽然读写锁的开销大于信号量或互斥锁,但若读者并发程度很高则可以弥补.

14.进程通信

-管道 Pipe:半双工,单向流动,只在有亲缘关系的进程间使用,允许一个进程和另一个和它有共同祖先的进程间通信.

协调以下三个方面:互斥,同步,确定对方是否存在.

管道速度慢容量有限,只有父子进程可通讯

流管道:双工,可双向传输

命名管道:允许无亲缘关系进程通信.

命名管道任何进程都可以通讯,但速度慢

-信号量:控制多个进程对共享资源的访问,作为锁机制.作为同步手段.

不能传递复杂消息,只能用来同步

-消息队列:消息的链表,存放在内核中由消息队列标识符标识.有权限的进程可以向队列中添加消息.被赋予读权限的进程可以读走队列中的消息.

相比信号量,可承载信息量大;相比管道,可以承载带格式字节流,缓冲区大小不受限.

但容量收到系统限制,且要注意考虑上次没有读完数据的问题

-内存映射:允许任何多个进程间通信,每一个进程把一个共享的文件映射到自己的进程地址空间

共享存储:映射一段能被其他进程访问的内存,让多个进程可以访问同一块内存空间,这段共享内存由一个进程创建,但多个进程都可以访问.

是最快的IPC方式,针对其他进程间通信方式运行效率低而专门设计.多与其他通信机制配合使用.

容量可控,速度快,弹药保持同步.

-套接字:可用于不同机器间进程通信.

-若用户传递信息比较少,需要信号来触发行为,可考虑软中断信号或信号量;若进程间传递信息量较大或存在交换数据的要求可考虑管道,命名管道或消息队列.

15.消息传递通信:以格式化消息为单位,隐藏通信的实现细节,简化复杂性.

-直接通信方式: Send(Receiver, message)和 Receive(Sender, message)原语.

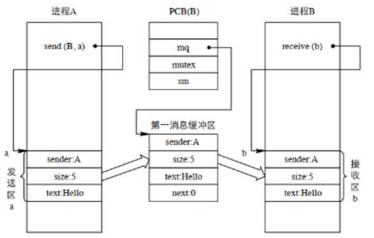
可以用来处理生产者消费者问题:生产者用 Send 原语发送给消费者而消费者则用 Receive 原语得到一个消息.

-间接通信方式:信箱创建原语和信箱撤销原语.利用信箱通信使用公享信箱 Send(mailbox, message)和 Receive(mailbox, message).信箱可由操作系统(公用信箱,提供给所有核准进程使用),用户进程(私有信箱,是创建者的一部分,其他进程只能写不能读;共享进程,指明共享用户,共享者可以写也可读)创建.

一对一, 多对一, 一对多(发送进程广播), 多对多

-通信链路:显示连接/利用发送原语让系统自动建点-点,多点; 单向,双向

-变长/定长消息格式.




```
type message buffer=record
        sender; 发送者进程标识符
        size; 消息长度
        text; 消息正文
        next; 指向下一个消息缓冲区的指针
    end
procedure send(receiver, a)
  begin
    getbuf(a.size,i);          根据a.size申请缓冲区;
    i.sender: = a.sender;      将发送区a中的信息复制到消息缓冲区之中;
    i.size: = a.size;
    i.text: = a.text;
    i.next: = 0;
    getid(PCB set, receiver,j); 获得接收进程内部标识符;
    wait(j.mutex);
    insert(j.mq, i); 将消息缓冲区插入消息队列;
    signal(j.mutex);
    signal(j.sm);
  end
```

```
type processcontrol block=record
  ...
  mq; 消息队列队首指针
  mutex; 消息队列互斥信号量
  sm; 消息队列资源信号量
  ...
  end
```

```
procedure receive(b)
  begin
  j : = internal name; j为接收进程内部的标识符;
  wait(j.sm);
  wait(j.mutex);
  remove(j.mq, i); 将消息队列中第一个消息移出;
  signal(j.mutex);
  b.sender : = i.sender; 将消息缓冲区i中的信息复制到接收区b;
  b.size : = i.size;
  b.text : = i.text;
  end
```

6 死锁

1.死锁:进程永远不能完成,系统资源被阻碍调用,组织其他作业执行.

2.必要条件:

-互斥(mutual exclusion): 至少一个资源处于非共享模式,只有在被释放后才可被申请

-占有并等待(hold and wait): 进程应占有至少一个资源且等待另一个资源

-非抢占(no preemption): 资源不能被抢占,只能被自愿释放

-循环等待(circular wait): 有一组进程循环等待资源

3.系统资源分配图(system resource-allocation graph):有向图,P为进程集合,R为资源集合,资源申请边从P到R,资源分配边从R到P.如果资源分配图没有环则系统没有进程死锁,如果有环则可能死锁.

4.死锁定理:当且仅当该状态的进程资源分配图是不可完全简化时,该状态为死锁状态.

可完全简化意味着可以将所有进程化成孤立节点.

5.处理死锁可以:

-允许进入死锁,然后检测后修复.需要死锁检测算法,可以强制抢占资源

-保证系统不进入死锁,这需要监视所有的情况,拒绝哪些可能会导致死锁的

-忽视死锁,鸵鸟算法,大部分系统都这样做

6.死锁恢复:

-立即结束所有进程,重启 OS,简单但损失大

-撤销陷于死锁的所有进程,解除死锁继续运行

-逐个撤销陷于死锁的进程,回收资源直至死锁解除

-剥夺陷于死锁的进程占用的资源,但不撤销进程

-回退 checkpoint,直到解除死锁

-若存在某些未卷入死锁的进程可能在执行到结束后可以释放资源解除死锁

7.死锁预防:破坏四个必要条件

-互斥:通常无法破坏

-持有等待:①进程在执行前申请获得所有资源.②进程仅在没有资源时才可申请资源,申请更多资源前应释放所有资源.

资源利用率低,可能发生饥饿.

-无抢占:如果一个进程持有资源并申请另一个不能立即分配的资源,即进入等待状态,那么其占有的资源可被抢占.仅适用于 CPU 寄存器和内存,不适用于互斥锁信号量.

-循环等待:对所有资源类型进行完全排序,要求按照递增顺序请求资源.

8.静态分配:一个进程必须在执行前申请全部资源.

层次分配:进程得到某一层的一个资源后只能再申请较高层次的资源.要释放某层的资源时必须先释放更高层次的资源.得到某层的资源后想申请同层另一个资源时必须先释放该层的已占资源.

按序分配资源:排序后按递增顺序申请.

9.死锁避免:每次进行资源分配时,判断系统状态来决定分配后是否存在不安全状态.

方式:每个进程生命其所需每种资源的最大数目,动态检查当前资源分配状态.

安全状态 safe state:如果系统能按照某个顺序为每个进程分配资源并能避免死锁,则系统状态是安全的.只有存在一个**安全序列**,系统才处在安全状态.并非所有非安全状态都导致死锁.因此确保系统不进入非安全状态,就可以确保系统不进入死锁.

10.在资源分配图中,只在某条资源申请边变为字眼分配边不会成环时允许申请,可保证不进入非安全状态.

11.银行家算法:总资源 E,已分配资源 P,剩余资源 A

```
type state= record        /*全局数据结构*/
  resource,available:array[0…m-1]of integer;
  claim,allocated:array[0…n-1,0…m-1]of
  integer;
end                       /*资源分配算法*/
```

```
if alloc[i,*]+request[*]>claim[i,*] then <error> /*申请量超过最大需求量*/
  else
    if request[*]>available[*] then <suspend process>
    else /*模拟分配*/
     <define newstate by:
      allocated[i,*]:=allocated[i,*]+request[*]
      available[*]:=available[*]-request[*]>
    end;
    if safe(newstate) then <carry out allocation>
    else
     <restore original state> <suspend process>
    end
  end
```

```
function safe(state:s):boolean;  /*banker's algorithm*/
 var currentavail:array{0…m-1} of integer;
   rest:set of process;
begin
 currentavail:=available;
 rest:={all process};
 possible:=true;
 while possible do
  find a P_k in rest such that
   claim[k,*]-alloc[k,*]≤currentavail;
  if found then
    currentavail:=currentavail+allocation[k,*];
    rest:=rest-[P_k];
     else
      possible:=false;
  end
 end;
 safe:=(rest=null)
end.
```

12.死锁检测

-每种资源一个实例:等待图,删除资源分配图里的资源类型节点,合并适当边.即等待图里只有进程节点,且一条边代表等待关系.**当且仅当等待图里有环时系统死锁.**

-每种资源多个实例:使用银行家算法

7 内存管理

1.时间/空间局部性:低端存储容量大,高端存储速度快

2.内存:由很大一组字(节)组成,每个字(节)有自己的地址

3.输入队列:磁盘上等待进入内存并执行的进程集合

4.内存共享需要考虑

-进程和内核工作状态取决于存储在内存/寄存器的数据

-不同的进程/线程控制部分不能使用内存同一部分.物理上,不同部分的数据不能使用内存的同一地址.

-不同线程的内存资源也需要 private 化(protection)

5.存储管理**功能**

-地址变换/地址重定位:将程序地址空间的逻辑地址变成主存中的地址,建立虚实地址的对应关系.

-主存分配:按照一定算法把空闲的主存区分配给进程

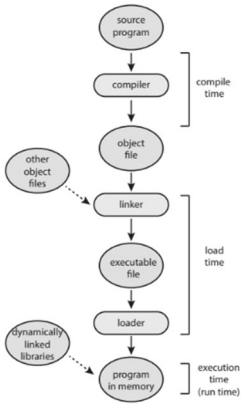
-存储保护:保证进程在各自的存储区域操作,互不干扰

-虚拟存储:使用户程序的大小和结构不受主存容量和结构的限制,即使在用户程序比实际主存容量还要大的情况下,程序也能正确运行

6.**地址变换**:逻辑地址(用户编程时使用的地址)与物理地址(内存中若干大小相等的存储单元对应编号);逻辑地址空间(程序生成的逻辑地址集合,一维或多维)与物理地址空间(内存中所有物理地址的集合,一维线性)

用户程序看不到真正的物理地址,只生成逻辑地址.将程

序中逻辑地址变换成内存中的物理地址由内存管理单元(Memory Management Unit, **MMU**)完成.
7.**内存保护**:保护操作系统不受用户进程影响,保护用户进程不受其他用户进程影响.
-上下界保护:下界寄存器存放开始地址,上界寄存器存放末地址.
-基址限长寄存器:基址寄存器存起始地址,限长寄存器存放最大长度
对于合法访问地址,两者效率相同.对于不合法访问地址,上下界保护浪费 CPU 时间更多.
8.地址绑定
-编译时,确定绝对代码.如发生改变,需要重新编译代码.不能做任何移动,否则会出错.
-加载时,确定可重定位代码,如发生改变只需重新加载用户代码.由装入程序将用户程序转换.无需硬件支持,但移动依然较困难.且装入时就要全部转换
-执行时,进程在执行时可在内存中移动,需要特定硬件支持.基址寄存器此时称**重定位寄存器**.**动态**变换,不执行的程序就不做地址转换的工作,节省 CPU 时间.重定位寄存器由特权指令设定,比较灵活.



-动态加载,子程序只在调用时才被加载到内存.不用的子程序不被装入内存,保证更好的内存空间利用率.适合大量代码处理不经常发生的事件.不需要操作系统的特别支持,通过程序设计实现.
-动态链接:链接到用户程序来运行.链接被推迟到执行时期.二进制映像中每个库程序的引用有一个存根(stub,一小段代码,指出如何定位适当的内存驻留库程序)使用语言库的所有进程只需一个库代码副本.动态链接需要操作系统的帮助,因为只有操作系统可以检查所需程序是否在某个进程的内存空间内或多个进程访问同一内存地址.
9.动态地址变换的优点:提供用户程序任意分配内存区的能力;可实现虚拟存储;提供重新分配的能力;一个用户程序可以分配到多个不同的存储区.
10.连续内存分配
主存常分为操作系统保留区以及用户进程区.默认操作系统位于低内存,用户进程保存在高内存.
每个程序占用连续的空间,分单用户连续存储管理,固定分区存储管理,可变分区存储管理.定位又分动态和静态.
-单用户连续存储管理:**单分区**模式,适合单用户,任何时刻主存最多一道程序.主存分系统区和用户区.静态重定位,栅栏寄存器保护;动态重定位,定位寄存器保护.
缺点:系统利用率低.
-固定分区:定长分区或**静态分区**,满足多道程序设计的最简单的存储管理.给进入主存的用户作业划分一块连续的存储区域,把作业装入该连续存储区域.若多个作业装入主存,可以并发执行.适合已知作业大小和出现频率.
系统启动时,系统操作员根据作业情况静态分割用户空间,每个空间位置固定,大小可能不同,每个分区同时最多装入一道程序.
静态定位(绝对地址是否落在分配的用户分区)与动态定位(上下限寄存器)
调度策略:每个等待作业被选中,排到一个能够装入它的最小分区的等待队列(可能导致分区使用不均);所有作业排队,调度其中一个,选择可容纳它的最小可用分区
缺点:系统运行时,难以预知分区大小.主存空间利用率低.无法适应动态扩充主存.分区数目预先确定,限制了多道运行程序的数量.
-可变分区:变长分区.按作业的大小划分分区,划分的时间、大小、位置均动态确定,在将作业装入主存执行前不建立分区.
需要已分配区表和未分配区表.
11.动态存储分配问题的算法
-首次适应:按空闲区首址升序成表.从表首起依次比较到满足要求.若空闲大于请求,将空闲的一部分分给请求者,修改空闲区的大小和首址.
实质:尽可能利用低地址部分的空闲区,尽量保证高地址的大空闲区不被切小.在有大作业到来时有足够大的空闲区满足请求者.
回收时若释放区和某个空闲区相邻应合并.
-最佳适应:按空闲区大小升序顺序成表.将申请者放入与其大小最接近的空闲区.切割后的空闲区最小.若有与申请大小相等的空闲区,一定能正好分配对应大小.
缺点:分割后的空闲区将很小,直至无法使用,造成浪费.
-最坏适应:按空闲区大小降序顺序成表.每次分配时将最大的空闲区切出一部分分给请求者.避免空闲区越分越小.每次总取第一个表项,若不能满足,则失败.
-下次适应:用额外的指针保留上一次找到的空闲分区的下一分区的地址,下次查找时从此地址开始查找,并且采用循环查找方式
-快速适应:把具有相同容量的所有空闲分区单独设置一个空闲分区链表,再设立一张管理索引表管理所有分区表.在进行分配时,先根据进程长度从索引表找到能容纳它的最小空闲区大小对应的链表,然后直接取出其中的第一块即可
12.**碎片**:分区存储管理的系统中会形成一些非常小的分区,不能被任何程序使用而浪费.外部碎片(整个内存空间可以满足一个请求,但不是连续的)内部碎片(分配的内存可能比申请的内存大)
通过**移动**技术来减少外部碎片,把小的空闲内存结合成一个大块.只有在执行时进行动态重定位才能移动.
13.在需要的内存空间比主存大时,需要使用**交换**技术,但开销会比较大.同时要考虑进程真正需要的空间和总共需要的空间的不同.需要确保被换出的部分是真正空闲的,因为可能在等待 IO.

14.**覆盖**技术:将大程序划分为一系列的覆盖,每个是相对独立的程序单位,把程序执行时不需要同时装入内存的覆盖成组为覆盖段.每个覆盖段内的覆盖共享同一区域,称覆盖区,大小由覆盖段内最大覆盖决定.

15.**挑战**:进程动态增长,碎片,交换延时长.分页/分段

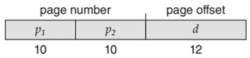
进程增长:每个进程可以使用任意空闲段.碎片:进程不一定要连续,不再需要去移动.交换延时长:只交换某一部分.

16.**分页式**存储管理:允许作业存放在若干个不相邻的分区中,免去移动内存信息的工作量,充分利用主存空间,尽量减少主存碎片.

程序地址空间分成大小相等的页面,并把内存也分成与页面大小相等的块.装入内存以页为单位.

-帧 frame:主存空间的分区

-页 page:程序逻辑地址的分区



-虚地址:

映射方式:页表数组,常数复杂度,但可能会占用较大空间.

页表基址寄存器 PTBR+页表限长寄存器 PTLR

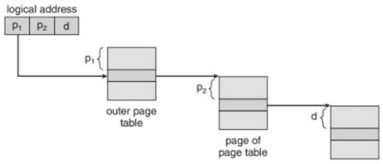
(链表太慢,hash 无法保证有界,树需要访问多次)

用户看,内存是线性数组;OS 看,每个进程有多个帧.不会有外部碎片,但可能有内部碎片,平均大小为页大小一半.

缺陷:每次访问页都需要两次访存,一次页表一次页.

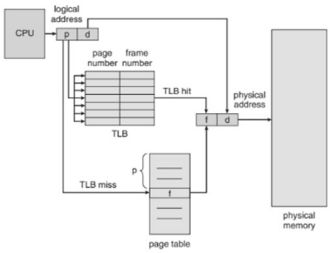
17.解决页表规模大:**多级页表**,页表部分存在内存中.

e.g.二级页表,一次主存访问需要三次访存,访问页目录,访问页表,访问数据.



18.解决多次访存:TLB(translation look-aside buffer)局部性原理

块表:存放在 TLB 中的部分页表内容.



并行搜索,如果在 TLB 中则得到对应帧号,否则查找内存中页表得到帧号.设命中率α,TLB 查找时间ε,访存时间t,有效访问时间(Effective Access Time)

$$EAT = (t+\varepsilon)\alpha + (2t+\varepsilon)(1-\alpha)$$

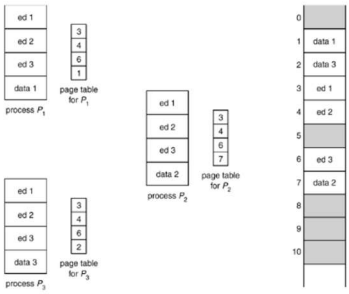
19.页表的标志位,由 OS 指定,硬件更新

-valid:**是否在进程的逻辑地址空间内**,无效对应不允许访问,需要触发异常

-protection:可读/可写/可执行,存储保护

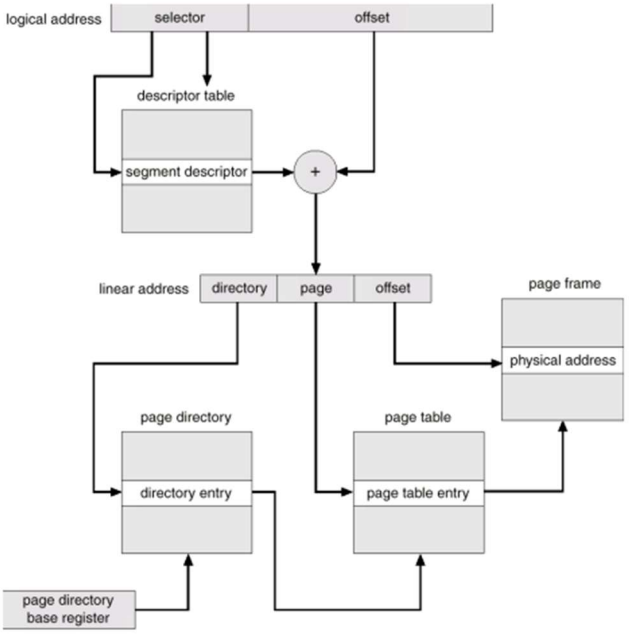
-reference/dirty:最近被访问/修改,用于换页算法

20.共享页:一段只读(可重入)代码可由进程共享.共享代码出现在进程逻辑地址空间的相同位置.私有代码和数据:每个进程保留代码和数据的私有副本,可以出现在逻辑地址空间的任何地方.



21.**分段式**存储管理:满足程序设计和开发(模块为单位的装配、共享和保护),程序是一些段的集合,段是逻辑单位.段表,含段基址和段长.段共享通过不同作业段表项指向相同段基址实现.

22.页是物理单位,分页是为了减少外部碎片.段是逻辑单位,分段是为了更好满足用户需要.页大小固定由系统决定,逻辑地址划分为页号和页内偏移是由硬件实现,系统内只有一种大小的页面,段长不固定,取决于用户编写的程序,由编译程序在编译时划分.分页的作业地址空间是一维的,只有地址,而分段是二维的,既有段名又有段地址.



8 虚拟内存

1.事实:数组、链表和表通常分配比需要更多的内存.程序某些部分可能很少使用,即使需要完整程序,也不会同时需要.

2.保存部分程序在内存中,可以运行一个比物理内存大的程序.逻辑地址空间比物理地址空间大.可以有更多程序同时运行.运行若干进程共享地址空间.进程创建更高效.

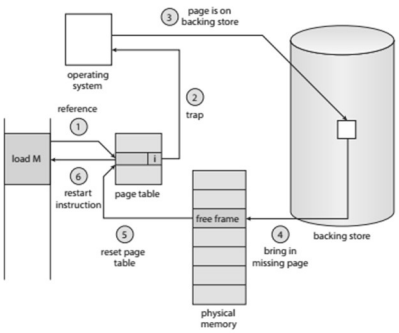
3.理论基础:空间局部(同时只访问一小部分数据/代码)时间局部(最近访问的数据/代码可能很快又被访问)

4.虚存以页或段为单位而交换技术以进程为单位.进程所需主存大于系统空闲仍能用虚存运行,交换技术不行.

5.请求分页式:将作业信息分成多个页面,当作业被调度运行时仅装入需要立即访问的页面,执行中如果需要访问页面不在主存再动态装入.

响应快,适合多用户

硬件支持:MMU(页表基址寄存器+TLB)负责地址转换,产生需要的硬件中断



6.需要调页但没有空闲帧:将暂时不能运行或暂时不用的数据调出到辅存,再把需要的数据调入主存.
7.问题:主存与辅存的一致性,逻辑地址到物理地址的转换,部分装入和部分对换
8.页面装入策略:
-请页式:缺页中断驱动,一次调一页
-预调式:按某种预测算法动态预测并调入若干页
9.清楚策略:
-请页式:仅当被选中替换时,将被修改的写回辅存
-预约式:内容被修改页面成批写回辅存,被替换前发生
10.有效访问时间=(1-p)内存访问时间+p(页错误开销+换出开销+换入开销+重启开销), p 为缺页概率,换出开销=换入开销*被修改概率.可计算出需要降低缺页率.
11.写时复制:复制进程后可以共享页,只在写其中内容的时候进行复制.
12.影响缺页率因素:页替换算法,帧数,页大小,程序特性.
帧分配算法:固定/可变(缺页次数多,分配更多帧)
页替换算法:局部/全局替换
13.帧分配算法
-平均分配:未考虑进程本身大小
-按比例分配:根据进程大小按比例分配帧.应取整.
-考虑优先权分配:给重要、紧迫作业分配较多的内存空间.一部分按比例分配,另一部分根据进程优先权分配.
实时控制系统可能完全按优先权分配帧.
14.页替换算法:
-FIFO:队列管理,调入加到队尾,替换时选择最旧的
-最优页替换:未来最长时间不被使用的页先替换,因为需要未来的信息不易实现,可用来衡量其他算法的性能
-随机替换:TLB 采用,结构简单,性能难以预测
-LRU 替换:记录每个页最后使用时间,选择最近最长未被使用的页.可用栈实现,被访问则移到栈顶,栈底就是需要的.也可用计数器实现,选择最小计数器.
评价:LRU 可以降低缺页率,但也不一定.
添加内存大小可以降低 LRU 的缺页率,但对于 FIFO 也可能升高(Belady 异常)
15.LRU 近似替换:每个页有一个 reference bit,被访问则将其置为 1,每次替换第一个 reference bit 为 0 的页
-时钟算法/二次机会算法:FIFO+reference bit.形成循环队列,每次访问则置 1.替换时检查,若引用位为 1 则置 0,看下一个,若为 0 则替换.
-增强型二次机会算法:加入修改位,从(00)到(11)代表最近未被使用且未被修改;最近未被使用且已被修改;最近已被使用但未被修改;最近已被使用且已被修改.替换最低类型的第一个页面.这可能需要扫多次.
16.基于计数的替换
-LFU:最不经常使用页先换出去
-MFU:最常使用页替换算法,因为使用次数少的可能是刚被替换进来的,而最常使用的可能之后不用了
17.**内存抖动** thrashing:一个进程没有足够页,缺页率较高,频繁换入换出页,CPU 利用率低下,操作系统认为应该继续添加程序,导致缺页率更高.
原因:局部内存和大于总内存
解决:需要检查进程真正需要多少帧. 操作系统选择挂起一个进程,把对应页换出.
如果缺页率太低就回收一些帧,如果缺页率太高就分配一些帧.
18.预先调页:降低缺页率,可能造成浪费.
19.减小页大小对应增大页表大小.
-减少碎片:小页更好利用内存
-减少 IO 开销:寻道延迟大于传输时间
-局部性:小页适合页匹配程序的局部
20.增大页大小:可能增加碎片,考虑提供多种页大小.
21.TLB 范围:TLB 条数*页大小.理想情况,每个进程工作都在 TLB 里.
22.锁定主存页,某些页(如在做 IO 的)不能被替换.

9 固态硬盘 Solid State Drive

1.固态硬盘是带电池的 RAM,快
2.NOR flash:按字存取,写较慢,密度低,启动嵌入式设备
3.NAND flash:读写按页(512B),擦除按块(16KB)
4.Intel 傲腾:字存取(快速随机存取),比 RAM 慢但比 NAND flash 快,无需写放大,磨损较小,但比 NAND flash 贵.
5.没有移动的部分,没有寻道延迟/旋转延迟.读比写快.
-空页可以被写
-被写的页再次写需要擦除
-但擦除是按块的,且较慢,每次写会影响整个块
6.优势:
-在便携式设备上更可靠,且没有噪声(没有移动的部分),
-启动快(无需转起来),
-极低读延迟(没有寻道时间,25us/4KB),
-确定的读性能(与数据位置无关)
7.劣势:较贵,写擦除次数有限,写入速度较慢,耗电量大
8.1 页 4KB,1 块 64 页,1 面 2048 块,1 片 4 面
9.写放大:只想写一块中的某个部分,却需要将整块读入内存,修改该部分,擦除 SSD 中整块,再写回.导致写放大,磨损也随之放大.
10.热区磨损:文件系统倾向于重复写入同一块,使得磨损更大.
解决方案:磨损分级,即虽然用户认为写的是某一块,但用 OS 将该地址重映射到别的地址上.FTL-flash translation layer

10 外存管理

1.存储介质:磁盘,磁带,光盘
2.物理块:文件的存储设备常被划分为若干大小相等的物理块,文件信息也被划分为相同大小的逻辑块,统一编号,以块为单位进行信息存储、传输和分配.
3.磁带:永久保存大容量数据的顺序存取设备,前面的物理块被访问后才存取后续的内容.存取速度较慢,多用于

后备存储,或存储不常用信息,或传递数据.
4.机械硬盘 Hard Disk Drive:盘面,磁道,扇区.
5.访问:读/写,设备号,盘面号,磁道号,扇区号,内存地址(源/目)寻道,旋转,传输
对于变换磁道:固定头(速度快,成本高)与移动头(速度慢成本低)
硬盘延迟=排队时间+控制器时间+寻道时间+旋转时间+传输时间
6.硬盘的权衡:扇区间需要有无效区域.假如扇区过小,空间利用率低,需要带宽小,假如扇区过大,空间利用率高,需要的带宽大
7.硬盘访问优化:
-循环排序:根据当前在读的扇区以及请求顺序而定
-优化:按照数据处理规律合理安排分布,提高处理效率
每次读完后还有处理时间.
考虑交替地址:数据的冗余存放提高访问速度,但会小号较多的空间,适合数据总是读取的情况,否则保证数据一致性损耗大
8.可靠性:Failure(actual deviates from specified behavior),Error(defect that result in failure),Fault(cause of error)
9.Reliability:平均出错时间 MTTF(Mean Time To Failure)
平均修复时间 MTTR(Mean Time To Repair)
Availability=MTTF/(MTTF+MTTR)
10.fualt:硬件,设计,操作,环境;瞬时,间歇,永久
11.提高可靠性:避免犯错,容错,去除,预测
12.RAID(Redundant Arrays of Independent Disk)独立磁盘冗余阵列,用一组较小容量的、独立的、可并行工作的磁盘驱动器组成阵列代替单一的大容量磁盘,加入冗余技术.
优势:数据分布存储,提高单个 IO 请求的处理性能;数据冗余,提高系统可靠性.
一组驱动器,但被操作系统看作是单一的磁盘驱动器.
-RAID0:数据划分成条块存在所有磁盘上,逻辑上连续的数据物理上横向相邻.吞吐量提高,可并行;延迟降低,每块硬盘有单独的排队队列.但没有冗余,实际上可靠性低于单块硬盘(如果其中一块坏了,可能很多逻辑上的空间都坏了)
-RAID1:磁盘镜像,写需要改两个数据子块
-RAID2:并行存取,有纠错
-RAID3:仅使用一个冗余盘,位交错
-RAID4:块交错
-RAID5:快交错分布式
-RAID6:两种冗余码
13.搜索定位:磁盘调度
-FCFS
-SSTF:Shortest-Seek-Time-First,可能导致饥饿
-SCAN:电梯算法
-C-SCAN:循环扫描
-C-LOOK:移到最远端

11 文件系统

1.用户直接操作和管理辅存信息易出错,引入文件系统
2.文件是由文件名字标识的一组有序信息的集合.用户不必关心实现细节.内容由文件的建立者和使用者解释.必须从逻辑文件和物理文件来了解.
文件系统将用户从复杂的物理存储地址管理中解放出来,对文件提供各种安全、保密和保护措施,实现对文件的共享.
3.文件系统:管理文件的软件即存取和管理信息的机构。它是操作系统中负责存取和管理信息的模块,它用统一的方式管理用户和系统信息的存储、检索、更新、共享和保护,并为用户提供一整套文件使用和操作方法,是管理文件需要的数据结构总体.
用户视角:文件有什么组成,如何命名,如何保护文件,何种操作
操作系统视角:文件目录如何实现,怎样管理存储空间,文件存储位置,磁盘实际运作方式等
4.功能:文件的按名存取,文件目录建立和维护,逻辑文件到物理文件的转换,文件存储空间的分配和管理,提供合适文件存取方法,实现文件的共享、保护和保密,提供一组可供用户使用的文件操作
5.文件系统的特点:友好的用户接口,对文件按名存取,某些文件可被多个用户或进程共享,可存储大量信息
6.文件操作:创建,改写,读取,重定位(搜索),删除,截短(保留文件属性,长度变为 0),重命名,拷贝,扩展
7.文件分类:对不同文件进行管理,提高系统效率,提高用户界面友好性
文件性质和用途:系统、用户、库文件
信息保存期限:临时、永久、档案文件
文件的保存方式:只读、读写、可执行、不保护文件
文件的逻辑结构:流式、记录式文件
文件的物理结构:连续、链接、索引文件
文件的存取方式:顺序存取、随机存取文件
设备的类型:磁盘、磁带、打印文件
文件的内容:普通、目录、特殊文件
8.文件属性:基本属性(文件名,所有者,授权者,长度)类型属性,保护属性(读,写,可执行),管理属性(创建时间,最后存取/修改时间)
文件保护:防止系统崩溃造成的文件破坏(定时转储,多副本)防止文件主和其他用户非法操作(三元组权限)
9.文件结构:逻辑结构与物理结构
-逻辑结构:无结构文件(字节或字序列)即流式文件,无格式限制.有结构文件即记录式文件,有格式限制.
选取原则:便于用户操作,使信息占据最小存储空间,方便文件系统定位需要修改的基本信息单位,且减少对已存储好的文件信息的改动.
-物理结构:顺序文件连续存储,存取效率最高.但查找修改单个记录,性能可能差.增加删除记录困难.

链接文件,使用链接字或指针来表示各记录之间的串联关系,存储空间利用率高,文件易动态扩充修改,顺序存取效率高.但随机存取效率低.
直接文件:在记录的关键字和物理地址间建立关系,哈希散列文件.
索引文件:文件的信息存放在不连续物理块中,建立索引表,保持链接结构优点,又利于顺序存取又利于随机存取,满足动态增长.但索引表本身有系统开销,且寻道次数多时间长.
10.构建物理结构方法:计算法(直接寻址文件,计算寻址文件,顺序文件)与指针法(索引文件,索引顺序文件,链接文件)
11.存取方法
-顺序存取:最简单,严格按照文件信息单位排列顺序存取,不必给出具体存取位置,总在前一次基础上进行
-直接存取:存取确定其实位置
-索引存取:基于索引文件(实际多用多级索引)
12.卷(物理卷),块(物理记录),逻辑记录(应用程序处理单位),存储记录(文件管理程序处理单位)
逻辑记录是按信息在逻辑上的独立含义划分的单位,块是存储介质上连续信息组成区域.一个逻辑记录可能扎用一块或多块,一个物理块也可以包含多个逻辑记录.
13.文件目录:文件系统建立和维护的关于系统的所有文件的清单,每个目录项对应一个文件的信息描述.每个目录项又称文件控制块(FCB)
操作系统为每个文件开辟一个存储区,其中记录文件的有关信息.FCB 中有文件控制信息,结构信息,使用信息,管理信息.
14.文件目录结构:
-一级目录结构:线性表,简单易实现,但可能重名,难以实现文件共享,文件平均检索时间长
-二级目录结构:一级为所有用户文件目录地址,二级为用户文件目录,解决文件重名问题和文件共享问题,查找时间低,但增大系统开销
-树形目录结构:文件的全名从根目录起到文件名止,较好反映现实世界数据集合间层次关系,不在同一目录可以重名,容易以目录为单位进行保护、保密和共享,高效检索,灵活组织,但因为放在外存访盘多次影响速度
可用绝对路径名和相对路径名去检索
15.文件系统的使用(控制台命令+系统调用)
16.文件系统的挂载:文件卷(文件子系统)存放文件和目录信息.硬盘一个分区称一卷.需要挂载才能被访问.
17.文件共享:可以节省大量外存空间,有效减少文件复制而增加的访问外存次数,进程间通过文件交换信息.
三种:存取权限控制,各用自己的读写指针,共享读写指针.静态共享,动态共享,符号链接共享.
一致性:一个用户对数据的修改是否可以何时可以被其他用户观察到
18.虚拟文件系统:支持多种文件系统,为用户提供一致接口,提供网络共享文件支持,可扩充新的文件系统
对多个文件系统的共同特征进行抽象,形成与具体文件系统无关的虚拟层,并在层上定义一致性接口.每个文件系统自包含,实现各自细节.
19.可靠性:抵抗和预防各种物理性和人为性破坏的能力处理坏块.备份.恢复.若写回磁盘前系统崩溃会出现不一致,要能检查磁盘和目录系统.
20.安全性:数据丢失,软硬件故障,人失误,可通过备份解决.入侵者,要考虑清楚.
原则:系统设计公开,缺省属性不可访问,检查当前权限,赋予进程最小可能权限,保护机制简单一致,采取方案必须可接受
21.文件系统效率:磁盘分配和目录管理算法,保留在文件目录条目中的数据类型
性能:磁盘缓存,马上释放和预先读取,虚拟磁盘

- FAT文件系统（MS-DOS文件系统、msdos）
  它是MS-DOS操作系统使用的文件系统，它也能由Windows98/NT、linux、SCO UNIX等操作系统访问。文件地址以FAT表结构存放，文件目录32B，文件名为8个基本名加上一个“.”和3个字符扩展名。
- FAT32文件系统（vfat）
  它是Windows98使用的扩展的DOS文件系统，它在MS-DOS文件系统基础上增加了对长文件名（最多到256B）支持。
- NTFS（NT文件系统）
  它是Windows NT操作系统使用的文件系统，它具有很强的安全特性和文件系统恢复功能，可以处理巨大的存储媒体，支持多种文件系统。
- S51K/S52K（sysv）
  它是AT&T UNIX S V 操作系统使用的1KB/2KB文件系统。

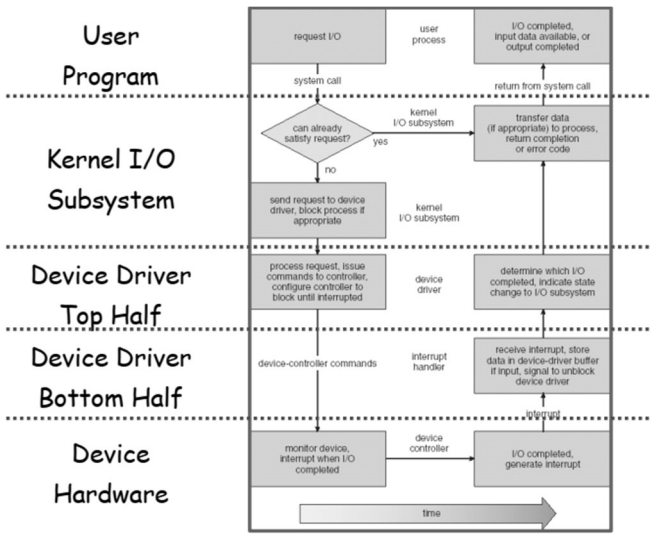
- UFS文件系统（Unix File System）
  基础块大小从 1024 字节增加到 4096 字节
  增加 Cylinder Group 的概念，优化数据和元数据在磁盘上的分布，减少读写文件时磁头寻道的次数（减少磁头寻道次数是 HDD 时代文件系统在性能优化上的一个主要方向）。
- HFS文件系统（HFS+）
  Time machine：将文件系统完整的回滚到之前的某一个时间点，可以用于对文件系统做备份，找回历史版本文件，也可以用于做系统迁移。
  硬伤太多，例如大小写不敏感，不支持 Sparse File 等
- APFS （苹果文件系统）
  能够适应多种应用场景，从智能手表，智能手机，到笔记本电脑，确保数据安全可靠，能够充分利用多核 CPU 以及新硬件设备的并发性

- ext系列（多级扩展文件系统）
  它是Linux操作系统使用的高性能磁盘文件系统，它是对Minux操作系统中使用的文件系统扩展（ext）的扩展。它支持256字符的文件名，最大可支持到4TB的文件系统大小。
- HPFS（高性能文件系统、hpfs）
  它是OS/2操作系统使用的文件系统。
- CD-ROM文件系统(iso9660)
  它是符合ISO9660标准的支持CD-ROM的文件系统，它有High sierra CD-ROM和Rock Ridge CD-ROM二种类型。
- UDF通用磁盘格式文件系统
  UDF(Universal Disk Format)文件系统是依据光学储存技术协会（Optical Storage Technology Association, OSTA）的通用磁盘格式文件系统规格1.02版所制定的。它提供了对 UDF格式媒体的只读访问（例如DVD光盘）。Windows98提供对UDF文件系统支持。

早期 UNIX FS 简单但慢,物理存储块过小,空闲区块未能有效组织,数据局部性不好.

12 I/O 设备管理

1.计算机外围设备分为存储型和输入输出型设备.设备管理目的:方便用户使用各种外围设备,提高外围设备并行性,提高利用率



2.设备管理功能:控制各类设备实现用户目标,向用户提供方便的设备接口,提高设备运行效率,实现对设备的管理和保护

3.输入输出系统:IO 设备+控制部件+通道+管理软件

4.设备分类:

传输速率:低速、中速、高速设备

信息交换的单位:块(传输速率高,可寻址,DMA)、字符设备(传输速率低,不可寻址,中断)

资源分配:独占(临界资源)、共享(提高设备利用率)、虚拟

输入输出特征:输入型、输出型、存储型

存取方式:顺序型、直接型

5.输入/输出处理:IO 控制器(管理软件)轮询,中断,DMA,通道

-轮询:程序直接控制,忙等,CPU 和 IO 串行,效率低

-中断:外围设备可以表达工作状态,CPU 和 IO 部分并行,数据传输时仍为穿行,效率有所提高

-DMA:IO 直接与主存交换数据,无需 CPU 过多干预,CPU 和 IO 间可以并行,但 CPU 需要在块与块之间对 IO 进行干预

-通道:减少 CPU 对 IO 的干预,只在开始启动时执行响应指令,在操作结束时中断通知执行代码,CPU 和 IO 完全并行,效率高

6. 总线系统:

- 计算机系统中的各部件如CPU、内存以及各种I/O设备之间的联系，都是通过总线来实现的。总线性能是用时钟频率、带宽和相应的总线传输速率来衡量。

1 ISA（Industry Standard Architecture）：原先带宽8位，2Mb/s速率，后为16位，16Mb/s速率。

2.EISA（Extended ISA）：带宽32位，速率32Mb/s。

3.局部总线：是指将多媒体卡、高速LAN网卡、高性能显示卡等，从总线上分离，通过局部总线控制器，直接接到CPU总线上，使之具有高速数据交换能力。

4.VESA（Video Electronic Standard Association）：32位，可达132Mb/s，但控制器无缓冲，且连Pentium也不支持。

5.PCI（Peripheral Component Interface）：支持64位系统，它在CPU和外设之间加了一层复杂的管理系统，用于协调数据传输和提供一致的接口。能适应高频率的CPU。

- USB技术：USB（Universal Serial Bus）通用串行总线，是一种连接I/O串行设备的技术标准。

（1）冲破了计算机技术发展的两个历史局限性：由于I/O设备的接口标准的不一致和有限的接口数量已无法满足各种应用迫切需要；传统的I/O设备的接口无法满足实时数据传输与多媒体应用的需求。

（2）USB以WDM（Windows Driver Model）模型为基础，WDM包含一套通用的I/O服务和二进制兼容的设备驱动程序。

（3）USB支持同步数据传输方式和异步数据传输方式，其数据传输率有低速1．5Mbps和全速12Mbps乃至上百Mbps，比标准串口快上百倍。

（4）USB可以主动为外部设备提供电源，允许外部设备快速连接，具有即插即用的功能。

（5）允许外部设备的热插拔。

（6）控制器的功能：控制器主要负责执行由控制器驱动程序发出的命令。

（7）控制器驱动程序：控制器驱动程序在控制器与USB设备之间建立通信信道。

（8）USB芯片驱动程序：USB芯片驱动程序提供了对USB的支持。

（9）USB设备分为两类：

1.USB集线器：本身可再接其他USB外围设备。

2.USB设备：连接在计算机上用来完成特定功能并符合USB规范的I/O设备单元，如鼠标、键盘等。

（10）4种不同的数据传输方式：

1.等时传输方式：以固定的传输速率，连续不断传输数据，发生错误时，USB不处理，而是继续传送新的数据。用于需要连续传输，且对数据的正确性要求不高而对时间极为敏感的外部设备，如麦克风、音箱以及电话等。

2.中断传输方式：该方式传送的数据量很小，但这些数据需要及时处理，以达到实时效果，此方式主要用在键盘、鼠标以及游戏手柄等外部设备上。

3.控制传输方式：处理主机的USB设备的数据传输，包括设备控制指令、设备状态查询及确认命令。当USB设备收到这些数据和命令后将按照先进先出的原则按队列方式处理到达的数据。

4.批量传输方式：用来传输要求正确无误的数据。通常打印机、扫描仪和数码相机以这种方式与主机连接。除等时传输方式外，其他3种方式在数据传输发生错误时，都会试图重新发送数据以保证其准确性。

- SCSI接口技术:

（1）小型计算机系统接口（Small Computer System Interface）：一个SCSI I/O设备控制器，可将新型高速I/O设备增加到计算机系统中。

（2）SCSI 设备控制器的智能化I/O控制：降低了计算机系统的负担，使计算机系统具有更高的I/O能力。

7.缓冲:改善 CPU 和外围设备之间速度不匹配,协调逻辑记录和物理记录大小不一致,减少 IO 操作对 CPU 中断,放款对 CPU 中断响应时间要求

思想:输出缓冲区,将数据送到缓冲区

-单缓冲，简单效率低

-双缓冲，两个缓冲轮流工作

-多缓冲，循环缓存

8.缓存:保留数据拷贝的高速内存.缓冲只保留数据现有的一个现存拷贝而缓存提供一个驻留在其他地方的数据的一个拷贝.

9.即插即用技术 Plug and Play:无需设置.便于设备更换

- PnP技术的特点：
  （1）支持I/O设备及部件的自动配置，使用户能够简单方便地使用系统扩充设备；
  （2）减少由制造商装入的种种用户支持和限制，简化部件的硬件跳接设置，使I/O附加卡和部件不再具有人工跳接线设置电路。
  （3）在主板和附加卡上保存系统资源的配置参数和分配状态，有利于系统对整个I/O资源的分配和控制。
  （4）支持和兼容各种操作系统平台，具有很强的扩展性和可移植性。
  （5）在一定程度上具有“热插入”、“热拼接”技术。

- PnP技术的功能：
  （1）附加卡的识别与确认。
  （2）资源分配。
  （3）附加卡自动配置。
- 多方支持：具有PnP功能的操作系统、配置管理软件、软件安装程序、设备驱动程序等；网络设备的PnP支持；系统平台的支持（如PnP主板、控制芯片组和PnP BIOS等）；各种支持PnP规范总线的I/O控制卡和部件。

10.调度:确定一个好的顺序执行请求,应用程序发出的系统调用顺序不一定最优,调度可以改善系统整体性能,在进程间公平共享设备访问,减少 IO 完成需要的平均等待时间

请求队列:公平

11.设备分配

设备的独立性:作业与设备间独立.作业指定逻辑设备,不指定特定物理设备.设备管理程序负责转换.

系统增减外围设备不影响程序

对独占设备静态分配,共享设备不预先分配

FIFO，优先级

虚拟设备:Spooling 用一类物理设备模拟另一类物理设备,使独立设备变成共享设备,匹配处理器与外围设备的速度

假脱机和设备预约

简答~30 问答~40 设计 30

必考:同步、互斥，临界区，经典同步问题，实现方式，伪代码

必考:死锁，必要条件，预防避免，检测，银行家算法，死锁定理，归约图

必考:内存管理，地址变换，连续内存分配(总结与三个挑战)，局部性原理，创造局部性（存储分布优化），根据局部性去设计

寻道延迟，旋转延迟，内存和硬盘的分析，RAID

文件系统，挂载，共享

设备，输入输出、缓冲、驱动调度

内存一致性/内存模型不考