--最后的胜利(1.2-1.8)

Header

期末考时搜集整合几篇博客+笔记而成，深度的知识区折叠了

删除了部分考试章节中没有的部分，同时偷懒了一点，毕竟目的是考试

由于相比于上一次计算机组成原理时间紧迫得很(只有5天复习)，因此部分内容只能等到日后复盘整理

我想指出的是，这里为了拟合期末复习的知识点作了大量的删节，同时也发现大佬作的总结更加完善

发现自己所学不过是OS的冰山一角，大量的知识点被无情的删除了，仅仅因为期末考不考

呜呼！仅仅为了通过考试的作品完全称不上良好，以下内容仅供参考

- [ ] 占位
- [ ] 填充内容刷洗
- [ ] 转换为背诵版本

1OS引论

基本信息

定义

操作系统是控制和管理计算机硬件和软件资源、合理地组织和管理计算机的工作流程以方便用户使用的程序的集合.

发展目标

1. 方便性
2. 有效性
    - 提高系统资源利用率
    - 提高系统吞吐量
3. 可扩充性
4. 开放性

地位

操作系统为建立用户与计算机之间的接口，为裸机配置的一种系统软件

操作系统是裸机上的第一层软件，位于系统硬件之上，所有其它软件之下（是其它所有软件的共同环境）

历史

- 未配置OS的计算机系统
    - 人工操作方式

      用户独占全机 CPU等待人工操作 严重降低了计算机资源的利用率
    - 脱机输入/输出(Off-Line I/O)方式

      减少了CPU的空闲时间 提高了I/O速度 效率仍然不理想
- 批处理
    1. 单道批处理系统
    2. 多道批处理系统
- 分时
- 实时

s

人工操作方式

手工把纸带装入纸带输入机，启动输入机输入计算机，启动计算机计算，取走结果。用户即使操作员，又是操作员。该操作方式下计算机工作特点 ：

（1） 用户独占全机，资源利用率低

（2） CPU等待人工操作，CPU利用率低

批处理阶段

为了提高计算机的利用率，人们提出了批处理的概念。批处理是指把一批作业按照一定的顺序排好队，计算机按照一定的顺序自动运行，直到运行完所有的作业为止。

- 单道批处理系统（Simple Batch Processing）

  > 也有资料称之为脱机批处理，脱机输入/输出

  把系统的输入/输出工作交给一台外围计算机。用户把程序和数据输入到外围计算机，外围计算机把输入的数据和程序转存到磁带上，然后把磁带送到主机，主机把程序和数据装入内存，执行程序，把结果送到磁带上，再由外围计算机把磁带上的结果输出到打印机上。

  提高了CPU利用率和输入输出的速度。但是，用户仍然需要等待作业完成才能获得结果，CPU仍然需要等待人工操作。

- 多道批处理系统（Multi-Programming Batch Processing）

  多道程序设计技术的出现给计算机系统的管理带来了巨大挑战，如：内存共享与保护问题，处理器调度问题，I/O设备的共享及竞争问题等，为解决这些问题，形成了现代操作系统的雏形。所以说多道程序设计技术是操作系统形成的标志。

  多道程序在内存中同时驻留，它们共享CPU和其他资源。当一个作业需要等待某事件发生时，CPU可以立刻切换到另一个作业上去执行，从而提高了CPU的利用率。

  但是，多道程序环境下，程序的执行是异步的，程序的执行速度不可预知，因此，需要采用某种措施来保证程序的执行顺序。

  为了保证程序的执行顺序，需要引入作业调度和内存管理两个概念。

分时操作系统（Time-Sharing System）

分时操作系统是一种允许多个用户通过终端同时连接到一台计算机上的操作系统。分时系统的基本特征是：多路性、独立性、及时性、交互性。

分时系统的基本原理是：把CPU的时间分成很多个时间片，每个时间片分配给一个用户，用户在这个时间片内独占CPU。由于CPU的时间片很短，用户感觉好像计算机在同一时间为他服务一样，这就是分时系统的基本原理。

分时系统的出现，使得用户不再需要等待作业完成才能获得结果，而是可以实时地与计算机交互。

分时系统不能优先处理一些紧急任务，因此引入了实时操作系统。

实时操作系统（Real-Time System）

在限定时间内完成某些紧急任务，而不需要时间片排队。

实时操作系统分为硬实时操作系统和软实时操作系统。

- 硬实时操作系统：某个动作必须绝对地在规定的时间内完成，否则就会产生严重的后果。例如：导弹控制系统、核反应堆控制系统等。
- 软实时操作系统：某个动作必须在规定的时间内完成，但是如果不能完成，后果并不严重。例如：飞机订票系统，银行管理系统等。

特性原理

特征

并发和共享是多用户(多任务)OS的两个最基本的特征。它们又是互为存在的条件

- 并发
  - 区别并行和并发

    并发是进程宏观一起运行，微观上交替运行，而并行是指同时运行
- 共享
  - 1.互斥共享方式（互斥共享的资源称为临界资源）
  - 2.同时访问方式（一段时间内允许多个进程同时访问）
- 虚拟

> 把物理实体变为若干个逻辑的对应
>
> 两种虚拟技术
>
> - 时分复用技术
> - 空分复用技术

- 异步

> 进程的执行走走停停，以不可预知的速度向前推进

运行机制

另见CPU

中断

广义中断 = 异常(内部) + 狭义中断(IO/外部)

- 外中断

> 由 CPU 执行指令以外的事件引起，如 I/O 完成中断，表示设备输入/输出处理已经完成，处理器能够发送下一个输入/输出请求。此外还有时钟中断、控制台中断等。

- 异常

> 由 CPU 执行指令的内部事件引起，如非法操作码、地址越界、算术溢出等。

系统调用

从用户态转换到核心态，这是由硬件完成的

输入输出必然在核心态执行

导致用户从用户态切到内核态的操作有：某个东西导致了异常中断、I/O等等

在用户程序中使用系统调用。如果一个进程在用户态需要使用内核态的功能，就进行系统调用从而陷入内核，由操作系统代为完成。

系统调用是指用户自编程序通过操作系统提供的接口，向操作系统发出服务请求，从而获得操作系统的服务。是用户态到核心态的唯一入口。

体系结构模型

- 微内核（micro kernel）

  基本思想：内核尽可能小，只实现操作系统的基本功能；将更多的操作系统功能放在核心之外，作为独立的服务进程运行。

  其特点是：内核简单，安全可靠，可移植性好。
- 单内核（monolithic kernel）

  基本思想：将核心分为若干个模块，模块间的通信通过调用其它模块中的函数来实现。

  其特点为：执行效率高，但可移植性相对减弱。

2硬件

CPU

构成

- 运算器
- 控制器
- 寄存器

  重要寄存器

- PC
- IR
- PSW

状态

- 特权指令：只能由操作系统内核程序运行的指令，例如：关中断、修改时钟中断向量等。
- 非特权指令：用户自编程序可以运行的指令，例如：加法、减法、乘法、除法等。

在具体实现上，根据程序的不同，CPU所处状态分为核心态和用户态。

- 核心态：又称为管态、内核态。CPU执行操作系统内核程序时所处的状态，此时CPU可以运行特权指令。
- 用户态：又称为目态。CPU执行用户自编程序时所处的状态，此时CPU不可以运行特权指令。

在CPU中设置程序状态寄存器（PSW）来标志处理器当前处于的状态。

CPU处于核心态可以调用除了访管指令以外的所有指令

> 从用户态转到核心态会用到访管指令. 访管指令在用户态使用，所以它不可能是特权指令，也不可能是管态(核心态)使用的

功能

取指令然后执行

取值周期 <-> 执行周期

存储器体系结构

程序的存储器访问局部性原理　　能提高存储器体系效能的关键

金字塔结构

- 寄存器
  - Cache
    - 内存
      - 外存(硬盘)
        - 磁带、光盘...

3进程

程序执行特征

程序并发执行

- 程序的并发执行
- 程序并发执行时的特征
  - 间断性
  - 失去封闭性
  - 不可再现性

定义

进程是进程实体的运行过程

在系统中能独立运行并作为资源分配的基本单位

分类

从操作系统角度

- 系统进程
- 用户进程

组成

PCB	**Process Control Block，进程控制块。包含着进程的描述和控制信息，是进程存在的唯一标志**
程序	“纯代码”部分，也称为“文本段”，描述了进程要完成的功能，是进程执行时不可修改的部分
数据	进程执行时用到的数据（全局变量，静态变量，常量）
工作区	即堆栈区，用于内存的动态分配，保存局部变量，传递参数、函数调用地址等

和程序区别

进程是程序的一次执行(动态)

程序是完成任务功能的指令的有序序列(静态)

基本状态

进程的三种基本状态

- 就绪状态ready
- 执行状态running
- 阻塞状态block

状态转换

- 就绪状态 -> 运行状态：分配到CPU
- 运行状态 -> 就绪状态：时间片用完，或被更高优先级的进程抢占CPU
- 运行状态 -> 阻塞状态：等待某事件发生，例如需要分配资源，申请I/O操作，但系统暂时不能进行分配
- 阻塞状态 -> 就绪状态：等待的事件发生

> 阻塞态->运行态和就绪态->阻塞态这二种状态转换不可能发生

进程控制块 PCB

包含信息

1. 进程标识符
2. 处理机状态
3. 进程调度
4. 进程控制

组织方式

1. 链接方式

   相同状态组织成一个队列

2. 索引方式

   建立索引表，记录PCB地址

控制

系统对进程的控制通过操作系统内核中的原语实现

原语

由若干条指令构成的可完成特定功能的程序段，必须在管态下运行，它是一个“原子操作(atomic operation)”过程，执行过程不能被中断

进程控制的基本原语

- 创建进程：创建一个新进程，为其分配资源，初始化PCB，将其插入就绪队列
- 撤销进程：终止一个进程，释放其占有的资源，回收其PCB
- 阻塞进程：将一个进程由运行状态转换为阻塞状态
- 唤醒进程：将一个进程由阻塞状态转换为就绪状态
- 挂起进程：内存不足时使用
- 激活进程：解除挂起状态

层次结构

- 父进程
- 子进程

终止

引起进程终止的事件

1. 正常结束
2. 异常结束
3. 外界干预

调度

调度层级

- 高级(作业调度)
  - 给作业获得竞争处理机的机会
- 低级(进程调度)
  - 从就绪队列选进程分配处理机
- 中级(内存调度)
  - 提高内存利用率和系统吞吐量

调度方式

- 非剥夺方式
- 剥夺方式

调度准则

(略)用于比较的特征准则有

- CPU使用率
- 吞吐量：一个单位时间内完成进程的数量
- 周转时间：从进程提交到进程完成的时间称为周转时间
- 平均周转时间：多个作业周转时间平均值
- 带权周转时间：周转时间/运行时间
- 平均带权周转时间：多个作业带权周转时间的平均值

- 等待时间：为最就绪队列中等待所花的时间，调度算法不影响进程运行时间，只影响在队列中的等待时间
- 响应时间：从提交请求到响应请求的时间

调度算法

- 先来先服务
  - 按照进程到达就绪队列的先后顺序进行调度，是一种非抢占式调度算法
- 时间片轮转法
  - 系统规定一个时间长度(时间片)作为允许一个进程运行的时间，如果在这段时间该进程没有执行完，则必须让出CPU给下一个进程使用，自己则排到就绪队列末尾，等待下一次调度；如果时间片结束之前被阻塞或结束，则CPU立即切换
- 基于优先调度算法
  - 为系统中的每个进程规定一个优先数，就绪队列中具有最高优先数的进程有优先获得处理机的权利；如果几个进程的优先数相同，可则对它们实行RR调度策略
    - 静态优先数法：进程的优先数在进程创建时就确定下来，不再改变
    - 动态优先数法：进程的优先数随着时间的推移而改变
- 多级反馈队列调度算法（Multilevel Feedback Queue Scheduling）：
  - 系统中维持多个不同优先级的就绪队列。每个就绪队列具有不同长度的时间片。优先级高的就绪队列里的进程，获得的时间片短；优先级低的就绪队列里的进程，获得的时间片长。新进程进入时加入优先级最高的就绪队列的末尾

线程

引入　简化线程间的通信，以小的开销来提高进程内的并发程度(轻装运行)

与进的区别联系

- 二者都用于实现程序的并发执行

- 进程是操作系统资源分配的基本单位，而线程是处理器任务调度和执行的基本单位
- 进程有自己独立的地址空间和资源；线程共享所属进程的地址空间和资源，但有自己的栈和寄存器，线程之间的切换和通信开销较小
- 进程的并发性较差，一个进程崩溃可能导致整个系统崩溃；线程的并发性较好，一个线程崩溃只会影响该线程所属的进程
- 进程可以拥有多个线程，一个进程至少有一个线程；线程不能拥有其他线程，但可以创建和销毁其他线程

实现机制

分类

- 用户级线程
- 内核支持线程

两者比较

- 线程调度和切换速度
- 系统调用
- 执行时间

4进程同步

基本概念

进程间的相互作用

- 同步：需要相互合作，协同工作的进程间的关系
- 互斥：多个进程因争用临界资源而互斥执行
  - 临界资源：一段时间内只允许一个进程访问的资源

信号量和P、V操作

- 信号量(semapore)：一个整形变量，通过初始化以及P、V操作来访问
  - 公有信号量：用于进程间的互斥，初值通常为1
  - 私有信号量：用于进程间的同步，初值通常为0或n
- P操作：荷兰语中的"proberen"，意为"测试"，用于申请临界资源
  - 若申请成功，信号量 $s = s - 1$ ，进程继续执行
  - 若申请失败，进程进入阻塞状态，进入S的等待队列等待信号量变为非0
- V操作：荷兰语“verhogen”—“增量/升高”之意，意味着释放/增加一个单位资源

P、V操作通常被设置为原语操作，不可分割，不可中断

管程

管程(monitor)是关于共享资源的一组数据结构和在这组数据结构上的一组相关操作。管程将共享变量以及对共享变量能够进行的所有操作集中在一个模块中，以过程调用的形式提供给进程使用。

同步机制规则

- 空闲让进
- 忙则等待
- 有限等待
- 让权等待

进程同步机制

信号量机制

- 整型信号量

- 记录型信号量
  - 由于整型信号量没有遵循让权等待原则，记录型允许负数，即阻塞链表
- AND型信号量
- 信号量集
  - 理解:AND型号量的wait和signal仅能对信号施以加1或减1操作，意味着每次只能对某类临界资源进行一个单位的申请或释放。当一次需要N个单位时，便要进行N次wait操作，这显然是低效的，甚至会增加死锁的概率。此外，在有些情况下，为确保系统的安全性，当所申请的资源数量低于某一下限值时，还必须进行管制，不予以分配。因此，当进程申请某类临界资源时，在每次分配前，都必须测试资源数量，判断是否大于可分配的下限值，决定是否予以分配
  - 操作
    - Swait(S1, t1, d1…Sn, tn, dn)
    - Ssignal(S1, d1…Sn, dn)
  - 特殊情况

经典进程同步问题

- 生产者-消费者问题
- 读者-写者问题
- 哲学家进餐问题
- 嗜睡的理发师问题

进程同步问题例子

典型问题：生产者-消费者问题

这是一个非常重要而典型的问题，进程同步60%以上的题目都是生产者消费者的改编

问题描述：有一群生产者进程在生产产品，并将这些产品提供给消费者进程去进行消费。使用一个缓冲区来保存物品，只有缓冲区没有满，生产者才可以放入物品；只有缓冲区不为空，消费者才可以拿走物品。

因为缓冲区属于临界资源，因此需要使用一个互斥量 mutex 来控制对缓冲区的互斥访问。（缓冲区用循环队列就可以模拟）

为了同步生产者和消费者的行为，需要记录缓冲区中物品的数量。数量可以使用信号量来进行统计，这里需要使用两个信号量：`empty` 记录空缓冲区的数量，`full` 记录满缓冲区的数量。其中，`empty` 信号量是在生产者进程中使用，当 `empty` 不为 `0` 时，生产者才可以放入物品；`full` 信号量是在消费者进程中使用，当 `full` 信号量不为 `0` 时，消费者才可以取走物品。

注意，不能先对缓冲区进行加锁，再测试信号量。也就是说，不能先执行 `down(mutex)` 再执行 `down(empty)`。如果这么做了，那么可能会出现这种情况：生产者对缓冲区加锁后，执行 `down(empty)` 操作，发现 `empty = 0`，此时生产者睡眠。消费者不能进入临界区，因为生产者对缓冲区加锁了，消费者就无法执行 `up(empty)` 操作，`empty` 永远都为 `0`，导致生产者永远等待下，不会释放锁，消费者因此也会永远等待下去。

顺序执行的时候显然没有任何问题，然而在并发执行的时候，就会出现差错，比如共享变量counter，会出现冲突。解决的关键是将counter作为临界资源来处理。

```
##define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer() {
    while(TRUE) {
        int item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer() {
    while(TRUE) {
        down(&full);
        down(&mutex);
        int item = remove_item();
        consume_item(item);
        up(&mutex);
        up(&empty);
    }
}
```

实际例子：吃水果问题

桌子上有一个盘子，可以放一个水果。爸爸每次放一个苹果，妈妈每次放一个桔子；女儿每次吃一个苹果，儿子每次吃一个桔子。

```
semaphore S =  1, SA = 0, SO = 0; // 盘子的互斥信号量、苹果和桔子的互斥信号量

father(){
  while(1){
    have an apple;
    P(S);
    put an apple;
    V(SA);
  }
}

mother(){
  while(1){
    have an orange;
    P(S);
    put an orange;
    V(SO);
  }
}

daughter(){
  while(1){
    P(SA);
    get an apple;
    V(S);
    eat an apple;
  }
}

son(){
  while(1){
    P(SO);
    get an orange;
    V(S); // 吃完之后要V(S)才能接着让爸爸妈妈接着放
    eat an orange;
  }
}
```

典型问题：哲学家就餐问题

这是由Dijkstra提出的典型进程同步问题

5个哲学家坐在桌子边，桌子上有5个碗和5支筷子，哲学家开始思考，如果饥饿了，就拿起两边筷子进餐（两支筷子都拿起才能进餐），用餐后放下筷子，继续思考

多个临界资源的问题

只考虑筷子互斥：可能会产生死锁，比如所有人都同时拿起右边筷子，左边无限等待
再考虑吃饭行为互斥：同时只让一个人吃饭，肯定不会冲突或死锁，但是资源比较浪费

解决方法1：允许4位哲学家同时去拿左边的筷子

```
semaphore chopstick[5] = {1, 1, 1, 1, 1}; // 筷子信号量
semaphore eating = 4; // 允许四个哲学家可以同时拿筷子

void philosopher(int i){ // 第i个哲学家的程序
  thinking();
  P(eating);
  P(chopstick[i]); // 请求左边的筷子
  P(chopstick[(i+1)%5]); // 请求右边的筷子
  eating();
  V(chopstick[(i+1)%5]);
  V(chopstick[i]);
  V(eating);
}
```

解决方式2：奇数位置的哲学家先左后右，偶数位置的哲学家先右后左

典型问题：读者写者问题

读进程：Reader进程
写进程：Writer进程

允许多个进程同时读一个共享文件，因为读不会使数据混乱；但同时仅仅允许一个写者在写

写-写互斥，写-读互斥，读-读允许

纯互斥问题，没有同步关系没有先后之分

需要多加一个读者计数器，并且修改这个时要同步，所以要再加一个变量保证修改count时的互斥

```
semaphore rmutex = 1, wmutex = 1; // readcount的互斥信号量，数据的互斥信号量
int readcount= 0;

Reader(){ // 每次进入和退出因为涉及readcount变化，要保证同时只有一个，所以分别设置rmutex
  while(1){
    P(rmutex); // 抢readcount信号量，防止多个reader同时进入 导致readcount变化不同
    if(readcount==0){
      P(wmutex); // 第一个进来的读者，抢公共缓冲区
    }
    readcount += 1;
    V(rmutex); // 其他reader可以进来了

    perform read operation;

    P(rmutex); // 再抢一次，使每次只有一个退出
    readcount -= 1;
    if(readcount==0){
      V(wmutex); // 最后一个reader走了，释放公共缓冲区
```

```
    }
    V(rmutex);
  }
}

Writer(){ // 写者很简单，只需要考虑wmutex公共缓冲区
  while(1){
    P(wmutex);
    perform write operation;
    V(wmutex);
  }
}
```

上面的算法可能会导致写者可能会被插队，如果是RWR，中间的W被堵了，结果后面的R还能进去，W还要等后面的R读完

变形：让写者优先，如果有写者，那么后来的读者都要阻塞，实现完全按照来的顺序进行读写操作。
解决方法：再增加一个wfirst信号量，初始为1，在读者的Read()和之前的阶段和写者部分都加一个P(wfirst)作为互斥入口，结尾V(wfirst)释放即可

变形：让写者真正优先，可以插队
解决方法：增加一个writecount统计，也就是加一个写者队列，写者可以霸占这个队列一直堵着

例子：汽车过窄桥问题

来往各有两条路，但是中间有一个窄桥只能容纳一辆车通过，方向一样的车可以一起过

```
semaphore mutex1=1, mutex2=1, bridge=1;
int count1=count2=1;

Process North(i){
  while(1){
    P(mutex1);
    if(count1==0){
      P(bridge); // 第一辆车来了就抢bridge让对面的车进不来
    }
    count1++;
    V(mutex1);

    cross the bridge;

    P(mutex1);
    count1--;
    if(count1==0){
      V(bridge); // 最后一辆车走了就释放bridge让对面的车可以进来
    }
    V(mutex1);
```

```
  }
}

Process South(i){
  while(1){
    P(mutex2);
    if(count2==0){
      P(bridge);
    }
    count2++;
    V(mutex2);

    cross the bridge;

    P(mutex2);
    count2--;
    if(count2==0){
      V(bridge);
    }
    V(mutex2);
  }
}
```

进程通信

类型

- 共享存储器系统
  - 基于共享数据结构
  - 基于共享存储区
- 消息传递系统
  - 直接通信
  - 间接通信
- 管道通信系统

死锁

如果一组进程中的每一个进程都在等待仅由该进程中的其他进程才能引发的事件，那么该组进程是死锁的

原因

1. 竞争资源
    - 可剥夺和非剥夺性资源
    - 竞争非剥夺性资源
    - 竞争临时性资源
2. 进程推进顺序非法

条件

原因之上更近一步

- 互斥
- 请求和保持
- 不可抢占（不剥夺）
- 循环等待（环路等待）

解决

- 预防

- 避免
- 检测
- 解除

预防

静态方法，在进程执行前采取的措施，通过设置某些限制条件，去破坏产生死锁的四个条件之一，防止发生死锁

- 破坏"请求和保存"条件

  所有进程在开始运行之前，必须一次性地申请其在整个运行过程中所需的全部资源

  优点

  简单，易行，安全

  缺点

  - 资源被严重浪费，严重地恶化了资源的利用率
  - 使进程经常会发生饥饿现象

- 破坏"不可抢占"条件

  当一个已经保存了某些不可被抢占资源的进程，提出新的资源请求而不能得到满足时，它必须释放已经保持的所有资源，待以后需要时再重新申请

- 破坏"循环等待"条件

  对系统所以资源类型进行线性排序，并赋予不同的序号。所有进程对资源的请求必须严格按资源序号递增的次序提出

避免

动态的方法，在进程执行过程中采取的措施，不需事先采取限制措施破坏产生死锁的必要条件，而是在进程申请资源时用某种方法去防止系统进入不安全状态，从而避免发生死锁

- 系统安全状态

  某时刻，对于并发执行的n个进程，若系统能够按照某种顺序如<p1,p2…pn>来为每个进程分配所需资源，直至最大需求，从而使每个进程都可顺利完成，则认为该时刻系统处于安全状态，这样的序列为安全序列

  不按照安全顺序进行分配资源，则可能发生由安全状态向不安全状态的转换
- 不安全状态：不存在一种顺序使得每个进程都能够顺利完成。不安全状态不一定发生死锁，但死锁一定是不安全状态
- 银行家算法

检测

- 资源分配图
  - 简化步骤
    - 选择一个没有阻塞的进程p
    - 将p移走，包括它的所有请求边和分配边
    - 重复步骤1，2，直至不能继续下去
- 死锁定理
  - 利用资源分配图

    若一系列简化以后不能使所有的进程节点都成为孤立节点
- 检测算法

资源分配图

1. 圆(椭圆)表示一个进程；

2. 方块表示一个资源类，其中的圆点表示该类型资源中的单个资源；
3. 从资源指向进程的箭头表示资源被分配给了这个进程；
4. 从进程指向资源的箭头表示进程申请一个这类资源；

临时性资源，即可消耗的资源。如信号，邮件等。特点是没有固定数量，不需要释放

含临时性资源的资源分配图规则：

1. 圆表示一个进程；
2. 方块表示一个资源类，其中的圆点表示该类型资源中的单个资源；
3. 由进程指向资源的箭头表示该进程申请这种资源，一个箭头只表示申请一个资源；
4. 由资源类指向进程的箭头表示该进程产生这种资源，一个箭头可表示产生一到多个资源，每个资源类至少有一个生产者进程。

检测

1. 检查有无环路，如果没有环路，则系统一定不会发生死锁；
2. 如果有环路，则检查环路上的资源类是否都只有一个资源，如果是，则系统一定会发生死锁；
3. 如果环路上的资源类都有多个资源，查找即非阻塞又非独立的进程，消去与之有关的所有有向边
4. 如果最终能够消去所有的有向边，则系统不会发生死锁，否则系统会发生死锁

死锁定理：如果一个系统状态为死锁状态，当且仅当资源分配图是不可完全化简。也即，如果资源图中所有的进程都成为孤立结点，则系统不会死锁；否则系统状态为死锁状态

解除

- 抢占资源
- 终止（（或撤销））进程

5存储

定义

目的

方便用户　　将逻辑和物理地址分开

常规存储管理方式的特征

- 一次性
- 驻留性

程序操作

步骤

1. 编译

   源程序 ->目标模块（Object modules）--------Compiler

   由编译程序对用户源程序进行编译，形成若干个目标模块

2. 链接

   一组目标模块 ->装入模块 （Load Module）----------Linker

   由链接程序将编译后形成的一组目标模板以及它们所需要的库函数链接在一起，形成一个完整的装入模块

3. 装入

   装入模块 ->内存 --------Loader

   由装入程序将装入模块装入内存

链接

- 静态链接方式
- 装入时动态链接
- 运行时动态链接

装入

- 绝对装入方式
- 可重定位装入方式

  一个程序通常需要占用连续的内存空间，程序装入内存后不能移动。不易实现共享
- 动态运行时的装入方式

  装入模块装入内存后，并不立即把装入模块中的逻辑地址转换为物理地址，而是把这种地址转换推迟到程序真正要执行时才进行

  可以将一个程序分散存放于不连续的内存空间，可以移动程序，有利用实现共享。

地址绑定

（略）

将指令和数据绑定到内存地址有三种情况：

- 编译时：生成绝对代码，编译时就知道了进程在内存中的驻留地址，所生成的编译代码就可以从该位置往后扩展（开始位置发生变化就要重新编译）
- 加载时：生成可重定位代码
- 执行时：绑定延迟到执行时才进行

- 逻辑地址（相对地址，虚拟地址）：用户程序经过编译、汇编后形成目标代码，目标代码通常采用相对地址的形式，其首地址为`0`，其余地址都相对于首地址而编址
- 物理地址（绝对地址，实地址）：内存中储存单元的地址，可直接寻址

编译和加载时的地址绑定方法生成相同逻辑地址和物理地址，而执行时的地址绑定方案生成不同的逻辑和物理地址，这种情况通常称逻辑地址为虚拟地址

运行时从虚拟地址到物理地址的映射由**内存管理单元**来完成

管理方式

碎片

- 内碎片：占用分区之内未被利用的空间
- 外碎片：占用分区之间难以利用的空闲分区（通常是小空闲分区）

连续分配

- 单一连续分配

  （最简单）
- 分区分配
  - 固定分区
  - 可变分区

可变分区

指在作业装入内存时，从可用的内存中划出一块连续的区域分配给它，且分区大小正好等于该作业的大小。可变式分区中分区的大小和分区的个数都是可变的，而且是根据作业的大小和多少动态地划分

分区分配DS

- 已分分区表 + 空闲分区表
- 空闲分区链

分区分配AL

算法	思想	空闲分区	优点	缺点
最先适应算法	从前往后找到第一个满足要求的空闲分区	地址递增排序	简单，快速	产生大量无法利用的小碎片
邻近适应算法	从上次分配的空闲分区开始找，直到找到第一个满足要求的空闲分区	地址递增排序（循环）	简单，快速	产生大量无法利用的小碎片
最佳适应算法	从所有满足要求的空闲分区中找到最小的一个	容量递增排序	大分区得以保留	产生小碎片。开销大，回收分区要重新对空闲分区排队
最坏适应算法	从所有满足要求的空闲分区中找到最大的一个	容量递减排序	减少小碎片	大分区不能保留。开销大，回收分区要重新对空闲分区排队

- 首次适应算法（first fit,FF）
  - 顺序找，找到一个满足的就分配，但是可能存在浪费
  - 这种方法目的在于减少查找时间
  - 空闲分区表（空闲区链）中的空闲分区要按地址由低到高进行排序
- 循环首次适应算法（next fit，NF）
  - 相对上面那种，不是顺序，类似哈希算法中左右交叉排序
  - 空闲分区分布得更均匀，查找开销小
  - 从上次找到的空闲区的下一个空闲区开始查找，直到找到第一个能满足要求的的空闲区为止，并从中划出一块与请求大小相等的内存空间分配给作业。
- 最佳适应算法（best fit，BF）
  - 找到最合适的，但是大区域的访问次数减少
  - 这种方法能使外碎片尽量小。
  - 空闲分区表（空闲区链）中的空闲分区要按大小从小到大进行排序，自表头开始查找到第一个满足要求的自由分区分配。
- 最差适应算法（worst fit，WF）
  - 相对于最好而言，找最大的区域下手，导致最大的区域可能很少，也造成许多碎片
  - 空闲分区按大小由大到小排序

分配操作

回收操作

几种类型

优缺点

紧凑

离散分配

（非连续分配）

将程序分为若干块，分别放在不同的空闲区中，从而使得存储空间的利用率得到提升。

各种方式的访问内存次数

- 基本两个　　2次

  > 第一次从内存中找到页表，然后找到页面的物理块号，加上页内偏移得到实际物理地址；第二次根据第一次得到的物理地址访问内存取出数据

- 基本两个加快表命中　　1次
- 多级基本两个　　多一级加一次
- 多级基本两个加快表　　多一级加一次，多一个快表命中减一次
- 段页式　　3次
- 段页式加快表　　3次，多一次TLB命中减一次

基本分页

方便系统内存利用率提升

概念

- 页面
  - 将一个进程的逻辑地址空间分成若干个大小相等的片
- 页框（frame）
  - 内存空间分成与页面相同大小的存储块
- 地址结构
  - 页号P + 位移量W(0-31)
- 页表
  - 在分页系统中，允许将进程的各个页离散地存储在内存在内存的任一物理块中，为保证进程仍然能够正确地运行，即能在内存中找到每一个页面所对应的物理块，系统又为每个进程建立了一张页面映像表，简称页表
  - 页表的作用是实现从页面号到物理块号的地址映射
- 地址变换机构

  页表寄存器PTR(Page-Table Register)　　存放页表在内存的始址-页表长度

  工作原理　　页号(判断地址越界)->查PTR->获得页表信息查页表(在内存中)->转换为块号->拼接块号和块内地址(拼)->找到内存块

采用分页技术不会产生外部碎片，但是会有内部碎片，产生的原因是进程所需内存不是页的整数倍，最后一页可能装不满最后一个帧（物理块），导致产生页内碎片

分页的一个重要特点是用户视角的内存和实际物理内存的分离，用户看到的是一整块内存用于一个进程，但实际的物理内存则是分散的，逻辑地址到物理地址的映射用户不知道。

用户进程不能访问该进程非占用的内存，即无法访问页表规定之外的内存

+快表

有快表情况下地址变换过程为

1. CPU给出有效地址
2. 地址变换机构自动地将页号送入高速缓存，确定所需要的页是否在快表中。
3. 若是，则直接读出该页所对应的物理块号，送入物理地址寄存器；
4. 若快表中未找到对应的页表项，则需再访问内存中的页表
5. 找到后，把从页表中读出的页表项存入快表中的一个寄存器单元中，以取代一个旧的页表项。

多级页表

将页表进行分页，每个页面的大小与内存物理块的大小相同，并为它们进行编号，可以离散地将各个页面分别存放在不同的物理块中

两级页表的分页系统访问一次需要三次访存

基本分段

方便用户

在分段存储管理方式中，作业的地址空间被划分为若干个段，每个段是一组完整的逻辑信息，每个段都有自己的名字，都是从零开始编址的一段连续的地址空间，各段长度是不等的。

内存空间被动态的划分为若干个长度不相同的区域，称为物理段，每个物理段由起始地址和长度确定

概念

格式　段号+段内地址

- 段表

  段表实现了从逻辑段到物理内存区的映射。
- 地址变换机构

  段表寄存器(+)

分段分页区别

> 分页是系统管理的需要，分段是用户应用的需要。一条指令或一个操作数可能会跨越两个页的分界处，而不会跨越两个段的分界处

- 页是信息的物理单位　　段是信息的逻辑单位
- 页的大小固定由系统决定　　段的大小不固定由用户程序决定
- **从用户角度看 分页的地址空间是一维的 + 透明的　分段的地址空间是二维的 + 可见的**

段页式

用户程序按段划分，物理内存按页划分，即以页为单位进行分配

基本原理

段号S	页号P	页内地址W

- 段表记录了每一页的页表的起始地址和页表长度
- 页表记录了每一段所对应的逻辑页号与内存块号的对应关系，每一段有一个页表

通过三次访问取指令或数据

1. 访问内存中的段表，获得页表地址
2. 访问内存中的页表，获得该页所在的物理块号
3. 真正根据所得的物理地址取出指令或者数据

虚存

- 分页虚拟
- 分段虚拟

6虚拟存储器

概念

局部性原理

（核心）

程序在执行时将呈现出局部性特征，即在一较短的时间内，程序的执行仅局限于某个部分，相应地，它所访问的存储空间也局限于某个区域

- 时间局限性
- 空间局限性

基本原理

- 在程序装入时不必将其全部读入内存，而只需要将当前需要执行的部分页或段读入内存就可以开始执行
- 在程序执行过程中，如果需要用到的指令或者数据不在内存中，则通知操作系统把需要的页或段调入内存，继续执行
- 操作系统将内存中暂时不使用的页或段调出保存在外存上，从而腾出空间存放将要装入的程序以及将要调入的页或段–具有请求调入和置换功能，只需程序的一部分在内存就可执行

定义

具有请求调入功能和置换功能，能从逻辑上对内存容量加以扩充的一种存储器系统

优点

- 大程序：可在较小的可用内存中执行较大的用户程序；
- 大的用户空间：提供给用户可用的虚拟内存空间通常大于物理内存(`real memory`)
- 并发：可在内存中容纳更多程序并发执行；
- 易于开发：不必影响编程时的程序结构
- 以CPU时间和外存空间换取昂贵内存空间，这是操作系统中的资源转换技术

实现

- 虚拟页式（请求分页）
- 虚拟段式（请求分段）
- 虚拟段页式（请求分段页）

特征

- 离散性
  - 指在内存分配时采用离散的分配方式，它是虚拟存储器的实现的基础
- 多次性
  - 指一个作业被分成多次调入内存运行，即在作业运行时没有必要将其全部装入，只须将当前要运行的那部分程序和数据装入内存即可。多次性是虚拟存储器最重要的特征
- 对换性
  - 指允许在作业的运行过程中在内存和外存的对换区之间换进、换出。
- 虚拟性
  - 指能够从逻辑上扩充内存容量，使用户所看到的内存容量远大于实际内存容量。

分页虚拟存储

硬件机构

- 缺页中断机构
- 地址变换机构

页面置换算法

- 最佳置换算法(不能实现)
- FIFO

- 最近最久未使用（LRU）
  - 寄存器支持
  - 特殊的栈结构
- 简单clock
  - 对访问位A的判断
- 改进clock
  - 增加对修改位M思维判断

LRU

每个页面都有两个状态位：R 与 M，当页面被访问时设置页面的 R=1，当页面被修改时设置 M=1。其中 R 位会定时被清零。可以将页面分成以下四类：

- R=0, M=0
- R=0, M=1
- R=1, M=0
- R=1, M=1

当发生缺页中断时，NRU 算法随机地从类编号最小的非空类中挑选一个页面将它换出。

NRU 优先换出已经被修改的脏页面（R=0，M=1），而不是被频繁使用的干净页面（R=1，M=0）。

内存分配策略

最小物理块数　　即能保证进程正常运行所需的最小物理块数

物理块分配策略

- 固定分配局部置换
- 可变分配全局置换
- 可变分配局部置换

缺页中断

在请求分页系统中，每当所要访问的页面不在内存时，就会产生缺页中断

调页策略

- 请求调页
- 预调页

抖动问题

即刚被换出的页很快又要被访问，需要将它重新调入，此时又需要再选一页调出

原因　　进程太多了

预防

- 局部置换策略
- 工作集算法
- L=S准则
- 挂起一些进程

分段虚拟存储

简单

- 共享段
  - 可重入代码/纯代码 – 允许多个进程同时访问

7用户接口

简单

概念

OS接口

- 命令接口
  - 脱机命令
  - 交互式命令
- 程序接口
- 图形接口

系统调用

类似中断处理

略

8文件

存储介质

- 顺序存储设备　　磁带
- 直接存储设备　　磁盘

结构

逻辑结构

- 纪录式(有结构)文件
- 流式(无结构)文件

物理结构

（讲直接存储设备）

- 连续文件
- 连接文件
  - 隐式连接
  - 显式连接
    - FAT
- 索引文件
- 直接文件
- 物理文件 of NTFS文件系统

文件目录

文件控制块（FCB）

- 文件名+inode(属性)

简单的文件目录

- 单级文件目录
  - 查找慢
  - 不允许重名
  - 不便于实现文件共享
- (二)多级文件目录
  - 提高检索速度，从M*N到M+N

FCB

- FCB的有序集合称为文件目录，一个FCB就是一个文件目录项
- 逻辑结构的组织形式取决于用户，物理结构的组织形式取决于储存介质特性
- 文件目录项（FCB）不包括FCB的物理位置

改进

索引节点

$(M+1)/2 > (N+1)/2 + 1$ 才可以改进(前的平均查找次数大于改进后+1)

外存空间管理

- 空闲块表法
- 空闲链表法
  - 空闲盘块链
  - 空闲盘区链
- 位示图法
- 成组连接法

文件使用

文件共享

- 符号链
- 索引节点

安全性

影响因素

- 人为

- 系统
- 自然

防止系统因素

坏块管理

磁盘容错

1. 第一级容错技术
    - 双份目录 + 双份文件分配表
    - 热修复重定向 + 写后读校验
2. 第二级容错技术
    - 磁盘镜像
    - 磁盘双工
3. 廉价磁盘冗余阵列
    - RAID 0~6级

备份

一致性

略

磁盘调度

IO时间

- 查找(可改进)

- 等待
- 传输

移臂调度

- FCFS　先来先服务
  - 优点：公平，简单
  - 缺点：可能导致某些进程的请求长期得不到满足
- SSTF　最短寻道时间优先
- SCAN　扫描算法/电梯调度
  - 扫描算法不仅考虑到欲访问的磁道与当前磁道间的距离，更优先考虑的是磁道当前的移动方向
  - 可防止低优先级进程出现“饥饿”的现象
- CSCAN　循环扫描
  - 规定磁头单向移动，这里不算跳的距离
- NStepScan　N步扫描算法
  - 子队列依次处理
- FSCAN
  - 使用子队列简化N步SCAN

9设备

设备控制器

主要功能：控制一个或多个I/O设备，以实现I/O设备和计算机之间的数据交换

基本功能

- 接收和识别命令
- 数据交换
  - 实现CPU与控制器，控制器与设备间的数据交换
- 标识和报告设备的状态
- 地址识别
- 数据缓冲区
- 差错控制

I/O通道

是一种特殊的处理机，具有通过执行通道程序完成I/O操作的指令

特点

- 指令单一(局限于与I/O操作相关的指令)
- 与CPU共享内存

目的

建立独立的I/O操作(组织，管理和结束)，使由CPU处理的I/O工作转由通道完成（解放CPU，实现并行)

基本过程

- CPU向通道发出I/O指令->通道接收指令->从内存取出通道程序处理I/O->向CPU发出中断

瓶颈问题

- 原因是通道不足
- 解决办法：增加设备到主机间的通路，而不增加通道

通道类型

- 字节多路通道
  - 低中速连接子通道时间片轮转方式共享主通道
  - 字节多路通道不适于连接高速设备，这推动了按数组方式进行数据传送的数组选择通道的形成。
- 数组选择通道
  - 这种通道可以连接多台高速设备，但只含有一个分配型子通道，在一段时间内只能执行一道通道程序， 控制一台设备进行数据传送， 直至该设备传送完毕释放该通道。这种通道的利用率很低。
- 数组多路通道
  - 含有多个非分配型子通道，前两种通道的组合，通道利用率较好

I/O控制方式

宗旨

减少CPU对I/O控制的干预，充分利用CPU完成数据处理工作

- 程序查询
- IO中断
- DMA（直接存储器访问）
- IO通道

缓冲

原因

- 缓和CPU与I/O设备间速度不匹配的矛盾
- 减少对CPU的中断频率，放宽对CPU中断响应时间的限制
- 提高CPU和I/O设备之间的并行性

类型

- 单缓冲
- 双缓冲
- 循环缓冲
- 缓冲池

缓冲池

（理解为更大的缓冲区）

组成

- 空缓冲队列（emq）
- 输入队列（inq）
- 输出队列（outq）

Getbuf+Putbuf过程

- 收容：缓冲池接收外界数据
- 提取：外界从缓冲池获得数据

工作方式

- 收容输入

- 提取输入
- 收容输出
- 提取输出

设备驱动程序

主要任务是接受来自它上一层的与设备无关软件的抽象请求，并执行这个请求

功能

- 接收由I/O进程发来的命令和参数， 并将命令中的抽象要求转换为具体要求
- 检查用户I/O请求的合法性，了解I/O设备的状态，传递有关参数，设置设备的工作方式。
- 发出I/O命令，如果设备空闲，便立即启动I/O设备去完成指定的I/O操作；如果设备处于忙碌状态，则将请求者的请求块挂在设备队列上等待。
- 及时响应由控制器或通道发来的中断请求，并根据其中断类型调用相应的中断处理程序进行处理。
- 对于设置有通道的计算机系统，驱动程序还应能够根据用户的I/O请求，自动地构成通道程序。

DMA控制器组成

- 主机与DMA控制器的接口
- DMA控制器与块设备的接口
- I/O控制逻辑

设备分配

数据结构

显然，在有通道的系统中，一个进程只有获得了通道，控制器和所需设备三者之后，才具备了进行I/O操作的物理条件

- 设备控制表DCT

- 控制器控制表COCT
- 通道控制表CHCT
- 系统设备表SDT
- 逻辑设备表LUT

考虑因素

- 设备固有属性
- 设备分配算法
- 设备分配中的安全性

独占设备的分配程序

基本分配程序

- 分配设备
- 分配控制器
- 分配通道

改进的分配程序

- 增加设备独立性
- 考虑多通路情况

虚拟设备

Spooling假脱机系统

是对脱机输入/输出系统的模拟

组成

- 输入/输出井
- 输入/输出缓冲区
- 输入/输出进程

共享打印机

工作流程

- 由输出进程在输出井中为之申请一块空闲盘块区，将要打印的Data存入
- 输出进程为用户进程申请一张空白的用户请求打印表，并将用户的打印要求填入，再讲其挂到请求打印队列上