操作系统期末复习资料

一　操作系统引论

1. 操作系统目标 ：

   有效性、方便性、可扩充性、开放性

2. 操作系统作用：

   为用户和计算机之间提供接口、管理计算机系统资源、实现对计算机资源的抽象

3. 操作系统发展：

   人工操作方式、脱机输入输出方式、单道批处理系统、多道批处理系统、分时系统、实时系统。

   单道批处理系统特点：

   自动性：磁带上的作业能自动逐个依此运行

   顺序性：各道作业是顺序进入内存，顺序完成操作（类似队列）

   单道性：内存中只有一道程序运行

   多道批处理系统：

   用户提交的作业都先放在外存排成一个队列，称为后备队列；之后，由作业调度程序按一定的算法从后备队列中选择若干作业调入内存，共享　　CPU和系统资源。

   多道批处理系统的优缺点：

   资源利用率高、系统吞吐量（单位时间内完成的总工作量）大、平均周转时间（从作业进入系统，到完成并退出系统为止的时间）长，缺点在于无交互能力。

4. 操作系统五大功能：

   处理机管理、内存管理、　　I/O 设备管理、文件管理、作业管理

5. 分时系统：

   为了弥补多道批处理系统交互性问题，引入分时系统，可以将一台计算机提供给多个用户同时使用，提高计算机利用率。

   分时系统的特点：

   多路性：宏观上，允许多用户同时工作。微观上，每个用户作业轮流运行一个时间片。

   独立性：每个用户各占一个终端

   及时性：用户请求可在较短时间内相应

   交互性：人机对话

6. 实时系统：

   系统能及时响应外部事件的请求，在规定时间内完成对该事件的处理，并控制所有实时任务协调一致的运行。

   实时系统与分时系统特点的区别：

   多路性：分时系统中的多路性与用户情况有关，时多时少。

   独立性： 实时信息处理系统中，　每个终端用户提出请求时，　互不干扰。 实时控制系统中，对信息采集和控制也是彼此互不干扰。

   及时性：实时控制系统的及时性要求比实时信息处理系统，分时系统更加严格。

   交互性：实时信息处理系统的交互性仅限于访问系统中的专用服务程序。

   可靠性：实时系统的可靠性更高

7. 操作系统发展：

   单用户单任务、单用户多任务、多用户多任务

8. 操作系统的基本特征：

   1. 并发性：

   并发性指的是多个事件在同一时间间隔内发生。并行性是多个事件在同一时刻发生。

   进程：指系统中能独立运行并作为资源分配的基本单位，　　由机器指令，　数据和堆栈组成。

   线程：一个进程包含若干线程，可利用进程的资源。进程是分配资源的基本单位，线程是独立运行和独立调度的基本单位。

2． 共享性：

即资源共享，有互斥共享方式、同时访问方式。

3． 虚拟技术：

分为时分复用技术、空分复用技术。

如果虚拟的实现是通过时分复用方式，即对物理设备进行分时使用，设 N 是谋设备所对应的逻辑设备数，则每台虚拟设备的平均速度必然小于等于 1/N。类似，空分复用实现虚拟，空间利用也小于等于 1/N 。

4． 异步性：

进程的推进速度不可预知。

9． 操作系统五大功能

1． 处理机管理

进程控制：为作业创建进程，撤销结束的进程，以及控制进程的状态转换

进程协调方式：进程互斥、进程同步两种方式

进程通信：

调度：

作业调度，即分配内存。将若干作业调入内存，为其建立进程，使之成为就需进程，并按一定规则插入就绪队列。

进程调度：即分配 CPU。从进程的就绪队列中，按一定算法选出一个进程，为其分配 CPU。

2． 存储器管理：

内存分配：为每道程序分配内存空间

内存保护：保证每道用户程序互不干扰

地址映射：将地址空间的逻辑地址转换为内存空间的物理地址

内存扩充：借助虚拟存储技术，从逻辑上扩充内存

3． 设备管理：

缓冲管理：

设备分配：根据用户进程的 I/O 请求，为之分配所需设备。

设备处理：实现 CPU与设备控制器之间的通信

4． 文件管理：

文件存储空间管理：为每个文件分配外存空间

目录管理：为每个文件建立目录项

文件读写管理和保护

5． 操作系统与用户间的接口：

用户接口、程序接口

10 操作系统结构设计

1． 传统的操作系统结构：

无结构操作系统：

模块化结构：将大的功能分为若干子功能，每个子功能为一个模块，再进一步细分，使之每一个模块只实现一个子功能。需要考虑模块的独立性，即模块的内聚性，耦合性。

分层式结构：将一个操作系统分为若干层，每层由若干模块组成。各层之间只存在单向依赖关系，即高层仅依赖紧邻它的低层。保证系统的正确性，易于扩展，但效率低。

2． C/S 模式

由客户机、服务器、网络系统构成。完成一次交互可分为，客户发送请求信息，服务器接受信息，服务器反馈消息，客户机接受消息。此种模式实现了数据的分布存储，便于集中管理，可扩展性。但可靠性差。

3． 面向对象程序设计 ：

4． 微内核操作系统结构：

将操作系统分为：微内核和多个服务器。有如下功能，进程线程管理、低级存储器管理、中断和陷入处理。

二．进程管理

1． 程序顺序执行的特征：

顺序性：每一操作必须在上一个操作完成后开始

封闭性：程序运行独占全部资源，不受外界影响

可再现性：只要程序执行环境和初始条件相同，当程序重复执行时，结果相同

2． 程序并发执行的特点：

间断性：并发执行的程序由于共享资源，以及为了完成同一任务相互合作，相互制约。将导致并发程序具有“执行 -暂停 -执行”间断性活动规律。

失去封闭性：多个程序共享资源。

不可再现性：由于失去封闭性，也就失去了再现性。即使执行环境和初始条件相同，结果却各不相同。

3． 进程的特征：

结构特征：进程由程序段，相关数据段和 PCB三部分组成

动态性：进程的实质是进程实体的一次执行过程。而程序只是一组有序指令集合。

并发性：多个进程同时存在于内存中，在同一时间段内同时运行。而程序不行。

独立性：进程实体可以独立运行，独立分配资源和独立接受调度（线程） 。而未建立 PCB的程序不能作为一个单独的单位参与运行。

异步性：指进程按各自独立的，不可预知的速度向前推进。

4． 进程的三个基本状态：

就绪状态：进程已分配到除了 CPU之外的所有必要资源，只要再获得 CPU，便可立即执行。

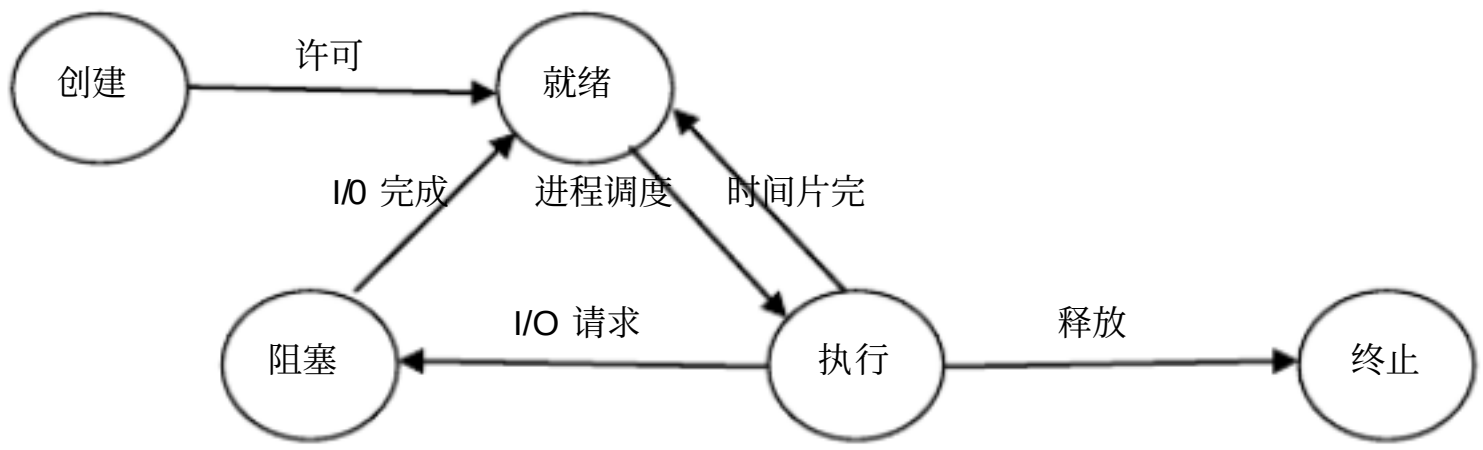
执行状态：进程已获得 CPU，程序正在运行

阻塞状态：进程的暂停状态称为阻塞状态。

进程的挂起状态：

当发生终端用户请求、父进程请求、负荷调节的需求、操作系统需求时，可以发生挂起。

进程状态转换：



进程创建：首先创建一个 PCB，将该进程转入就绪状态并插入就绪队列

进程终止：首先等待操作系统进行善后处理，然后清空 PCB，并将 PCB空间返还系统。

5． 进程控制块 PCB

PCB记录操作系统所需的，用于描述进程当前情况以及控制进程运行的全部信息。 使得一个在多道程序环境下不能独立运行的程序成为一个可独立运行的基本单位。 PCB是进程存在的

唯一标识。

进程控制块信息：

1. 进程标识符：用于唯一标识一个进程

内部标识符：为了方便系统使用。

外部标识符：由创建者提供，由用户进程访问该进程时使用。

2. 处理机状态：由处理机各种寄存器内容组成。

3. 进程调度信息：

进程状态、进程优先级、进程调度所需其他信息、事件（阻塞原因）　　。

4. 进程控制信息：

进程控制块的组织方式：

1. 链式方式：把同一状态的　PCB，用链接字链接成一个队列，形成就绪队列。
2. 索引方式：根据进程状态建立索引表，在每个索引表中，记录有相应状态的某个　　PCB在PCB表中的地址。

5. 进程控制

原语：由若干指令组成，用于完成一定功能的一个过程，是原子操作。在管态（核心态）下执行，常驻内存。

进程创建：

可以由进程树来描述，　子进程可以继承父进程的所拥有的资源，　当子进程被撤销时，　应将其从父进程中获取的资源归还父进程。当父进程被撤销时，同时撤销子继承。

引起进程创建的事件：

在多道程序环境中，只有进程才能在系统中运行，为了能让程序运行，需要建立进程。引起进程创建的事件有，用户登录、作业调度、提供服务、应用请求。

进程创建步骤：

申请空白　PCB，为新进程分配资源、初始化进程控制块、将新进程插入就绪队列

进程终止：

引起进程终止的事件：

正常结束、异常结束、外界干预（父进程请求，父进程终止，操作系统干预等）

进程终止过程：

1. 根据被终止进程的标识符，从　PCB集合中检索出该进程的　PCB，从中读出该进程状态。
2. 若该进程正处于执行状态，则立即终止其执行，并置调度状态为真，表示该进程被终止后应重新进行调度。
3. 若该进程还有子进程，则将其子进程终止。
4. 将被终止进程的全部资源，或归还父进程，或归还操作系统
5. 将被终止进程的　PCB从所在队列中移出。

进程的阻塞状态：当正在执行的进程，发现阻塞事件时，由于无法继续执行，于是进程调用 block 原语把自己阻塞。进程的阻塞是进程自身的一种主动行为。

进程的唤醒过程：首先将被阻塞的进程从等待队列中移出，将其　　PCB 中的现行状态改为就绪，然后将该　PCB插入就绪队列。

进程挂起与激活：

进程挂起：首先检查被挂起进程的状态，若处于活动就绪状态，便将其改为静止就绪；对于活动阻塞状态，改为静止阻塞。

进程激活：将进程从外存调入内存，检查其现行状态，若是静止就绪，便改为活动就绪；若是静止阻塞，改为活动阻塞。

6． 进程同步

1． 由于资源共享和进程合作，进程间存在两种形式的制约关系：

间接相互制约关系：源于资源共享

直接相互制约关系：源于进程合作

2． 临界资源：进程间应采用互斥方式，实现对这种资源的共享

临界资源实例 - 生产者消费者

```
/*
生产者消费者例子
采用循环队列方式存放数据，当队列中为空时，消费者不能取数据，
当队列为满时，生产者不能输入数据
*/
#include   <stdio.h>
#define    MAX 10
typedef    struct    queue{
        int   buffer[MAX];
        int   front;
        int   rear;
}queue;
void   producer(queue &q,     int   data){
        if  (isfull(q)==1){        //  如果队列为满，生产者无法插入数据
        } else {
                enqueue(q,data));
        }
}
void   customer(queue &q){
        if  (isempty(q)==1){     //  如果队列为空，消费者取不到东西
        } else {
                int    data = outqueue(q);
        }
}
```

单独运行生产者或消费者函数，　　都不会出现错误，　但当两者并行执行时，　会因为同时访问了同一个数据量而引起错误，因此，需要互斥的令生产者和消费者访问变量。

3． 临界区：

进程中访问临界资源的那段代码称为临界区，临界区前面需要增加一段用于冲突检验的代码，叫做进入区。相应的，在临界区后面加上一段退出区代码，用于将临界区正被访问的标志恢复为未被访问的标志。余下区域为剩余区。

同步机制应当遵循的原则：

空闲让进、忙则等待、有限等待、让权等待

4． 信号量机制

整型信号量：定义一个用于表示资源数目的整型量 S，除初始化外，仅能通过两个标准的原子操作 wait(), signal() 来访问，即 P，V 操作。原子操作在执行时不可中断。

```
void wait( int s){
    while (s<=0) // 当没有资源可以利用时，等待
        ;
    s=s-1; // 当有资源时，使用。每使用一个，资源个数减一
}
void signal( int s){
    s=s+1; // 释放资源，资源个数加一
}
```

记录型信号量：

当信号量 S<=0 时，就会不断检测，未遵循让权等待。因此，除了需要一个用于代表资源数目的整型变量 value 外，还需增加一个进程链表指针 L, 用于链接上述因 s<=0 而等待的进程。

```
// 记录型结构体
typedef struct semaphore{
    int value; // 记录资源个数
    struct semaphore *L; // 链接等待进程
}semaphore;
void wait(semaphore &s){
    //value 表示系统中某类资源数目，每次 wait 操作，系统中可供分配的资源数减一
    s.value = s.value-1;
//value 值可以一直减下去，为负数也可以，当为负数的时候，说明有进程自我阻塞了
    if (s.value<0)
        block(s.L); // 当没有资源可用时，进程调用 block 将自己阻塞。
                    // 此时， value 的绝对值表示该信号量链中已阻塞的进程数
}
void signal(semaphore &s){
    s.value = s.value+1;
    if (s.value<=0) // 若加一后， value 值还是小于等于 .则说明链表中还有阻塞的进程，需调用唤醒
        wakeup(s.L);
}
```

注：若 value 初值为 1，表示只允许一个进程访问临界资源，此时的信号量化为互斥信号量

AND 信号量：

有时，一个进程需要先获取多个共享资源后方能执行。 则这些共享资源都作为临界资源。 AND 同步机制是： 将进程在整个执行过程中所需所有资源， 一次性全部分配给进程， 待进程使用完后再一起释放。则在信号量中又添加一个 AND条件。

```
typedef struct source{
    int arr[MAX]; //source 资源结构体，数组中存放每种资源的对应可用数目
    int length; // 总的资源种类数
}source;
```

```
void  signal(source &s){
        for ( int   i=0;i<s.length;i++)
              s.arr[i]++;      // 一次性全部释放
}


void  wait(source &s){
        int   flag = 0;
        for ( int   i=0;i<s.length;i++){
              if  (s.arr[i]>=1)      // 判断所需的全部种类资源是否都能够分配给进程
                    flag = 1;
              else {
                    flag = 0;
                    break ;
              }
        }
        if  (flag==1){    // 如果能，则一次性全部分配给相应的进程
              for（int   i=0;i<s.length;i++)
                    s.arr[i]--;
        }else { }
}
```

信号量的应用：

1. 利用信号量实现进程互斥

为了使多个进程能互斥地访问谋临界资源，只需为该资源设置一互斥信号量 mutex（资源个数），并设初值为 1，然后将各个进程访问该资源的临界区至于 wait（mutex）和 signal(mutex) 操作之间。在使用信号量实现互斥时， wait 操作和 signal 操作需要成对出现。

2. 利用信号量实现前驱关系

5. 管程机制

管程：代表共享资源的数据结构， 以及由对该共享数据结构实施操作的一组过程所组成的资源管理程序，共同构成操作系统的资源管理模块。

管程组成：管程名称，管程内部的共享数据结构，对该数据结构进行操作的一组过程，对管程内共享数据设置初始值的语句。

```
public    class   Monitor{
        private     int   local_value;
        private    string monitor_name;
        private     void  init_value();
        private     void  use_value();
        public    void   public_use_value();
};
```

局部与管程内的数据， 仅能被管程内部的过程访问， 任何管程外的过程都不能访问它。 反之，局部于管程内部的过程也仅仅能访问管程内的数据。 管程每次仅允许一个进程进入管程， 实现了进程互斥。

管程和进程的区别：

1. 进程定义的数据结构是私有数据结构 PCB，而管程定义的是公有，如消息队列。

2. 进程是由顺序程序执行有关操作，管程主要进行同步操作。
3. 设置进程的目的在于实现系统并发性，而管程则是解决共享资源的互斥使用问题。
4. 进程是主动工作方式， 管程是被动工作方式， 进程通过调用管程中的过程对共享数据结构进行操作。
5. 进程可并发执行，管程不行。
6. 进程具有动态性，管程是一个资源管理模块，供进程调用。

6. 经典的进程同步问题

采用记录型信号量解决生产者 - 消费者问题：

```
void  init(semaphore &s){
        s.empty = MAX;
        s.full = 0;
        s.mutex = 1;
}
// 这些操作都可以循环执行，如下的只是他们的一次活动
void  producer(queue &q,semaphore &s){
        wait(s.empty);    // 先执行对资源信号量的操作
        wait(s.mutex);    // 再执行对互斥信号量的操作
        enqueue(q,data);
        signal(s.mutex);
        signal(s.full);
}
void  customer(queue &q,semaphore &s){
        wait(s.full);
        wait(s.mutex);
        outqueue(q);
        signal(s.mutex);
        signal(s.empty);
}
```

在生产者 - 消费者问题中，每个程序实现互斥的 P,V 操作必须成对出现。其次，对于资源信号量的 P,V 操作也要成对出现。每个程序的多个 P 操作顺序不能颠倒。

采用 AND信号量解决生产者 - 消费者问题：

原理一样， 只需将 wait , signal 操作换成 swait(),signal() 。例如，wait(s.full); wait(s.mutex); 用 Swait(full,mutex); 替换

哲学家进餐问题：

```
// 采用循环队列方式，假设有 N==5个哲学家，
// 所有信号量初始化为，第 i 为哲学家一次的活动方式如下，可以循环执行如下活动
void  profess(semaphore &s){  //  采用记录型方式
  wait(s.chopstick[i]);//  拿左边的筷子
  wait(s.chopstick[(i+1)mod(s.n)]);//  拿右边的筷子
  eat;
  signal(s.chopstick[i]);
  signal(s.chopstick[(i+1)mod(s.n)]);
  think;
```

```
}
void  profess(semaphore &s){      //  采用 AND 信号量方式
        think;
        Swait(s.chopstick[i],s.chopstick[(i+1)mod(s.n));
        eat;
        Ssignal(s.chopstick[i],s.chopstick[(i+1)mod(s.n)]);
}
```

读者 - 作者问题：

为实现读者和作者进程在读写时的互斥而设置了一个互斥信号量 Wmutex,另外, 再设置一个整形变量 Readercount 表示正在读的进程数目。 只要有读者在, 就不能写。又因为 readcount 是一个可被多个读者进程访问的临界资源，也应为它设置一个互斥信号量。

```
void  init(semaphore &s){
        s.rmutex = 1;
        s.wmutex = 1;
        s.readcount = 0;
}
void  reader(semaphore &s){
        wait(s.rmutex);      //  判断有没有其他读者
        if  (readcount==0)
                wait(s.wmutex);     //  有没有作者正在写
        s.readcount++;   //  没有，则读者数加一
        signal(s.rmutex);       //  恢复 rmutex
        read; //  阅读
        wait(s.rmutex);      //  判断有没有其他读者
        s.readcount--;
        if  (s.readcount==0)
                signal(s.wmutex);      //  只有当没有读者时，才执行此操作，为写操作提供资源
        signal(s.rmutex);
}
void  writer(semaphore &s){
        wait(s.wmutex);
        write;
        signal(s.wmutex);
}
```

7. 进程通信

进程通信类型：

1. 共享存储器系统：相互通信的进程共享某些数据结构或共享存储区，进程间通过这些空间进行通信。
2. 消息传递系统 ： 目前最流行。进程间的数据交换以格式化的信息为单位。
3. 管道通信：管道，即用于连接一个读进程和一个写进程以实现它们之间通信的一个共享文件。

消息传递通信的实现方法：

直接通信方式：利用操作系统所提供的发送命令，直接把消息发送给目标进程。

间接通信方式： 利用共享数据结构的实体, 暂存发送进程发送的消息, 接受进程从该实体中取出消息。这个中间实体成为信箱。

信箱的分类：
私用信箱：用户进程可自行建立
公用信箱：由操作系统创建
共享信箱：由谋进程创建
信箱通信时，发送进程和接收进程间的关系：
一对一、一对多、多对一、多对多

8. 线程

操作系统引入线程， 是为了减少程序并发执行时所付出的时空开销。 一个进程拥有多个线程，线程是独立调度的基本单位， 线程一般不具有自己的资源， 他们会利用进程的资源。 在一个进程中的多个线程可以并发执行。进程的系统开销远高于线程。
线程的特点：
轻型实体、独立调度和分配的基本单位、可并发执行、共享进程资源
用户线程和内核线程：
用户线程： 在内核的支持上， 在用户层通过线程库实现创建， 调度和管理， 且无需内核支持。
内核线程：在操作系统上，在其内核空间内执行创建，调度和管理
多线程模式：
一对多：一个内核线程对应多个用户线程。效率较高，但如果一个线程执行阻塞，其他线程也就被阻塞。
一对一：一个内核线程对应一个用户线程，并发性好，但开销大。
多对多：多个内核线程对应更多用户线程，上面两者的折中。

三 处理机调度和死锁

1. 高级调度：

高级调度又称作业调度或长程调度， 主要功能是依据某种算法， 把外存上的处于后备队列的作业调入内存。
作业：在批处理系统中，是以作业为基本单位从外存调入内存的。
作业步： 在作业运行期间， 每个作业都必须经过若干相对独立， 又相互关联的顺序加工步骤才能得到，这些步骤叫做作业步（ job step ）。
一个典型的作业分三步：
编译、连接装配（将若干程序段装配为可执行程序）、运行（目标程序调入内存）
作业流：若干作业进入系统，被依此放在外存，形成输入的作业流。

作业控制块 JCB：
是作业在系统中存在的标志， 其中保存了系统对作业进行管理和调度所需的全部信息。 每当作业进入系统，系统为每个作业创建一个 JCB。当作业完成，则系统回收分配给它的资源，撤销 JCB。

作业调度：
根据 JCB中的信息， 审查系统能否满足用户作业的资源需求， 按照一定算法， 从外存后备队列中选择某些队列调入内存， 并为它们创建进程， 分配资源。 然后再将新进程插入就绪队列。
每次作业调度需要考虑两个问题：
决定接纳多少个作业，决定接纳哪些作业，取决于调度算法。最简单的 FCFS算法，较常用的是短作业优先算法，基于作业优先级算法，较好的是响应比高者优先算法。

2. 低级调度（进程调度）：

进程调度用于决定就绪队列中的哪个进程应获得处理机。

进程调度的功能：

保存处理机现场信息、按某种调度算法选取进程、把处理器分配给进程

进程调度的三个基本机制：

1. 排队器：为提高进程调度效率，应事先将系统中所有就绪进程按一定顺序排列成队列
2. 分派器：把进程调度算法所选的进程，从就绪队列中取出，然后进行上下文切换，将处理器分配给它。
3. 上下文切换机制：当对处理机切换时，会发生两对上下文切换。在第一对切换时，系统保存当前进程的上下文，而装入分派程序的上下文，以便分派程序运行。在第二对上下文切换时，将移出分派程序，而把新选进程的 CPU现场信息装入到处理机的相应寄存器。

进程调度方式

非抢占式：

一旦把处理机分配给某个进程后，会一直让它运行下去，直至此进程完成，或发生某些事件被阻塞时，才把处理机分配给其它进程。但难以满足紧急任务的需求。

抢占式：

允许调度程序根据某种原则暂停某个正在执行的进程，将已分配的处理机重新分配给另一个进程。但开销比非抢占式大。

抢占式基于的原则：

1. 优先权原则：通常一些重要和紧急的作业赋予较高的优先权。
2. 短作业（进程）优先原则：当新到达的作业（进程）比正在执行的作业（进程）明显短时，将暂停当前长作业（进程）的执行，使短作业优先执行。
3. 时间片原则：各个进程按时间片轮流运行。适用于分时系统，大多数实时系统，以及要求较高的批处理系统。

3 中级调度（中程调度）：

中级调度是为了提高内存利用率和系统吞吐量，为此，应使那些暂时不能运行的进程让出内存资源，将它们调至外存上去等待。

这三种调度方式中，进程调度频率最高，作业调度所需时间最长，中级调度基于这两者之间。

4. 选择调度方式和调度算法的若干准则：

面向用户的准则：

1. 周转时间短：

   周转时间：作业被提交给系统开始，到作业完成的这段时间间隔。包括四部分，作业在外存后备队列上等待调度时间，进程进入就绪队列上等待调度时间，进程执行时间，以及进程等待 I/O 操作完成的时间。

   可以把平均周转时间描述为：

$$T = \frac{1}{n}\left[\sum_{i=1}^{n} T_i\right]$$

   作业的周转时间 T 与系统为它提供的服务时间 Ts 之比，即 W =T/Ts，称为带权周转时间，则平均带权周转时间为：

$$T = \frac{1}{n}\left[\sum_{i=1}^{n} \frac{Ti}{Ts}\right]$$

2. 相应时间快：

   包括三部分时间：输入请求信息传送到处理机的时间，处理机对请求信息进行处理的时间，以及将所形成的相应信息送回到终端的时间。

3. 截止时间的保证：

   截止时间，即谋任务必须开始执行的最晚时间，或必须完成的最晚时间

4. 优先权准则：

   在批处理，分时和实时系统中都可以遵循优先权准则。

面向系统的准则：

1. 系统吞吐量高：

   用于评价批处理系统系能的另一个重要指标

2. 处理机利用率好：
3. 各类资源平衡利用：

5 调度算法

先来先服务 FCFS调度算法：

每次调度都是从后备作业队列中选作若干最先进入该队列的作业，将其调入内存，为其分配资源，创建进程，放入就绪队列。

FCFS算法有利于长作业（进程），而不利于短作业（进程）。有利于 cpu 繁忙型的作业，而不利于 I/O 繁忙型的作业。

短作业（进程）优先 SJ(P)F 调度算法：

是对短作业或短进程优先调度算法。短作业优先 SJF 调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业，将其调入内存。短进程优先 SPF调度算法是从就绪队列中选择一个估计时间最短的进程，将处理机分配给它。

该算法对长作业不利。未能完全考虑作业的紧迫程度。

非抢占式优先权调度算法：

系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程一直执行下去，直至完成，或因某些事件而放弃使用权。主要用于批处理系统中。

抢占式优先权调度算法：

系统同样把处理机分配给优先权最高的进程，但当有更高优先权的进程进入队列时，进程调度就会停止当前进程，转而将处理机分配给这个优先权更高的进程。

优先权类型：

静态优先权：是在创建进程时确定的，且在进程的整个运行期间保存不变。

确定进程优先权的依据有：进程类型、进程对资源的需求、用户要求

动态优先权： 在创建进程时所赋予的优先权，是可以随进程的推进或其等待时间的增加而改变的。

高响应比优先调度算法：

优先权 =（等待时间 +要求服务时间） / 要求服务时间 = （响应时间） / 要求服务时间

这样，优先权可以随动态优先权的变动而变化，更加灵活。

时间片轮转法　RR调度：

时间片轮转法：把　CPU分配给队首进程，并令其执行一个时间片，当执行的时间片用完，由一个计时器发送时钟中断请求，　调度程序便依据此信号终止此进程的执行，　并将其送至就绪队列的队尾，然后再把处理机分配给新的队列头进程，如此往复。适合分时系统。

多级反馈队列调度算法：

设置多个就绪队列，　并为各个队列赋予不同的优先级，　第一个队列的优先级最高，　其余各队列优先级逐级递减。　该算法赋予各个队列中进程执行时间的大小各不相同，　优先级队列越高的，进程获得的时间片越小。

当一个新进程进入进程后，首先将其放入第一个队列的队尾，按　FCFS算法排队等待调度，当轮到其执行时，如果其能在该队列时间片内完成，则让其执行完毕。若其未能完成，则调度程序将其转到下一个队列的队尾，同样按　FCFS方式等待调度。如此继续下去。当一个长进程或长作业从第一个队列依此降至第　n 个队列后，在第 n 个队列中便采用按时间片轮转方式运行。

仅当第 1-（i-1 ）队列均空时，才会调度第　i 个队列的进程运行。若处理机正在第　i 个队列中为某个进程服务时，　又有新进程进入优先权较高的队列，　则此时新进程将抢占正在运行的进程的处理机，　即把正在运行的进程放回到第　i 队列的末尾，　把处理机分配给新到的高优先级进程。

6 产生死锁的原因和必要条件

死锁，是指多个进程在运行过程中因争夺资源而造成的一种僵局，使得进程无法向前推进。

产生死锁的原因：

竞争资源、进程间推进顺序不当

1. 竞争资源

   系统中的资源分为可剥夺性资源和非可剥夺性资源。前者指进程在获得这类资源后，该资源可以再被其他进程或系统剥夺。　后者指当系统分配了这类资源后，　不能够再行剥夺。

   竞争非剥夺资源：

   例如，系统中只有一个打印机和一个磁带机，供两个进程共享。若　A 进程已占用了打印机，B 进程已占用了磁带机。若此时，　B 继续要求打印机，则由于产生　I/O 请求，　B 将阻塞；若 A 有要求磁带机，则　A 也阻塞。因而两者都等待对方的资源，但它们又因为阻塞而不能继续推进，从而导致不能释放自己的资源。产生死锁。

   竞争临时性资源：

   指由一个进程产生，被另一个进程使用一短暂时间后便无用的资源。

   例如下述进程请求方式可能产生死锁

   P1 : REQUEST(S3); RELEASE(S1);

   P2 : REQUEST(S1); RELEASE(S2);

   P3 : REQUEST(S2); RELEASE(S3);

2. 进程推进顺序不当：

   进程推进顺序不当会导致死锁。

7 产生死锁的必要条件

互斥条件：一段时间内某个资源只由一个进程占用。

请求和保持条件： 指进程已经保持了至少一个资源， 但又提出新的资源请求， 而该资源又被其他进程所占用。

不剥夺条件：指进程已获得的资源，在未使用完成前，不能被剥夺。

环路等待条件：指发生死锁时，必然存在一个进程 - 资源的环形链。

8 处理死锁的基本方法

预防死锁：破坏产生死锁的四个必要条件中的一个或几个条件

避免死锁：在资源动态分配过程中，用某种方式阻止系统进入不安全状态

检测死锁：不事先预防，也不检测系统是否进入不安全区，而是通过检测机制，及时检测出死锁的发生，然后采取适当措施消除死锁。

解除死锁：与检测死锁配套使用。常用方式是撤销或挂起一些进程，以便回收一些资源。

系统的安全状态： 指系统能按某种进程顺序， 来为每个进程分配其所需的资源， 直至满足每个进程对资源的最大需求。 如果系统无法找到这样一个安全序列， 则称系统进入不安全状态。

银行家算法：

银行家算法可以避免死锁，其数据结构为：

```
#define   MAX 10
/*
可利用资源向量，每个元素代表一类可利用的资源数目，
初始值是系统中所配置的该类全部可用资源的数目。如果 avaliable[j]=k, 则表示
系统中现有 Rj 类资源 K个。
最大需求矩阵 MAX, 定义了系统中 n个进程中的每个进程对 m类资源的最大需求。
若MAX[i][j] = k, 表示进程 i 需要 Rj 类资源的最大数目为 k
分配矩阵 allocation[i][j] = k, 表示进程 i 当前获得的资源 Rj 有 k个
需求矩阵 need[i][j] = k, 表示进程 i 当前还需要 Rj 类资源 k个
Need[i][j] = Max[i][j]-Allocation[i][j];
*/
typedef   struct   bank{
      int   avaliable[MAX];     //  可利用资源向量
      int   Max[MAX][MAX]; //  最大需求矩阵
      int   Allocation[MAX][MAX];     //  分配矩阵
      int   need[MAX][MAX];  //  需求矩阵
}bank;
```

设 request_i 是进程 Pi 的请求向量，如果 request_i[j] = k, 表示进程 Pi 需要 K 个 Rj 类型的资源。当 Pi 发出资源请求后，系统进行如下检测：

1） 若 request_i[j]<=need[i][j], 转向 2，否则认为出错。

2） 若 request_i[j]<=avaliable[j], 则转向 3，否则认为出错

3） 系统分配资源给进程，并修改数据结构中数据

```
Avaliable[j]=avaliable[j]-request_i[j];
Allocation[i][j]=allocation[i][j]+request_i[j];
Need[i][j]=need[i][j]-request_i[j];
```

4) 系统进行安全性算法，检查此次资源分配后系统是否处于安全状态。

```
void allocate(bank &ba, int request[][MAX], int i, int j){
    if (request[i][j]<=ba.need[i][j]){
        if (request[i][j]<=ba.avaliable[i][j]){
            // 资源分配
            ba.avaliable[i][j] = ba.avaliable[i][j]-request[i][j];
            ba.Allocation[i][j] = ba.Allocation[i][j]+request[i][j];
            ba.need[i][j] = ba.need[i][j]-request[i][j];

        }
    }
}
```

安全性算法：

工作向量 work，表示系统可以提供给进程继续运行所需的各类资源数目，初始，work = available 。

结束向量 Finish ，表示系统是否有足够的资源分配给进程，使之运行完成。初始，finish[i]=false; 当有足够资源分配给进程时，finish[i]=true;

算法如下：

```
void check(bank &ba, int work[], bool finish[], int i, int j){
    if (finish[i]== false &&ba.need[i][j]<=work[j]){
        // 进程 Pi 尚未结束，其所需资源系统可以满足
        // 说明可以获得资源，顺利执行。之后释放分配给它的资源。
        work[j] = work[j]+ba.Allocation[i][j];
        finish[i]= true ;
    } else {
        if (finish[i]== true )
            // 此进程终止
    }
}
```

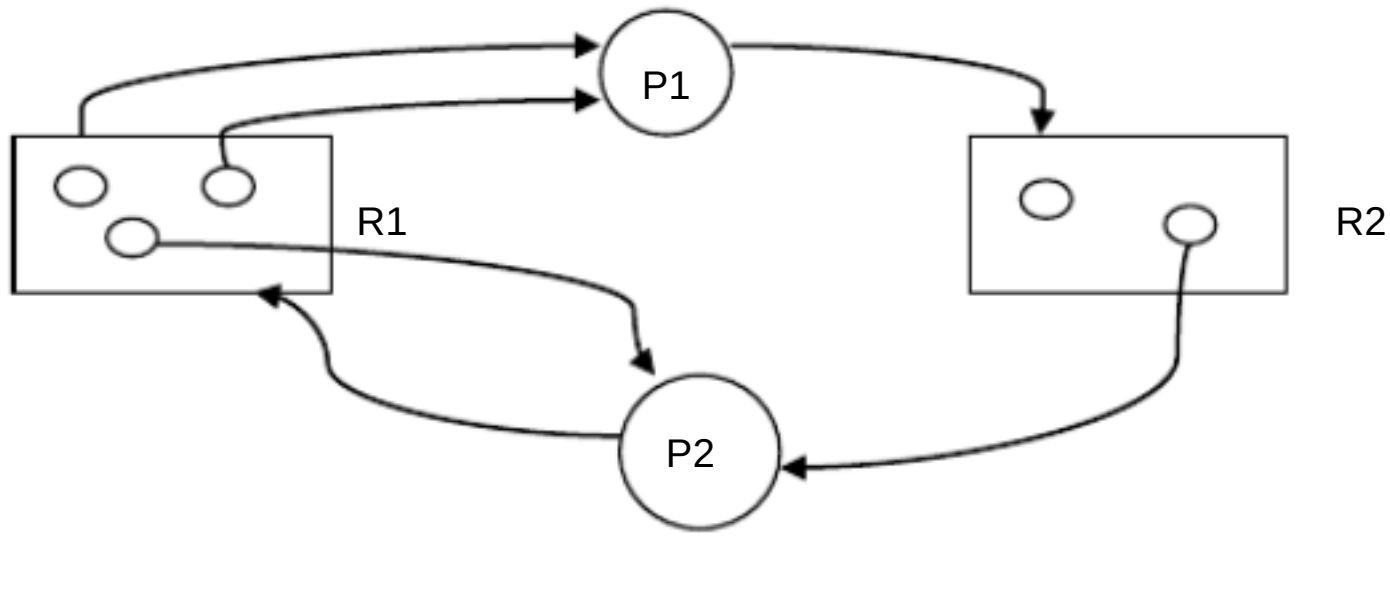
死锁检测：

1. 资源分配图：

死锁检测可以利用资源分配图来描述。该图由一组节点 N 和一组边 E 所组成的一个对偶 G = (N,E) 。

把 N 分为两个互斥的子集，即一组进程节点 P 和一组资源节点 R，N=P并 R。

凡是属于 E 的边，都连接着 P 和一个 R 中的节点， e = {pi,Rj} 表示资源请求边，由进程指向资源，表示进程申请资源。 E = {Rj,Pi} 表示资源分配图，把资源分配给进程。如图例：



注：分配边应始于一个资源点

2. 死锁定理

将资源图简化，来检测当系统处于 S 状态时是否为死锁状态。

方法为：在资源分配图中，找一个既不阻塞又非独立的进程节点 Pi，在顺利的情况下，Pi 可以获得所需资源而继续运行，直至运行完毕，再释放其所占的所有资源，相当于消去 Pi 所请求的请求边和分配边，使之成为孤立的节点。

如上图，将 P1 两个分配边和一个请求边消去，留下一个孤立的 P1 节点。这样，图中所有的资源由 P2 使用，当 P2 获得资源并执行完毕后，将 P2 的边也删去。利用这用简化方式，如果说，能够消去图中所有的边，使所有节点都成为孤立节点，则该图可以完全简化。

对于较复杂的资源分配图，可能有多个既非阻塞，又非孤立的进程节点，不同的简化顺序得到的简化图是相同的。

死锁的解除：剥夺资源，撤销进程

四 存储器结构

1. 程序的装入和链接

将一个用户源程序变为一个可在内存中执行的程序，首先要编译，由编译程序将源程序编译成目标模块，之后链接，由链接程序将编译后的目标模块，以及所需库函数链接起来，形成一个完整的装入模块。最后装入，由装入程序将装入模块装入内存。

程序装入：

绝对装入方式：在编译时，如果知道程序将驻留在内存的什么位置，则，编译程序将产生绝对地址的目标代码。

可重定位装入方式： 在多道程序环境下，所得到的目标模块的起始地址通常从 0 开始，程序中的其他地址也都是相对于起始地址计算的。此时可以采用可重定位方式装入，根据内存当前情况，装入到适当位置。采用可重定位装入程序将装入模块装入内存后，会使得装入模块的所有逻辑地址与实际装入内存的物理地址不同。

动态运行时装入方式：

可重定位装入方式不允许程序运行时在内存中移动位置，而动态方式可以。动态运行时的装入程序在把装入模块装入内存后，并不立即把装入模块中的相对地址装换为绝对地址，而是把这种地址转换推迟到程序真正要执行才开始。这就需要重定位寄存器的支持。

程序链接：

静态链接：在程序运行前，先将各自目标模块及其所需库函数，链接成一个装入模块，以后不再拆开。属于事先链接。

装入时动态链接：将目标模块，在装入内存时，采用边装入边链接的链接方式。

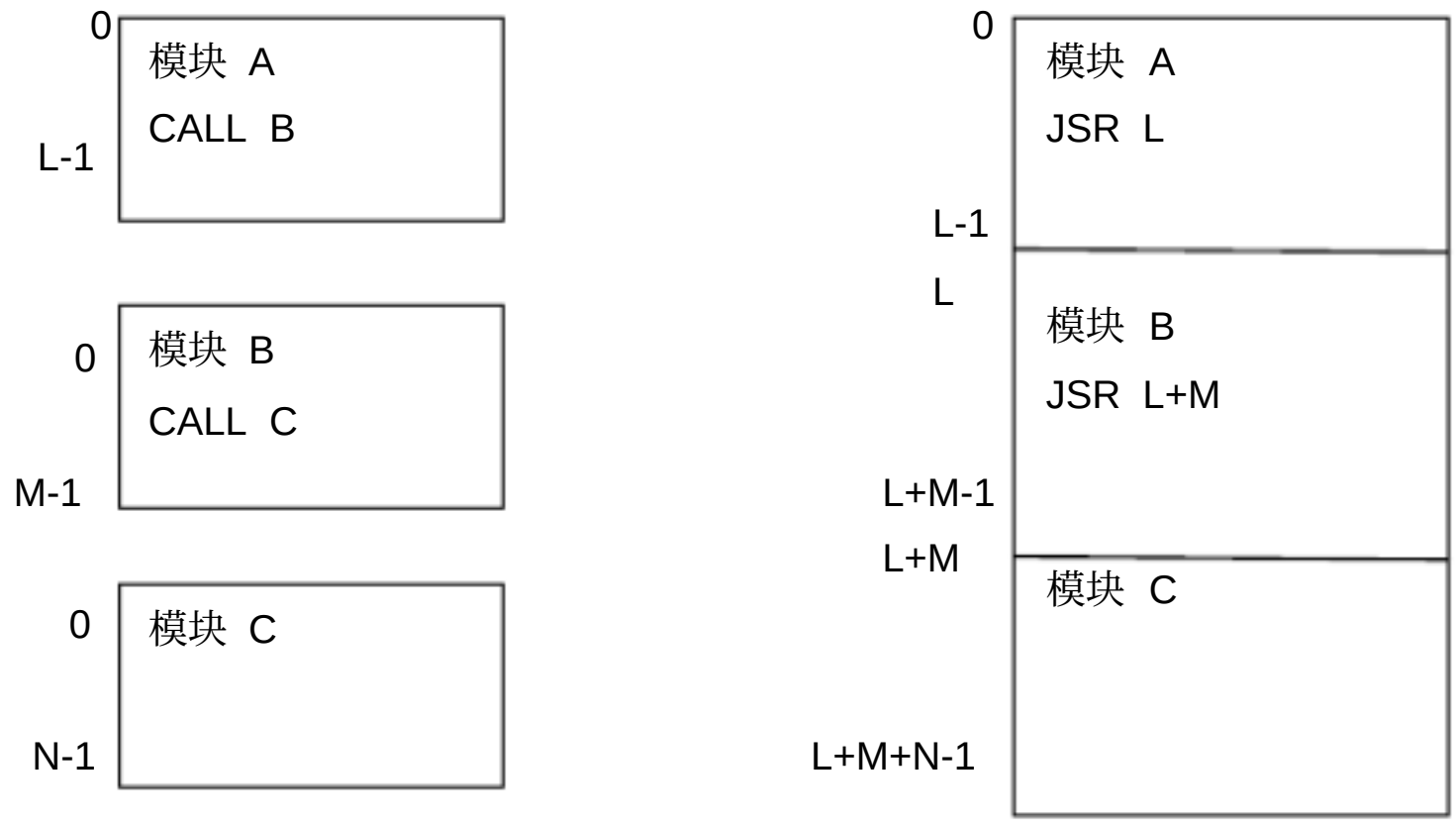
运行时动态链接： 对某些目标模块的链接，是在程序执行中需要该模块时，才对它进行链接。

静态链接

例如：有 A,B,C 三个目标模块，长度分别为 L,M,N。其中 A 中有 CALL B语句， B 中有 CALL C 语句。欲将 A,B,C 装配成一个装入模块。需要修改两个东西：

1） 对相对地址进行修改

2） 变换外部调用符号，将调用符号换成相对地址



装入时动态链接： 即在装入一个目标模块时， 若发生一个外部模块调用事件， 将引起装入程序去找出相应的外部目标模块， 并装入内存。 便于修改和更新， 便于实现对目标模块的共享。

内存的连续分配方式：

单一连续分配：适用于单用户，单任务的操作系统。

固定分区分配： 将内存用户空间划分为若干个固定大小的区域， 每个分区中只装入一道作业。允许有几道作业并发执行。

动态分区分配：根据进程的实际需求，动态分配空间。

动态分区分配算法：

1） 首次适应 FF 算法：

FF 算法要求空闲分区链以地址递增的次序链接。在分配内存时，从链首开始顺序查找，直至找到一个大小能满足要求的空闲分区为止，然后再按照作业的大小，从该分区中划出一块内存空间分配给请求者，剩余空闲分区仍留在空闲链中。

2） 循环首次适应 NF 算法：

在为进程分配内存空间时，不再是每次都从链首开始查找，而是从上一次找到的空闲分区的下一个空闲分区开始查找，直至找到满足要求的空闲分区，从中划分出一块满足要求大小的内存空间分配给作业。

3） 最佳适应 BF 算法：

每次为作业分配内存时，总是把能满足要求，又是最小的空闲分区分配给作业。

4） 最坏适应 WF 算法：

要扫描整个空闲分区表或链表，总是挑选一个最大的空闲分区分割给作业，优点在于可使剩下的空闲分区不至于太小，产生碎片的几率很小。

5） 快速适应 QF 算法：

将空闲分区根据其容量大小进行分类，对于每一类具有相同容量的所有空闲分区，单独设立一个空闲分区链表。这样，系统中存在多个空闲分区链表，同时在内存中设立一张管理索引表，该表的每一个表项对应一种空闲分区类型，并记录该类型空闲分区链表头指针。

在动态分区存储管理方式中，主要操作是分配内存和回收内存

伙伴系统：

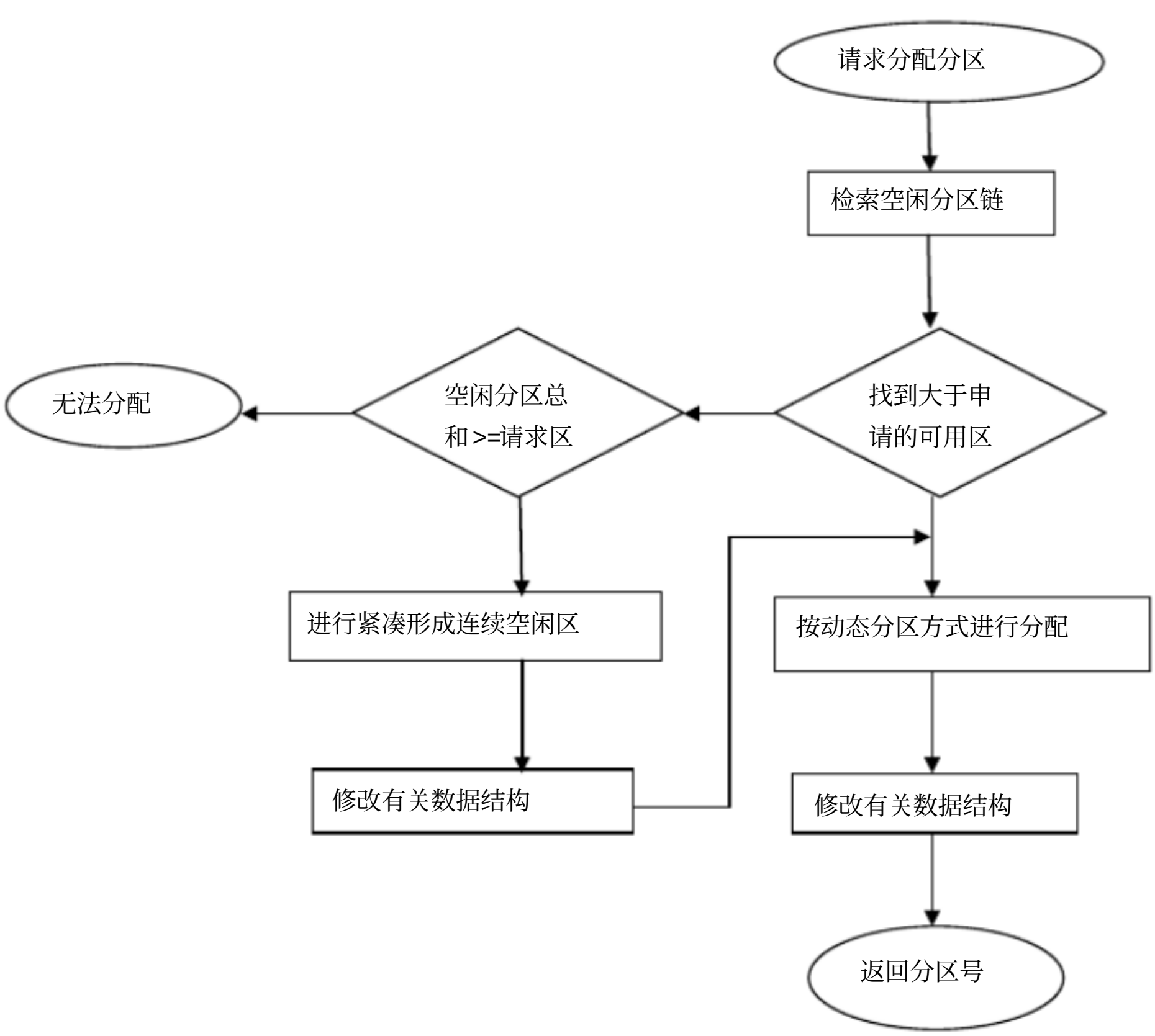
固定分区和动态分区都有不足之处，固定分区内存空间利用率低。动态分区系统开销大。

伙伴系统是二者的折中，规定，无论分配分区还是空闲分区，分区大小均为 2 的 k 次幂， k 为整数， 1<=k<=m ； 2 的一次幂为最小分区， 2 的 m 次幂为最大分区。

可重定位分区分配：

在内存分配时，若在系统中有若干小的分区，总容量足够大，但单独一个却不能容下程序，这些分区也不相邻，无法把程序装入内存。引起了内存浪费，产生碎片。

通过移动内存中的作业的位置，　把原来多个分散的小分区拼凑成一个大分区的方法，　称为紧凑法，提供内存利用率。但由于紧凑后的用户程序在内存中的位置发生改变，因此，需要重定位。



对换的引入：

将内存中暂时不能运行的进程或暂时不用的程序和数据调出到外存，　以便腾出足够的内存空间，再把已具备运行能力条件的进程或程序，　数据调入内存。　如果对换是以整个进程为单位的，称之为整体对换，或进程对换。用于分时系统。若对换以页或段为单位，则分别称之为页面对换，分段对换，用以支持虚拟存储系统。

2．　基本分页存储管理方式

连续分配方式会形成许多碎片，　从而离散分配方式，　采用页为分配的基本单位，　可以避免产生碎片。若采用段为基本单位，则称之为分段存储管理方式。

页面：

分页存储管理是将一个进程的逻辑地址空间分成若干大小相等的片，　　称之为页。 并给各个页加以编号，从　0 开始。相应的，内存也分成与页面等大的若干存储块，称为物理块或页框，同样为它们加以编号。　在为进程分配内存时，　以块为单位将进程的若干页分别装入到多个可以不相邻的物理块中。　由于进程的最后一页通常装不满而形成不可利用的碎片，　　称之为“ 页内碎片 ”。页面大小一般为　 2 的幂。

页面地址结构：

分页地址由页号，偏移量（页内地址）组成。如下例：

页号	位移量

若上图　0　－　11位为页内地址，即每页　 4KB，　12　－　31为页号，即地址空间最多允许　 2^20 页

若给定一个逻辑地址空间中的地址为　 A，页面大小为　L，则页号和页内地址　 D 为：

$$P = [A/L]$$

$$D = [A] \text{ MOD } L$$

页表：

系统要为每一个进程都建立一个页表。　 在进程地址空间内的所有页，　依此在页表中有一页表项，记录了相应页在内存中的对应的物理块号。页表是实现从页号到物理块号的地址映射。

地址变换机构：

该机构是实现从逻辑地址到物理地址的转换，　　页内地址和物理地址是一一对应的，　地址变换机构的任务是将逻辑地址中的页号，转换为内存中的物理块号。借助页表实现。

基本的地址变换机构：

如果页号大于或等于页表长度，　 表示本次所访问的地址超过进程的地址空间。　 若未越界，　则将页表地址与页号和页表项长度的乘积相加（即，基地址　　+偏移量）得到该表项在页表中的位置，从中得到物理块号。

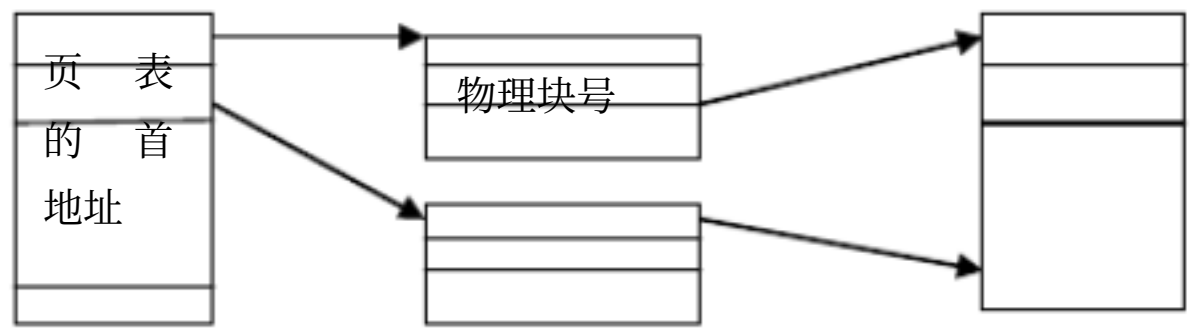
具有快表的地址变换机构：

为了提高地址变换速度，　在地址变换机构中增设一个具有并行查询能力的特殊高速缓冲寄存器，称为“ 快表 ”。在 CPU给出有效地址后，由地址变换机构自动将页号送入快表，并将此页号与快表中的所有页号做比较，　 若有匹配的，　则表示要访问的页表项子在快表中。　 这样可以直接从快表中读取对应的物理块号。若，不在，则还须再访问内存中的页表，找到后，将物理块号送至地址寄存器，同时将此表项加入快表的一个寄存器中，如果满了，则替换。

两级页表：

以 32 位逻辑空间为例，当页面大小为　 4KB（12 位），若采用一级页表结构，对应（　 32-12）20 位的页号。若采用二级页表，则需要对页表分页，外层页表中的页内地址为　　10 位，外层页号也为　10 位。

在页表中的每个表项中存放的是进程的某页在内存中的物理块号。　　而在外层页表的每个页表项中，所存放的是某页表分页的首地址。



在地址变换机构中同样增设了一个外层页表寄存器，用于存放外层页表起始地址，并利用逻辑地址中的外层页号，作为外层页表的索引，从中找到指定页表的起始地址，再找到指定的页表项，从中找到物理块号。在采用两级页表结构的情况下，对于正在运行的进程，必须将其外层页表调入内存，而对页表则只需调入一页或几页。

3. 分段存储方式：

分段存储的用途：

1） 方便编程：用户把作业按照逻辑关系划分为若干段，每段从 0 开始编址

2） 信息共享：分页系统中的页只存放信息的物理块，无完整意义，而段是信息的逻辑单位

3） 信息保护

4） 动态增长：处理数据段的动态增长

5） 动态链接：只有当目标程序运行过程中，才将某段目标程序调入内存并链接

分段的基本原理：

作业的地址空间被划分为若干个段，每段定义了一组逻辑信息。每段从 0 开始编址，并采用一段连续的地址空间。段的长度由相应的逻辑信息组的长度决定，因而，各段长度不等。

段号	段内地址

段号显示出这个作业最长有多少个段，段内地址显示出段的大小

段表：

在分段存储管理系统中，为每个分段分配一个连续的分区，而进程中的各段可以离散的移入内存中的不同分区中。在系统中，为每个进程创建一个短映射表。每段在表中有一个表项，记录该段在内存中的基地址和段的长度。用于实现从逻辑段到物理内存区的映射。

段地址变换机构：

物理地址 = 基地址（段表中起始地址和该段的段号查到的基地址） + 段内地址（若没越界）

分段和分页的区别：

1） 页是信息的物理单位，分页实现离散分配方式，消减内存的碎片，提高内存利用率。段是信息的逻辑单位，含有一组意义相对完整的信息。为了更好的满足用户需求。

2） 页的大小固定且由系统决定，分为页号和页内地址。系统中只有一种大小的页面。而段的长度不固定，由用户编写的程序决定。

3） 分页的作业地址空间是一维的。而分段的作业地址空间是二维的，在标识一个地址时，需要提供段名（确定基地址），还需要段内地址

4. 段页式存储

先将用户程序分成若干段，再把每段分成若干页，并为每个段赋予一个段名。在段页式系统中，其地址结构由段号，段内页号和页内地址组成。利用段表起始地址和段号求出该段所对应的段表项在段表中的位置，从中得到该段的页表起始地址，并利用逻辑地址中的段内页号来获得对应页的页表项位置，从中读出该页所在的物理块号，再利用块号和页内地址构成物理地址。段页式需要三次访问内存。为此，添加了一个缓存，每次先到缓存中找，若有，则直接得到相应页的物理块号。

5. 虚拟存储器：

虚拟存储器： 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。

虚拟存储器实现方法：

1） 分页请求系统：

在分页系统基础上，增加了请求调页功能和页面置换功能所形成的页式虚拟存储系统。

2） 请求分段系统：

在分段系统基础上，增加了请求调段及分段置换功能后所形成的段式虚拟存储系统。

虚拟存储器特征：

多次性：一个作业被分成多次调入内存运行 .

对换性 :允许在作业的运行过程中进行换进 ,换出 .

虚拟性 :从逻辑上扩充内存容量 .

请求分页存储管理方式 :

请求分页中的硬件支持 : 页表机制

缺页中断机构 : 当所要访问的页面不在内存中时 ,产生缺页中断 .请求操作系统将所缺页调入内存 .它与通常的中断有明显区别 :

1) 在指令执行期间产生和处理中断信号。一般, CPU都是在一条指令执行完毕后，才检验是否有中断请求达到。

2) 一条指令在执行期间，可能产生多次缺页中断。

内存分配策略和分配算法：

为进程分配内存时，需要考虑三个问题：

1） 最小物理块数的确定：是能保证进程正常运行所需要的最小物理块数。

2） 物理块的分配策略：

固定分配局部置换：为每一个进程分配一定数目的物理块，在整个运行期间都不改变。

如果进程再发生缺页，只能从给它分配的那些进程中置换出页来使用。

可变分配全局置换：先为系统中的每个进程分配一定数目的物理块，而 OS也保留一部分空闲物理块。当缺页时，由系统取出一部分空闲块给进程，只有当系统中的空闲块都用光了，才会采用调度策略换出一部分页。

可变分配局部置换 :为每个进程分配一定数目的物理块, 但发生缺页时, 只允许从该进程在内存中的页面选出一个页换出。

物理块分配算法：

1） 平均分配算法：

将系统中的所有可供分配的物理块平均分配给各个进程。

2） 按比例分配算法：

Bi = (si/s)*m s = $\sum_{i=1}^{n} si$ m 为物理块总数。 Si 为每个进程分配到的页面数（相当于进程大小）。即进程大小比例来分配物理块。

3） 考虑优先权的分配算法：

将内存中的可供分配的所有物理块分为两部分，一部分按比例分配给各个进程，另一部分则根据各进程的优先权，适时增加其相应份额后，分配给进程。

在请求分页系统中的外存分为两部分，用于存放文件的文件区和用于存放对换页面的对换区。

6. 页面置换算法：

1） 最佳置换算法：

所选择被淘汰的页面，是以后永远不使用的，或许长期不再访问的页面。

2） 先进先出页面置换算法：

总是淘汰最先进入内存的页面，即选择在内存中驻留时间最长的页面予以淘汰。

3） 最近最久未使用 LRU置换算法：其硬件支持是寄存器或栈

4） Clock 置换算法：

只需为每个页面设置一位的访问位， 再将内存中的所有页面都通过链接指针链接成一个循环。当某页被访问时，其访问位置 1。置换算法在选择一页淘汰时，只需检查页的访问位。若为 0，则选择该页被换出，若为 1，则重新置 0，暂时不换出，在按照 FIFO算法检查下一个页面，当检查到队尾时，若其访问还为 1，则返回队首继续检查。

5） 改进型 clock 置换算法：

选择页面换出时，既要是未被访问过的，还要是未被修改过的页面。把同时具有这两个条件的作为首选淘汰页面。