# 目录

# 1. 进程与线程

## 1.1. 进程描述

### 1.1.1. 进程描述与组成

进程：一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。

一个进程包括：程序的代码；程序处理的数据；程序计数器中的值，指示下一条指令；一组通用寄存器；一组系统资源；

### 1.1.2. 进程与程序的关系与区别

进程与程序的关系：

1) 程序是进程产生的基础；
2) 程序每次运行构成不同的进程；
3) 进程是程序功能的体现；
4) 通过多次执行，一个程序可对应多个进程；通过调用关系，一个进程可包括多个程序。

进程与程序的区别：

1) 进程是动态的，程序是静态的；程序是有序代码的集合，进程是程序的执行，进程有核心态、用户态。
2) 进程时暂时的，程序是永久的；
3) 进程与程序的组成不同：进程的组成包括程序、数据和进程控制块。

### 1.1.3. 进程的特点

动态性：动态创建、结束进程

并发性：进程可以被独立的调度并占用 CPU 运行，并发运行；

独立性：不同的进程的工作不互相影响（独立的地址空间、页表）；

制约性：因访问共享资源或进程间同步而产生制约。

### 1.1.4. 进程控制结构 TCB

进程控制块：OS 管理控制进程所用的信息集合。PCB 是进程存在的唯一标识。

**进程控制块包括：**

1) 进程标识信息：如本进程的标识，本进程的父进程；用户标识；

2) 处理器状态信息：保存程序的运行现场信息（地址等寄存器；控制和状态寄存器；栈指针）

3) 进程控制信息：调度和状态信息；进程间通信信息；存储管理信息；进程所用资源。

## 1.2. 进程的生命期管理

### 1.2.1. 进程状态

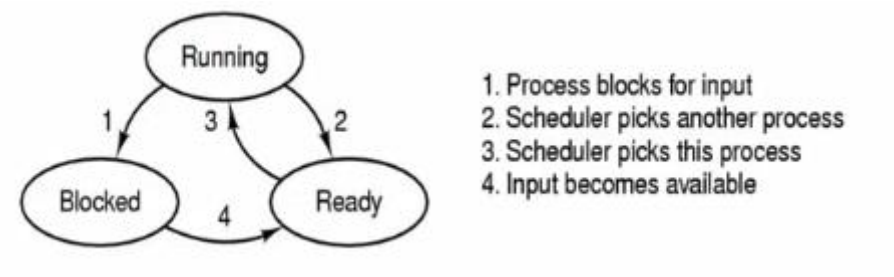
引起进程创建的 3 个主要事件：系统初始化、用户请求创建一个新进程、正在运行的进程调用了一个创建进程的系统调用。

**三态模型：**

1) 运行态：当一个进程正在处理机上运行

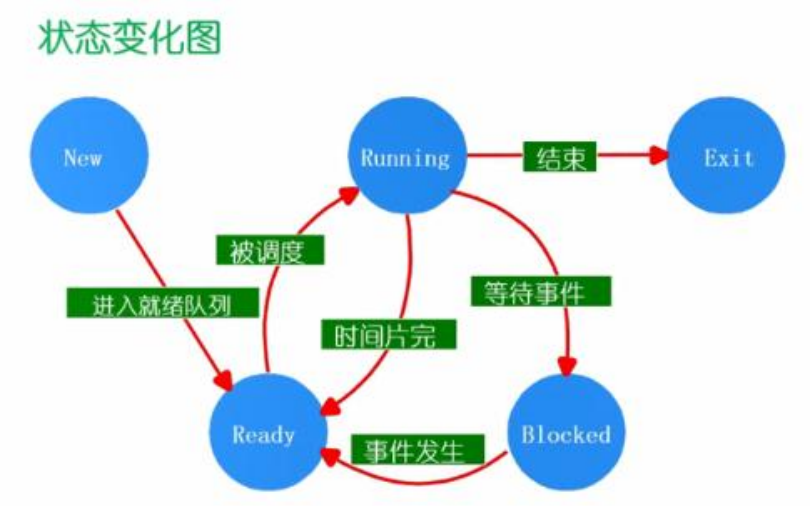
2) 就绪态：一个进程获得了除 CPU 之外的一切所需资源，一旦得到 CPU 即可运行

3) 阻塞态：一个进程正在等待某一事件而暂时停止运行。如等待输入输出完成。




进程的其他状态：

- 创建状态（new）：一个进程正在被创建，还没转到就绪态之前的状态；
- 结束状态（exit）：一个进程正在从系统中消失时的状态，这是因为进程结束或由于其他原因导致。



### 1.2.2. 进程挂起

进程在挂起状态时，意味着进程没有占用内存空间。处在挂起状态的进程映像在磁盘上。

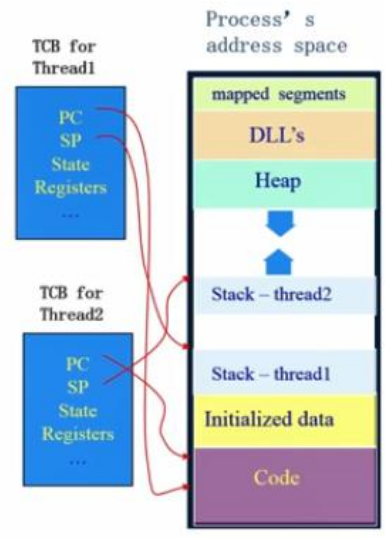
挂起状态：

- 阻塞挂起：进程在外存并等待某个事件的出现；
- 就绪挂起：进程在外存，但只要进入内存，即可运行；

挂起：把一个进程从内存转到外存，可能有以下几种情况：

1) 阻塞到阻塞挂起：没有进程处于就绪状态或者就绪进程要求更多的内存资源时，会进行这种转换，以提交新进程或运行就绪进程；
2) 就绪到就绪挂起：当有高优先权阻塞进程和低优先级就绪进程时，系统会选择挂起低优先级就绪进程；
3) 运行到就绪挂起：对抢占式分时系统，当有高优先级阻塞挂起进程因事件出现而进入就绪挂起时，系统可能会把运行进程转到就绪挂起状态。

## 1.3. 线程




进程中的一条执行流程，即线程。

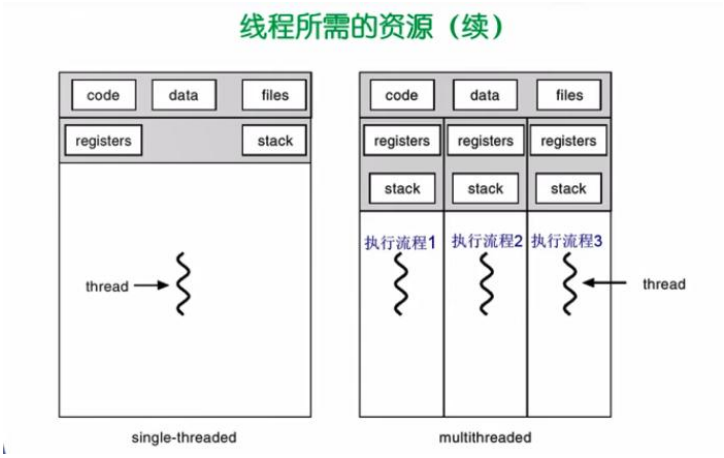
重新理解进程：

- 从资源组合的角度：进程把一组相关的资源组合起来，构成了一个资源平台，包括地址空间（代码段、数据段）、打开的文件等各种资源；
- 从运行的角度：代码在这个资源平台上的一条执行流程。

线程的优点：

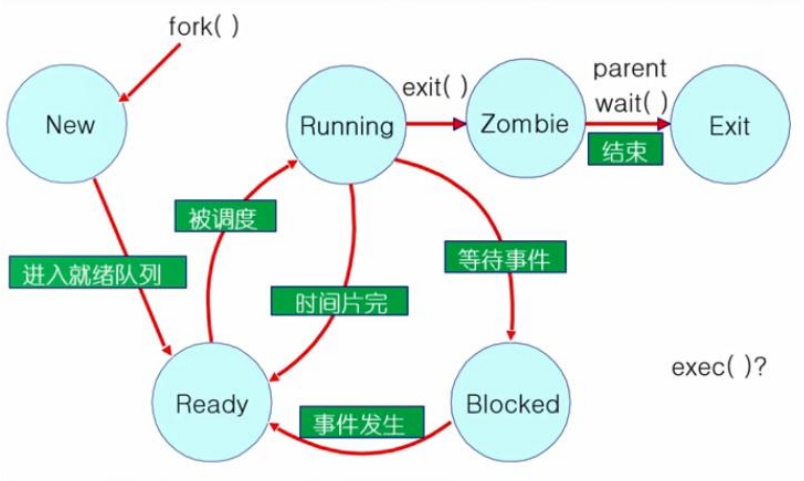
1) 一个进程中可以同时存在多个线程；
2) 各个线程之间可以并发执行；
3) 各个线程之间可以共享地址空间和文件等资源。

缺点：一个线程崩溃，会导致其所属进程的所有线程崩溃。




线程和进程的比较：

1) 进程是资源分配的单位，线程是 CPU 调度单位；
2) 进程拥有一个完整的资源平台，线程只独享必不可少的资源，如寄存器和栈；
3) 线程同样具有就绪、阻塞和执行三种状态，同样具有状态间的转换关系；
4) 线程能够减少并发执行的时间和空间开销：
   a) 线程的创建时间比进程短；
   b) 线程的终止时间比进程短；（不需要释放资源）
   c) 同一进程内的线程切换时间比进程短；（不需要切换地址空间和页表）
   d) 由于同一进程的各线程间共享内存和文件资源，可直接进行不通过内核的通信。




## 1.4. 进程与线程的区别

根本区别：进程是操作系统资源分配的基本单位，而线程是任务调度和执行的基本单位

1) **在开销方面**：每个进程都有独立的代码和数据空间（程序上下文），程序之间的切换会有较大的开销；线程可以看做轻量级的进程，同一类线程共享代码和数据空间，每个线程都有自己独立的运行栈和程序计数器（PC），线程之间切换的开销小。

2) **所处环境：**在操作系统中能同时运行多个进程（程序）；而在同一个进程（程序）中有多个线程同时执行（通过 CPU 调度，在每个时间片中只有一个线程执行）

3) **内存分配方面：**系统在运行的时候会为每个进程分配不同的内存空间；而对线程而言，除了 CPU 外，系统不会为线程分配内存（线程所使用的资源来自其所属进程的资源），线程组之间只能共享资源。

4) **包含关系：**没有线程的进程可以看做是单线程的，如果一个进程内有多个线程，则执行过程不是一条线的，而是多条线（线程）共同完成的；线程是进程的一部分，所以线程也被称为轻权进程或者轻量级进程。

## 1.5. 协程

以多线程实现生产者/消费者模式为例，却并不是一个高性能的实现。为什么性能不高呢？原因如下：

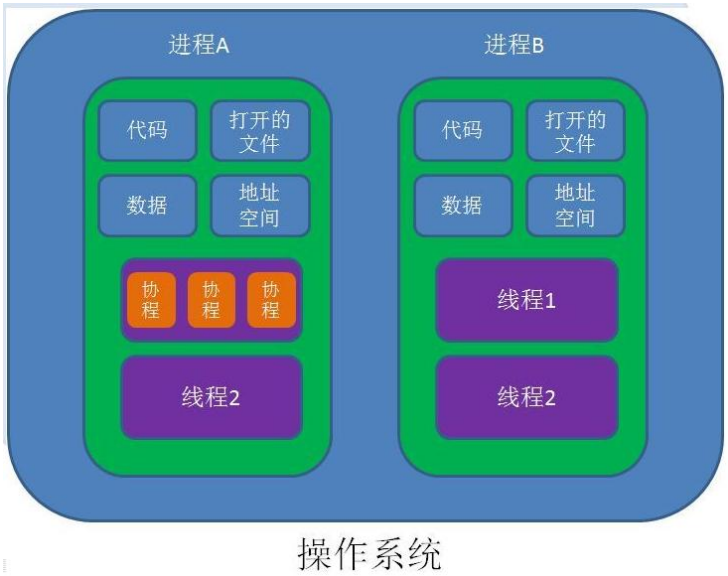
1.涉及到同步锁。

2.涉及到线程阻塞状态和可运行状态之间的切换。

3.涉及到线程上下文的切换。

以上涉及到的任何一点，都是非常耗费性能的操作。

协程，英文 Coroutines，是**一种比线程更加轻量级的存在**。正如一个进程可以拥有多个线程一样，**一个线程也可以拥有多个协程**。




最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制（也就是在用户态执行）。这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。**协程的暂停完全由程序控制**，线程的阻塞状态是由操作系统内核来进行切换。因此，协程的开

销远远小于线程的开销。

# 2. 进程间的通信方式

## 2.1. 管道（pipe）

### 2.1.1. 管道特点

- 1.管道只允许具有血缘关系的进程间通信，如父子进程间的通信。

- 2.管道只允许**单向通信**。

- 3.管道内部保证同步机制，从而保证访问数据的一致性。

- 4.面向字节流

-5.管道随进程，进程在管道在，进程消失管道对应的端口也关闭，两个进程都消失管道也消失。

### 2.1.2. 管道的原理和实现

**原理：**管道实为内核使用**环形队列机制**（以便管道可以重复利用），借助内核缓冲区（4k）实现

**本质：**是一个伪文件（实为内核缓冲区），由两个文件描述符引用，一个表示读端，一个表示写端。数据从管道的写端流入，从读端流出。

**管道缓冲区大小**

```
//命令查看当前系统中创建管道文件所对应的内核缓冲区的大小
ulimit -a
```

### 2.1.3. 管道的读写行为

#### 1).读管道

1.管道中有数据，read 返回实际读到的字节数。

2.管道中无数据：

（1）管道写端被全部关闭，read 返回 0

（2）管道写端没有全部被关闭，read 阻塞等待（等待管道中有数据可读）

#### 2).写管道

1.管道读端全部被关闭，进程异常终止。

2.管道读端没有全部关闭：

（1）管道已满，write 阻塞。

（2）管道未满，write 将数据写入，并返回实际写入的字节数。

## 2.2. 命名管道

有名管道也叫命名管道，在文件系统目录中存在一个管道文件。管道文件仅仅是文件系统中的标示，**并不在磁盘上占据空间**。在使用时，**在内存上开辟空间**，作为两个进程数据交互的通道。

### 2.2.1. 管道文件的创建：

1) 在 shell 中使用 mkfifo 命令

```
mkfifo filename
```

2) mkfifo 函数 (在代码中使用其创建管道文件)

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *filename,mode_t mode);
```

### 2.2.2. 使用方式：

1）使用 open 函数打开管道文件

如果一个进程以只读(只写)打开，那么这个进程会被阻塞到 open，直到另一个进程以只写(只读)或者读写。

2）使用 read 函数读取内容

read 读取普通文件，read 不会阻塞。而 read 读取管道文件，read 会阻塞运行，直到管道中有数据或者所有的写端关闭。

3）使用 write 函数发送内容，使用 close 函数关闭打开的文件。

## 2.3. 信号（signal）

信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生；

## 2.4. 消息队列

消息队列是消息的链接表，它克服了上两种通信方式中信号量有限的缺点，具有写权限得进程可以按照一定得规则向消息队列中添加新信息；对消息队列有读权限得进程则可以从

消息队列中读取信息；

## 2.5. 共享内存

可以说这是最有用的进程间通信方式。它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据得更新。这种方式需要依靠某种同步操作，如互斥锁和信号量等；

## 2.6. 信号量

主要作为进程之间及同一种进程的不同线程之间得同步和互斥手段

## 2.7. 套接字

这是一种更为一般得进程间通信机制，它可用于网络中不同机器之间的进程间通信，应用非常广泛。

# 3. 进程调度

## 3.1. 先来先服务

先来先服务(FCFS)调度算法是一种最简单的调度算法，该算法既可用于作业调度，也可用于进程调度。当在作业调度中采用该算法时，每次调度都是从后备作业队列中选择一个或多个最先进入该队列的作业，将它们调入内存，为它们分配资源、创建进程，然后放入就绪队列。在进程调度中采用 FCFS 算法时，则每次调度是从就绪队列中选择一个最先进入该队列的进程，为之分配处理机，使之投入运行。该进程一直运行到完成或发生某事件而阻塞后才放弃处理机。

## 3.2. 短作业(进程)优先调度算法

短作业(进程)优先调度算法 SJ(P)F，是指对短作业或短进程优先调度的算法。它们可以分别用于作业调度和进程调度。短作业优先(SJF)的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，将它们调入内存运行。而短进程优先(SPF)调度算法则是从就绪队列中选出一个估计运行时间最短的进程，将处理机分配给它，使它立即执行并一直执行到完成，或发生某事件而被阻塞放弃处理机时再重新调度。

## 3.3. 高优先权优先调度算法

为了照顾紧迫型作业，使之在进入系统后便获得优先处理，引入了最高优先权优先(FPF)调度算法。此算法常被用于批处理系统中，作为作业调度算法，也作为多种操作系统中的进程调度算法，还可用于实时系统中。当把该算法用于作业调度时，系统将从后备队列中选择若干个优先权最高的作业装入内存。当用于进程调度时，该算法是把处理机分配给就绪队列中优先权最高的进程，这时，又可进一步把该算法分成如下两种。

### 3.3.1. 非抢占式优先权算法

在这种方式下，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，该进程便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时，系统方可再将处理机重新分配给另一优先权最高的进程。这种调度算法主要用于批处理系统中；也可用于某些对实时性要求不严的实时系统中。

### 3.3.2. 抢占式优先权调度算法

在这种方式下，系统同样是把处理机分配给优先权最高的进程，使之执行。但在其执行期间，只要又出现了另一个其优先权更高的进程，进程调度程序就立即停止当前进程(原优先权最高的进程)的执行，重新将处理机分配给新到的优先权最高的进程。因此，在采用这种调度算法时，是每当系统中出现一个新的就绪进程 i 时，就将其优先权 Pi 与正在执行的进程 j 的优先权 Pj 进行比较。如果 Pi≤Pj，原进程 Pj 便继续执行；但如果是 Pi>Pj，则立即停止 Pj 的执行，做进程切换，使 i 进程投入执行。显然，这种抢占式的优先权调度算法能更好地满足紧迫作业的要求，故而常用于要求比较严格的实时系统中，以及对性能要求较高的批处理和分时系统中。

### 3.3.3. 优先级反转

优先级反转是指一个低优先级的任务持有一个被高优先级任务所需要的共享资源。高优先任务由于因资源缺乏而处于受阻状态，一直等到低优先级任务释放资源为止。而低优先级获得的 CPU 时间少，如果此时有优先级处于两者之间的任务，并且不需要那个共享资源，则该中优先级的任务反而超过这两个任务而获得 CPU 时间。如果高优先级等待资源时不是阻塞等待，而是忙循环，则可能永远无法获得资源，因为此时低优先级进程无法与高优先级进程争夺 CPU 时间，从而无法执行，进而无法释放资源，造成的后果就是高优先级任务无法获得资源而继续推进。

### 3.3.4. 优先级反转案例解释

不同优先级线程对共享资源的访问的同步机制。优先级为高和低的线程 tall 和线程 low 需要访问共享资源，优先级为中等的线程 mid 不访问该共享资源。当 low 正在访问共享资源时，tall 等待该共享资源的互斥锁，但是此时 low 被 mid 抢先了，导致 mid 运行 tall 阻塞。即优先级低的线程 mid 运行，优先级高的 tall 被阻塞。

### 3.3.5. 优先级倒置解决方案

(3.3.5.1）**设置优先级上限**：给临界区一个高优先级，进入临界区的进程都将获得这个高优先级，如果其他试图进入临界区的进程的优先级都低于这个高优先级，那么优先级反转就不会发生。

（3.3.5.2）**优先级继承**：当一个高优先级进程等待一个低优先级进程持有的资源时，低优先级进程将暂时获得高优先级进程的优先级别，在释放共享资源后，低优先级进程回到原来的优先级别。嵌入式系统 VxWorks 就是采用这种策略。

## 3.4. 时间片轮转法

在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把 CPU 分配给队首进程，并令其执行一个时间片。时间片的大小从几 ms 到几百 ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在给定的时间内均能获得一时间片的处理机执行时间。换言之，系统能在给定的时间内响应所有用户的请求。

## 3.5. 高响应比优先调度算法 HRRN

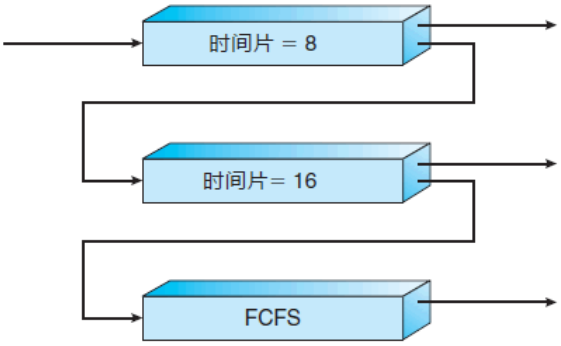
高响应比优先调度算法（Highest Response Ratio Next）是一种对 CPU 中央控制器响应比的分配的一种算法。HRRN 是介于 FCFS（先来先服务算法）与 SJF（短作业优先算法）之间的折中算法，既考虑作业等待时间又考虑作业运行时间，既照顾短作业又不使长作业等待时间过长，改进了调度性能。

高响应比优先调度算法既考虑作业的执行时间也考虑作业的等待时间，综合了先来先服务和最短作业优先两种算法的特点。该算法中的响应比是指作业等待时间与运行比值：

响应比 =（等待时间+要求服务时间）/ 要求服务时间

## 3.6. 多级反馈队列调度算法

前面介绍的各种用作进程调度的算法都有一定的局限性。如短进程优先的调度算法，仅照顾了短进程而忽略了长进程，而且如果并未指明进程的长度，则短进程优先和基于进程长度的抢占式调度算法都将无法使用。而多级反馈队列调度算法则不必事先知道各种进程所需的执行时间，而且还可以满足各种类型进程的需要，因而它是目前被公认的一种较好的进程调度算法。在采用多级反馈队列调度算法的系统中，调度算法的实施过程如下所述。




1) 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，第二个队列次之，其余各队列的优先权逐个降低。该算法赋予各个队列中进程执行时间片的大小也各不相同，在优先权愈高的队列中，为每个进程所规定的执行时间片就愈小。例如，第二个队列的时间片要比第一个队列的时间片长一倍，……，第 i+1 个队列的时间片要比第 i 个队列的时间片长一倍。

2) 当一个新进程进入内存后，首先将它放入第一队列的末尾，按 FCFS 原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二队列的末尾，再同样地按 FCFS 原则等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，如此下去，当一个长作业(进程)从第一队列依次降到第 n 队列后，在第 n 队列便采取按时间片轮转的方式运行。

3) 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行；仅当第 1～(i-1)队列均空时，才会调度第 i 队列中的进程运行。如果处理机正在第 i 队列中为某进程服务时，又有新进程进入优先权较高的队列(第 1～(i-1)中的任何一个队列)，则此时新进程将抢占正在运行进程的处理机，即由调度程序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权进程。

## 3. 线程同步的方式

1) **互斥量 Synchronized/Lock**：采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以可以保证公共资源不会被多个线程同时访问
2) **信号量 Semphare**：它允许同一时刻多个线程访问同一资源，但是需要控制同一时刻访问此资源的最大线程数量
3) **事件(信号)，Wait/Notify**：通过通知操作的方式来保持多线程同步，还可以方便的实现多线程优先级的比较操作

## 4. 并发性：互斥与同步

## 4.1. 并发的原理

多道程序设计系统中，进程被交替执行，表现出一种并发执行的外部特征。多道程序设计系统中，进程的相对执行速度不可预测，它取决于其他进程的活动、操作系统处理中断的方式及操作系统的调度策略。在全局资源共享、操作系统资源最优分配和定位程序设计错误方面引入了困难。

### 竞争条件(Race Condition)

**多个进程或者线程读写数据时，执行结果依赖于多个进程的指令执行顺序**。

### 原子操作

一次不存在任何中断或失败的执行。该执行成功结束或者根本没被执行，并且不应该发现任何部分执行的状态。

### 临界区

进程中的一段需要访问共享资源并且当另一个进程处于相应代码区域时不会被执行的代码区域。

### 互斥

当一个进程处于临界区并访问共享资源时，没有其他进程会处于临界区并且访问任何相同的共享资源。

**死锁**

两个或以上的进程，在相互等待完成特定任务，而最终没法将自身任务进行下去。

**饥饿**

一个可执行的进程被调度器持续忽略，以至于虽处于可执行状态却不被执行。

## 4.2. 临界区

特点：

1) 互斥：同一时间中，临界区中最多存在一个线程。
2) 前进：如果一个线程想进入临界区，那么它最终会成功。
3) 有限等待：如果一个线程 i 处于入口区，那么在 i 的请求被接受之前，其他线程进入临界区的时间时有限的。
4) 无忙等待：如果一个线程在等待进入临界区，那么它可以在进入之前被挂起。

### 4.2.1. 方法一：禁用硬件中断

为保证互斥，只需要保证一个进程不被中断即可。没有中断，即没有上下文切换，因此没有并发，硬件将中断处理延迟到中断被启用之后。进入临界区时关闭中断，离开临界区时开启中断。

### 4.2.2. 方法二：专用机器指令

硬件级别上，对存储单元的访问排斥对相同单元的其他访问。基于此点，处理器的设计者提供了一些机器指令，用于保证两个动作的原子性。在该指令执行过程中，任何其他指令访问内存将被阻止，而且这些动作在一个指令周期中完成。

1) **比较和交换指令：compare_and_swap**， 使用一个测试值 testval 检查一个内存单元，如果该内存单元的当前值是 testval，就用 newval 取代该值；否则保持不变。该指令总是返回旧内存值，因此，如果返回值与测试值相同，则表示该内存单元已被更新。
2) **交换指令：exchange**，交换一个寄存器和一个存储器的内容。

机器指令方法优点：适用于在单处理器或共享内存的多处理器上的任何数目的进程；简单且易于证明；可用于支持多个临界区；

缺点：使用了忙等待，当一个进程等待进去临界区时，会耗费 CPU 资源；可能饥饿，因为选择哪个进程进入临界区是任意的；可能死锁。

### 4.2.3. 基于软件的解决方案

## 4.3. 信号量

两个或多个进程可以通过简单的信号进行合作，一个进程可以被迫在某一位置停止，直到它接收到一个特定的信号。为了发信号，需要使用一个称为信号量的特殊变量。为通过信号量 s 传送信号，进程可执行原语 semSignal(s)；为通过信号量 s 接收信号，进程可执行原语 semWait(s)；如果相应的信号仍然没有发送，则进程被阻塞，直到发送完为止。

可把信号量视为一个具有整数值的变量，并定义三个操作：

1) 一个信号量可以初始化为非负数；
2) semWait 操作使信号量减 1，若值为负数，则执行 semWait 的进程被阻塞。否则进程继续执行；
3) semSignal 操作使信号量加 1，若值小于或等于 0，则被 semWait 操作阻塞的进程被解除阻塞。

### 4.3.1. 互斥

使用二元信号量来实现互斥，信号量一般被初始化为 1。进程进入临界区之前执行 semWait(s)，若 s 为负，则进程被阻塞。

```
const int n = number_of_process; // 进程数量
semaphore s = 1;
void P(int i)
{
    while(true)
    {
        semWait(s);

        // 临界区

        semSignal(s);

        // other code
    }
}
```

### 4.3.2. 生产者消费者

一个或多个生产者产生数据将数据放在一个缓冲区里；单个消费者每次从缓冲区中取出数据；在任何一个时间内只有一个生产者或小费制可以访问该缓冲区。

正确性要求：

1) 在任何一个时间内只有一个线程操作 Buffer（互斥）； → 二值信号量

2) 当 Buffer 为空，消费者必须等待生产者（同步）； → Buffer 为空，一般信号量

3) 当 Buffer 为满，生产者必须等待消费者（同步）； → Buffer 为满，一般信号量

针对上面每个约束，使用一个单独的信号量，如上所示。

```
class BounderBuffer{
    mutex = new Semaphore(1);       // 互斥：二元信号量
    fullBuffer = new Semaphore(0);  // 一开始，buffer为空
    emptyBuffer = new Semaphore(n); // 当前生产者可往Buffer写n个单位数据，或可对Buffer进入n次
};

// 生产者
BoundedBuffer:: produce(c)
{
    emptyBuffer->P();
    /***  互斥 ***/
    mutex->P();
    add c to buffer;
    mutex->V();
    /***  互斥 ***/
    fullBuffer->V(); // 增加了内容，信号量+1
}

// 消费者
BoundedBuffer::Remove(c)
{
    fullBuffer->P(); // 减少了内容，信号量-1
    /***  互斥 ***/
    mutex->P();
    remove c from buffer;
    mutex->V();
    /***  互斥 ***/
    emptyBuffer->V();
}
```

## 4.4. 管程

管程是一个程序设计语言结构，它提供了与信号量同样的功能，但更易于控制。管程是一个或多个过程、一个初始化序列和局部数据组成的软件模块，其主要特点如下：

1) 局部数据变量只能被管程的过程访问，任何外部过程都不能访问；

2) 一个进程通过调用管程的一个过程进入管程；

3) 在任何时候，只有一个进程在管程中执行，调用管程的任何其他进程都被阻塞，以等待管程可用。

因此，可以把共享数据结构放在管程中，由管程提供的互斥机制进行保护。为进行并发处理，管程通过使用条件变量提供对同步的支持，这些条件变量包含在管程中，并且只有在管程中才能被访问。有两个函数可以操作条件变量：

1) cwait(c)：调用进程的执行在条件 c 上阻塞，管程现在可被另一个进程使用；

2) csignal(c)：恢复执行在 cwait 后因为某些条件而阻塞的进程。如果有多个这样的进程，选择其中一个；如果没有这样的进程，什么也不做。→ 如果没有进程在条件

x 上等待，则 csignal(x)的执行不会产生任何效果。

## 4.5. 消息传递

# 5. 死锁

## 5.1. 死锁原理

可以把死锁定义为一组相互竞争系统资源或进行通信的进程间的永久阻塞。当一组进程中的每个进程都在等待某个事件（典型的情况是等待所请求的资源被释放），而只有在这组进程中的其他被阻塞的进程才可以触发该事件，这时就称这组进程发生死锁。

### 5.1.1. 死锁条件

1) **互斥**：依次只有一个进程可以使用一个资源，其他进程不能访问已分配给其他进程的资源；
2) **占有且等待**：当一个进程等待其他进程时，继续占有已经分配的资源；
3) **不可抢占**：不能强行抢占进程已占有的资源；
4) **循环等待**：存在一个封闭的进程链，使得每个进程至少占有此链中下一个进程所需要的一个资源。

前三个是死锁存在的**必要条件**，而不是充分条件，死锁产生后一定有前 3 个现象。4 个条件一起，称为死锁的充分必要条件。

## 5.2. 死锁预防

死锁预防策略是试图设计一种系统来排除发生死锁的可能性，防止前面列出的 4 个条件中的任何一个发生。

### 5.2.1. 互斥

不可能禁止这个条件，如果要对资源进行互斥访问，则操作系统必须支持互斥。

### 5.2.2. 占有且等待

可以要求进程一次性地请求所有需要的资源，并且阻塞这个进程直到所有请求都同时满足。该方法是低效的：

1) 进程可能被阻塞很长时间，以等待所有请求被满足；
2) 分配给一个进程的资源可能有相当长的一段时间不会被使用，但在此期间它们又不能被

其他进程使用；

3) 进程可能实现并不知道它所需要的所有资源。

### 5.2.3. 不可抢占

两种方法：

1) 如果一个进程请求当前被另一个进程占有的一个资源，则操作系统可以抢占另一个进程，要求它释放资源。

2) 如果占有某些资源的一个进程进一步申请资源时被拒绝，则该进程必须释放它最初占有的资源。

只有在资源状态可以很容易地保存和恢复的情况下，这种方法才是实用的。

### 5.2.4. 循环等待

可以通过定义资源类型的线性访问顺序来预防。如果一个进程已经分配到了 R 类型的资源，那么它接下来请求的资源只能是那些排在 R 类型之后的资源类型。

## 5.3. 死锁避免

通过约束资源请求，使得 4 个死锁条件中至少有一个被破坏。死锁避免需要知道将来的进程资源请求的情况。两种方法：

1) **进程启动拒绝：**如果一个进程的请求会导致死锁，则不启动进程；

2) **资源分配拒绝：**如果一个进程增加的资源请求会导致死锁，则不允许此分配。此即银行家算法

死锁避免的优点：不需要死锁预防中的抢占和回滚进程，并且比死锁预防的限制少。

限制：

1) 必须事先声明每个进程请求的最大资源；

2) 所讨论的进程必须是无关的，即他们的执行顺序没有任何同步的限制；

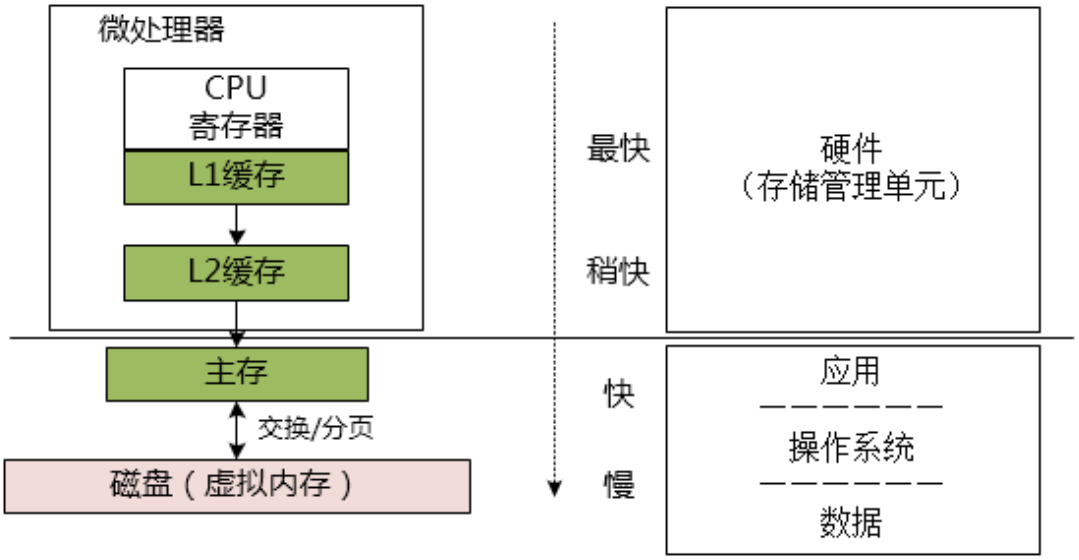
3) 分配的资源数目必须是固定的；

4) 在占有资源时，进程不能退出。

## 5.4. 死锁检测

# 6. 内存管理与虚拟内存

## 0. 内存分层体系




## 1. 内存管理的需求

**1. 重定位：**操作系统中可用的内存空间被多个进程共享，当一个进程被换出磁盘再换入内存时，不必将其放在被换出前相同的位置（否则会是很大的限制）。所以操作系统需要把进程重定位到内存的不同区域。处理器硬件和操作系统软件必须能够通过内存管理把程序中代码的内存访问转换成实际的物理内存地址，并反映程序在内存中的当前位置。

**2. 保护：**每个进程都受到保护，一个进程的程序不能访问其他进程的数据区。

**3. 共享：**保护机制必须具有一定灵活性，以允许多个继承访问内存的同一部分。

**4. 逻辑组织：**程序不必考虑底层细节，连续的逻辑地址空间

**5. 物理组织：**存储器分为两级，内存和外存，根据容量和存储、访问需求分别存储。

## 2. 内存分区

### 2.1. 固定分区

在系统生成阶段，内存被划分为许多静态分区。进程可以被装入到大于或等于自身大小的分钟。优点：实现简单。缺点：程序可能太大而不能放到一个分区中，有内存碎片（被装

入的数据块小于分区的大小），内存利用率低。

### 2.2. 动态分区

分区是动态创建的，因而使得每个进程可以被装入到与自身大小正好相等的分区中。缺点：随着时间推移，内存中会出现许多小的空洞，即**外部碎片**（分配的单间之间的未使用内存），内存的利用率随之下降。

解决外部碎片的办法：

**1. 压缩技术：**操作系统不时移动进程，使进程之间连续占用内存，所有已用内存连成一片，所有空闲内存连成一片。但压缩过程耗时，浪费处理时间，而且需要动态重定位能力。

**2. 放置算法：**操作系统利用放置算法，根据不同策略将进程分配到内存中。

1) 最佳适配：选择与要求的大小最接近的块；

2) 首次适配：从开始扫描内存，选择大小足够的第一块；

3) 下次适配：从上一次放置的位置开始扫描内存，选择下一个足够大小的块。

## 3. 分页与分段

### 3.1. 分页

固定分区和动态分区都会产生内存碎片，内存利用率极低。分页技术是把内存划分为大小固定相等的**页框**（**frame**），进程中称为**页**（**page**）的块可以指定到内存中称为**页框**的可用块。每个进程被划分成许多大小与页框相等的页。要装入一个进程，需要把进程包含的所有页都装入内存内不一定连续的某些页框中。

进程的所有页不一定会加载到内存的连续的页框中。操作系统为每个进程维护了一个**页表（page table）**。页表给出了进程中每一页对应的页框位置。划分物理内存为固定大小的frame，划分逻辑地址空间至相同大小的页。

一个逻辑地址是一个二元组(p,offset)，p 代表页号，offset 是偏移。若逻辑地址中使用 s 个 bit 来表示页号，那么逻辑地址(f,offset)表示的虚拟地址是 2^s*p+offset.

一个物理地址是一个二元组(f,offset)，f 代表页框号，offset 是偏移。物理地址(f,offset)表示的物理地址是：2^s*f+offset.

一个逻辑地址 addr 转换为物理地址的过程为：

1) 根据逻辑地址的前 s 位得到页号 p，根据后几位计算出偏移 offset；

2) 查找页表（由 OS 建立），找到该进程页对应的页框号 f；

3) 由 f*2^s+offset 得到物理内存地址。

## 3.2. 分段

采用分段技术，可以把程序和其相关的数据划分到几个**段（segment）**中。采用分段技术时，**逻辑地址由段号和偏移量组成**。段有一个最大长度限制，但并不要求所有程序的所有段的长度都相等。

每个进程都有一个段表。在分段方案中，逻辑地址转换为物理地址的过程与分页方案类似。

### 面试题：分页和分段有什么区别（内存管理）？

段式存储管理是一种符合用户视角的内存分配管理方案。在段式存储管理中，将程序的地址空间划分为若干段（segment），如代码段，数据段，堆栈段；这样每个进程有一个二维地址空间，相互独立，互不干扰。**段式管理的优点是：没有内碎片（因为段大小可变，改变段大小来消除内碎片）。但段换入换出时，会产生外碎片**。

页式存储管理方案是一种用户视角内存与物理内存相分离的内存分配管理方案。在页式存储管理中，将程序的逻辑地址划分为固定大小的页（page），而物理内存划分为同样大小的帧，程序加载时，可以将任意一页放入内存中任意一个帧，这些帧不必连续，从而实现了离散分离。页式存储管理的优点是：**没有外碎片（因为页的大小固定），但会产生内碎片**（一个页可能填充不满）。

两者的不同点：

1) **目的不同**：分页是由于系统管理的需要而不是用户的需要，它是信息的物理单位；分段的目的是为了能更好地满足用户的需要，它是信息的逻辑单位，它含有一组其意义相对完整的信息；
2) **大小不同**：页的大小固定且由系统决定，而段的长度却不固定，由其所完成的功能决定；
3) **地址空间不同**： 段向用户提供二维地址空间；页向用户提供的是一维地址空间；
4) **信息共享**：段是信息的逻辑单位，便于存储保护和信息的共享，页的保护和共享受到限制；
5) **内存碎片**：页式存储管理的优点是没有外碎片（因为页的大小固定），但会产生内碎片（一个页可能填充不满）；而段式管理的优点是没有内碎片（因为段大小可变，改变段大小来消除内碎片）。但段换入换出时，会产生外碎片（比如 4k 的段换 5k 的段，会产生 1k 的外碎片）。

# 4. 虚拟内存

进程中所有内存访问都是逻辑地址，这些逻辑地址在运行时动态地被换成物理地址。这意味着一个进程可以被换入换出内存，使得进程可以在执行过程中的不同时刻占据内存中的不同区域。同时，一个进程可以被分为很多块（页和段）。在执行过程中，这些块不需要连续地位于内存中。内存中保存着要取的下一条指令所在的块或页，以及将要访问的下一个数据单元的所在块，就可以保持程序暂时继续下去。

虚拟内存提供了三个重要的能力：

1) 它将主存看成是一个存储在磁盘上的地址空间的高速缓存，在主存中只保存活动区域，并根据需要在磁盘和主存之间来回传送数据，通过这种方式，高效使用内存。
2) 它为每个进程提供了一致的地址空间，从而简化内存管理。
3) 它保护了每个进程的地址空间不被其他进程破坏。

## 4.1. 局部性原理

局部性原理表现在以下两个方面：

- **时间局部性：**如果程序中的某条指令一旦执行，不久以后该指令可能再次执行；如果某数据被访问过，不久以后该数据可能再次被访问。产生时间局部性的典型原因，是由于在程序中存在着大量的循环操作。
- **空间局部性：**一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也将被访问，即程序在一段时间内所访问的地址，可能集中在一定的范围之内，这是因为指令通常是顺序存放、顺序执行的，数据也一般是以向量、数组、表等形式簇聚存储的。

时间局部性是通过将近来使用的指令和数据保存到高速缓存存储器中，并使用高速缓存的层次结构实现。空间局部性通常是使用较大的高速缓存，并将预取机制集成到高速缓存控制逻辑中实现。虚拟内存技术实际上就是建立了“内存一外存”的两级存储器的结构，利用局部性原理实现高速缓存。

## 4.2. 虚拟内存技术的实现

虚拟内存中，允许将一个作业分多次调入内存，虚拟内存的实现需要建立在离散分配的内存管理方式的基础上。虚拟内存的实现有以下三种方式：请求分页存储管理、请求分段存储管理、请求段页式存储管理。

不管哪种方式，都需要有一定的硬件支持。一般需要的支持有以下几个方面：

- 一定容量的内存和外存。
- 页表机制（或段表机制），作为主要的数据结构。
- 中断机构，当用户程序要访问的部分尚未调入内存，则产生中断。
- 地址变换机构，逻辑地址到物理地址的变换。

## 4.3. 请求分页管理方式实现虚拟内存

请求分页系统建立在基本分页系统基础之上，为了支持虚拟存储器功能而增加了请求调页功能和页面置换功能。请求分页是目前最常用的一种实现虚拟存储器的方法。

在请求分页系统中，只要求将当前需要的一部分页面装入内存，便可以启动作业运行。在作业执行过程中，当所要访问的页面不在内存时，再通过调页功能将其调入，同时还可以通过置换功能将暂时不用的页面换出到外存上，以便腾出内存空间。

为了实现请求分页，系统必须提供一定的硬件支持。除了需要一定容量的内存及外存的计算机系统，还需要有**页表机制**、**缺页中断机构**和**地址变换机构**。

### 4.3.1. 页表机制

请求分页系统的页表机制不同于基本分页系统，请求分页系统在一个作业运行之前不要求全部一次性调入内存，因此在作业的运行过程中，必然会出现要访问的页面不在内存的情况，如何发现和处理这种情况是请求分页系统必须解决的两个基本问题。为此，在请求页表项中增加了四个字段：

1) **驻留位 P**：用于指示该页是否已调入内存，1-内存，0-外存。
2) **访问位 A**：用于记录本页在一段时间内被访问的次数，或记录本页最近已有多长时间未被访问，供置换算法换出页面时参考。
3) **修改位 M**：标识该页在调入内存后是否被修改过。
4) **外存地址**：用于指出该页在外存上的地址，通常是物理块号，供调入该页时参考。

### 4.3.2. 缺页中断机构

在请求分页系统中，每当所要访问的页面不在内存时，便产生一个缺页中断，请求操作系统将所缺的页调入内存。此时应将缺页的进程阻塞（调页完成唤醒)，如果内存中有空闲块，则分配一个块，将要调入的页装入该块，并修改页表中相应页表项，若此时内存中没有空闲块，则要淘汰某页（若被淘汰页在内存期间被修改过，则要将其写回外存)。

缺页中断作为中断同样要经历，诸如保护 CPU 环境、分析中断原因、转入缺页中断处

理程序、恢复 CPU 环境等几个步骤。但与一般的中断相比，它有以下两个明显的区别：

1) 在指令执行期间产生和处理中断信号，而非一条指令执行完后，属于内部中断。
2) 一条指令在执行期间，可能产生多次缺页中断。

#### 4.3.3. 地址变换机构

在进行地址变换时，先检索快表：

1) 若找到要访问的页，便修改页表项中的访问位（写指令则还须重置修改位)，然后利用页表项中给出的物理块号和页内地址形成物理地址。
2) 若未找到该页的页表项，应到内存中去查找页表，再对比页表项中的状态位 P，看该页是否已调入内存，未调入则产生缺页中断，请求从外存把该页调入内存。

### 4.4. TLB

TLB - translation lookaside buffer，快表，直译为旁路快表缓冲，也可以理解为页表缓冲，**地址变换高速缓存**，**缓存近期访问的页帧转换表项**。由于页表存放在主存中，因此程序每次访存至少需要两次：一次访存获取物理地址，第二次访存才获得数据。提高访存性能的关键在于依靠页表的访问局部性。当一个转换的虚拟页号被使用时，它可能在不久的将来再次被使用到。内存管理硬件使用 TLB 来改善虚拟地址到物理地址的转换速度。使用 TLB 内核可以快速的找到虚拟地址指向物理地址，而不需要请求 RAM 内存获取虚拟地址到物理地址的映射关系。

当 CPU 要访问一个虚拟地址/线性地址时，首先根据虚拟地址的高 s 位(页号)在 TLB 中查找。如果是表中没有相应的表项，称为 TLB miss，需要通过访问慢速 RAM 中的页表计算出相应的物理地址。同时，物理地址被存加入到 TLB 表项中，以后对同一线性地址的访问，直接从 TLB 表项中获取物理地址即可，称为 TLB hit（TLB 命中）。

### 4.5. 局部页面置换算法

局部页面置换算法的目的：尽可能减少缺页中断次数

#### 4.5.0. Belady 现象

**描述：**在使用 FIFO 算法时，有时会出现分配的物理页面数增加，缺页率反而上升的异常现象。

**原因：**FIFO 算法的置换特征与进程的访问内存的动态特征是矛盾的，与置换算法的目标并不一致（即替换使用较少的页面）。

### 4.5.1. 最优页面置换算法

思路：当缺页中断发生，对于保存在内存中的每一个逻辑页面，计算在它下一次访问之前，还需要多长时间（实际上 OS 无从知道），从中选择等待时间最长的那个作为被置换的页面。

**这种算法是种理想情况，作为评价其他算法性能的依据。**

### 4.5.2. 先进先出算法 FIFO

思路：选择在内存中驻留时间最长的页面并淘汰。系统维护了一个链表，记录了所有位于内存当中的逻辑页面。从链表的排列顺序，链首页面驻留时间最长，链尾页面驻留时间最短。却也中断发生时，链首页面被交换，新页面被添加至链尾。

特点：性能差，调出的页面可能是经常访问的页面，且有 belady 现象。

### 4.5.3. 最近最久未使用算法 LRU (Least Recently Used)

思路：当一个缺页中断发生时，选择最久未使用的页面并淘汰。该算法是对最优页面置换算法的一个近似（依据历史预测未来），根据程序的局部性原理。

实现：需要记录各个页面的使用时间的先后顺序，开销较大。

- **链表实现**：最近刚使用的页面作为表头，最久未被访问的页面作为表尾。每次访问内存时，找到相应页面，从表中取出并放到表头。每次缺页中断时淘汰表尾。
- **栈实现**：访问某页时，页号压入栈。若栈内有与此页号相同的页，则抽出。缺页中断时，淘汰栈顶。

### 4.5.4. 最不常用算法 LFU (Least Frequently Used)

思路：缺页中断时，选择访问次数最少的页面并淘汰。需要为每个页面设置一个访问计数器。

### 4.5.5. 时钟页面置换算法 (Clock 算法)

1) 要用到页表项中的访问位（读写均为访问），该位初始化为 0，被访问时由硬件置 1；
2) 各个页面组成一个环形链表，类似时钟表面，把指针指向最老的页面（最先进来）；
3) 缺页中断时，考察指针指向的最老页面，若访问位为 0，立即淘汰。若访问位为 1，置为 0，指针往后移一格，如此下去，直到找到被淘汰的页面。

### 4.5.6. 二次机会算法 (Enhanced Clock Algorithm)

基于 Clock 算法，通过增加使用的位数目，可以使得 CLOCK 算法更加高效。在使用位的基础上再增加一个修改位，则得到改进型的 CLOCK 置换算法。这样，每一帧都处于以下

四种情况之一：

a) 最近未被访问，也未被修改(u=0, m=0)。

b) 最近被访问，但未被修改(u=1, m=0)。

c) 最近未被访问，但被修改(u=0, m=1)。

d) 最近被访问，被修改(u=1, m=1)。

算法执行如下操作步骤：

1) 从指针的当前位置开始，扫描帧缓冲区。在这次扫描过程中，对使用位不做任何修改。选择遇到的第一个帧(u=0, m=0)用于替换。

2) 如果第 1)步失败，则重新扫描，查找(u=0, m=1)的帧。选择遇到的第一个这样的帧用于替换。在这个扫描过程中，对每个跳过的帧，把它的使用位设置成 0。

3) 如果第 2)步失败，指针将回到它的最初位置，并且集合中所有帧的使用位均为 0。重复第 1 步，并且如果有必要，重复第 2 步。这样将可以找到供替换的帧。

## 4.6. 页面分配策略

### 4.6.1. 驻留集大小

对于分页式的虚拟内存，在准备执行时，不需要也不可能把一个进程的所有页都读取到主存，因此，操作系统必须决定读取多少页。也就是说，给特定的进程分配多大的主存空间，这需要考虑以下几点：

1) 分配给一个进程的存储量越小，在任何时候驻留在主存中的进程数就越多，从而可以提高处理机的时间利用效率。

2) 如果一个进程在主存中的页数过少，尽管有局部性原理，页错误率仍然会相对较高。

3) 如桌页数过多，由于局部性原理，给特定的进程分配更多的主存空间对该进程的错误率没有明显的影响。

基于这些因素，现代操作系统通常采用三种策略：

1) **固定分配局部置换：**它为每个进程分配一定数目的物理块，在整个运行期间都不改变。若进程在运行中发生缺页，则只能从该进程在内存中的页面中选出一页换出，然后再调入需要的页面。实现这种策略难以确定为每个进程应分配的物理块数目：太少会频繁出现缺页中断，太多又会使 CPU 和其他资源利用率下降。

2) **可变分配全局置换：**这是最易于实现的物理块分配和置换策略，为系统中的每个进程分配一定数目的物理块，操作系统自身也保持一个空闲物理块队列。当某进程发生缺页时，

系统从空闲物理块队列中取出一个物理块分配给该进程，并将欲调入的页装入其中。

3) **可变分配局部置换：**它为每个进程分配一定数目的物理块，当某进程发生缺页时，只允许从该进程在内存的页面中选出一页换出，这样就不会影响其他进程的运行。如果进程在运行中频繁地缺页，系统再为该进程分配若干物理块，直至该进程缺页率趋于适当程度； 反之，若进程在运行中缺页率特别低，则可适当减少分配给该进程的物理块。

### 4.6.2. 调入页面的时机

为确定系统将进程运行时所缺的页面调入内存的时机，可采取以下两种调页策略：

1) **预调页策略：**根据局部性原理，一次调入若干个相邻的页可能会比一次调入一页更高效。但如果调入的一批页面中大多数都未被访问，则又是低效的。所以就需要采用以预测为基础的预调页策略，将预计在不久之后便会被访问的页面预先调入内存。但目前预调页的成功率仅约 50%。故这种策略主要用于进程的首次调入时，由程序员指出应该先调入哪些页。

2) **请求调页策略：**进程在运行中需要访问的页面不在内存而提出请求，由系统将所需页面调入内存。由这种策略调入的页一定会被访问，且这种策略比较易于实现，故在目前的虚拟存储器中大多采用此策略。它的缺点在于每次只调入一页，调入调出页面数多时会花费过多的 I/O 开销。

### 4.6.3. 从何处调入页面

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。对换区通常是采用连续分配方式，而文件区采用离散分配方式，故对换区的磁盘 I/O 速度比文件区的更快。这样从何处调入页面有三种情况：

1) **系统拥有足够的对换区空间：**可以全部从对换区调入所需页面，以提高调页速度。为此，在进程运行前，需将与该进程有关的文件从文件区复制到对换区。

2) **系统缺少足够的对换区空间：**凡不会被修改的文件都直接从文件区调入；而当换出这些页面时，由于它们未被修改而不必再将它们换出。但对于那些可能被修改的部分，在将它们换出时须调到对换区，以后需要时再从对换区调入。

3) **UNIX 方式：**与进程有关的文件都放在文件区，故未运行过的页面，都应从文件区调入。曾经运行过但又被换出的页面，由于是被放在对换区，因此下次调入时应从对换区调入。进程请求的共享页面若被其他进程调入内存，则无需再从对换区调入。

### 4.7. 多级页表

假设虚拟地址空间为 32 位（4GB），每一页大小为 4KB，则进程需要的页表项数目为 4GB/4KB=1M。若每个页表项大小为 4B，则存储页表项总共需要 4M 内存。即使用一级页表时，页表占据 4M 内存空间。每个进程都要预先建立好 4M 的页表，进程数一多，内存中仅是页表项占据的内存空间已经足够大了。

每个进程都有 4GB 的虚拟地址空间，而显然对于大多数程序来说，其使用到的空间远未达到 4GB，何必去映射不可能用到的空间呢？也就是说，一级页表覆盖了整个 4GB 虚拟地址空间，但如果某个一级页表的页表项没有被用到，也就不需要创建这个页表项对应的二级页表了，即可以在需要时才创建二级页表。

二级页表由一个 4KB 的根页表和 1024 个 4K 的二级目录表项组成。对于一个 32 位的虚拟地址，高 10 位指出二级目录表项的地址，根据该地址定位到某个二级目录表，根据二级目录表中的 Frame Number 和 32 位虚拟地址的 offset 可以转换为物理地址。

## 5. 颠簸

颠簸本质上是指频繁的页调度行为，具体来讲，进程发生缺页中断，这时，必须置换某一页。然而，其他所有的页都在使用，它置换一个页，但又立刻再次需要这个页。因此，会不断产生缺页中断，导致整个系统的效率急剧下降，这种现象称为颠簸（抖动）。

内存颠簸的解决策略包括：

1) 如果是因为页面替换策略失误，可以修改替换算法来解决这个问题；
2) 如果是因为运行的程序太多，造成程序无法同时将所有频繁访问的页面调入内存，则要降低多道程序的数量；
3) 否则，还剩下两个办法：终止该进程或增加物理内存容量。

## 6.LeetCode 置换算法相关题目

### 6.1. LRU 题目

面试题 16.25. LRU 缓存（146. LRU 缓存机制）

https://leetcode-cn.com/problems/lru-cache-lcci/solution/

### 6.2. LFU 题目

# 7. IO 与磁盘调度

输入输出是在主存和外部设备（如磁盘驱动器、中断和网络）之间复制数据的过程。输入操作是从 IO 设备复制数据到主存，输出操作是从主存复制数据到 IO 设备。

## 7.1. Unix IO

一个 Linux 文件就是一个 m 个字节的序列。所有的 IO 设备都被模型化为文件，所有的输入输出操作都被当作对相应文件的读和写来执行。将设备映射为文件的方式，允许 Linux 内核引出一个简单的、低级的应用接口，即 Unix IO，使得输入和输出操作都能以一种统一的方式来执行：

1) **打开文件：**应用程序通过打开文件来告诉内核它想访问一个 IO 设备。内核返回一个非负整数，即描述符，用于在后续的操作中标识这个设备。内核会记录这个打开文件有关的信息，应用程序只需记住这个描述符。
2) **Linux Shell 创建的每个进程开始时都有三个打开的文件：**标准输入（描述符 0）、标准输出（描述符 1）和标准错误（描述符 2）。
3) **改变当前的位置：**对每个打开的文件，Linux 内核保持着一个文件位置 k，初始为 0。k 是从文件起始位置的字节偏移量。应用程序可以执行 seek 操作改变 k。
4) **读写文件：**读操作是从文件当前位置 k 开始，复制 n>0 个字节到内存，然后更新 k 为 k+n。如果一个文件大小 m 字节，当 k>=m 时，会触发一个 **EOF(end-of-file)**的条件。写操作是从内存复制 n>0 个字节到文件，从文件当前位置 k 开始，然后更新 k。
5) **关闭文件：**当应用完成了对文件的访问后，通知内核关闭该文件。内核释放文件打开时创建的数据结构，并将这个描述符恢复到可用的描述符池中。

### 7.2. 文件

每个 Linux 文件都有一个类型来标识它的角色：

1) **普通文件（binary file）**：包含任意数据，分为文本文件和二进制文件。文本文件只含有 ASCII 或 Unicode 字符；其他是二进制文件。
2) **目录（directory）**：包含一组链接的文件，其中每个链接都将一个文件名映射到一个文件，这个文件可能是另外一个目录。

3) **套接字（socket）**：用于与另一个进程通信。

## 7.2. I/O 系统的基本功能

I/O 系统管理的主要对象是 I/O 设备和相应的设备控制器。其最主要的任务是，完成用户提出的 I/O 请求，提高 I/O 速率，以及提高设备的利用率，并能为更高层的进程方便地使用这些设备提供手段。为了满足系统和用户的要求，I/O 系统应该具有以下几方面的基本功能

1) **隐藏物理设备的细节**：I/O 设备类型多，差异大。利用设备控制器（硬件）进行控制。隐藏物理设备的使用细节，仅向上层提高少量的，抽象的读写命令。
2) **与设备的无关性**：用户仅提供逻辑设备名来使用设备；例如打印时，它只需要提高读写命令和抽象的逻辑设备名即可，不必要指明是那一台打印机。
3) **提高处理机和 I/O 设备的利用率**：设备之间、设备与处理机之间均可并行操作。要求 CPU 快速响应 I/O 请求，减少对设备运行的干预时间。
4) **对 I/O 设备进行控制**：对 I/O 设备进行控制是驱动程序的功能。目前对 I/O 设备有四种控制方式：①采用轮询的可编程 I/O 方式；②采用中断的可编程 I/O 方式；③直接存储器访问方式；④I/O 通道方式。具体控制方式与设备的传输速率和传输数据单位有关。
5) **确保对设备的正确共享**：以共享属性来分类，分为独占设备、共享设备。
    a) 独占设备，进程应互斥地访问这类设备，即系统一旦把这类设备分配给了某进程后，便由该进程独占，直至用完释放。典型的独占设备有打印机、磁带机等。系统在对独占设备进行分配时，还应考虑到分配的安全性。
    b) 共享设备，是指在一段时间内允许多个进程同时访问的设备。典型的共享设备是磁盘，当有多个进程需对磁盘执行读、写操作时，可以交叉进行，不会影响到读、写的正确性。
6) **错误处理**：大多数的设备都包括了较多的机械和电气部分，运行时容易出现错误和故障。从处理的角度，可将错误分为临时性错误和持久性错误。对于临时性错误，可通过重试操作来纠正，只有在发生了持久性错误时，才需要向上层报告。

### 7.2.0. I/O 分类

按传输速率分为低速设备、中速设备和高速设备。

按信息交换的单位分类，分为：

1) 块设备：**由于信息的存取总是以数据块为单位**，所以存储信息的设备称为块设备。

它属于有结构设备，如磁盘等。磁盘设备的基本特征是传输速率较高，以及**可寻址**，即对它可随机地读/写任一块。

2) 字符设备：用于数据输入/输出的设备为字符设备，因为其传输的基本单位是字符。它属于无结构类型，如交互式终端机、打印机等。它们的基本特征是传输速率低、**不可寻址**，并且在输入/输出时常采用中断驱动方式。

### 7.2.1. 四种 IO 控制方式

1) **程序控制 IO**：CPU 代表一个进程给 IO 模块发送一个 IO 命令，该进程进入忙等待，直到操作完成才可以继续执行。

2) **中断驱动方式**：CPU 在向 IO 设备发出读命令后，可以转去做其它的事情，等到 IO 设备数据就绪，由 IO 设备主动发出中断请求打断 CPU。这样是 CPU 和设备都可以尽量忙起来。

3) **DMA 方式**：DMA 方式基本思想是，在主存和 IO 设备之间直接开辟数据通路，彻底解放 CPU。其特点是基本单位是数据块，所传送的数据是从设备直接送入内存的，或者相反。仅仅在一个或多个数据块传输开始或结束时才需要 CPU 的干预，这个数据块的传输是在 DMA 控制器的控制下完成的。

4) **通道控制方式**：IO 通道是指专门负责输入输出的处理机。它可以进一步减少 CPU 的干预。IO 通道与一般处理机的区别是通道指令类型单一，没有自己的内存，通道所执行的通道程序是放在主机的内存中的，也就是说通道与 CPU 内存共享。IO 通道与 DMA 方式的区别是，DMA 方式需要 CPU 来控制传输的数据块的大小、位置，而通道可自己控制。DMA 方式对应一台设备的数据传递，而通道可以控制多台设备与内存交换。

#### 7.2.1.1. 直接存储器访问 DMA

DMA 单元可以模拟处理器，能像 CPU 一样获得系统总线的控制权。DMA 技术工作流程如下，当 CPU 想读或者写一块数据时，它通过向 DMA 模块发送一下信息来给 DMA 模块发送一条命令：

1) 请求读操作或写操作的信号，通过在 CPU 和 DMA 模块之间使用读写控制线发送；

2) 相关的 IO 设备地址，通过数据线发送；

3) 从存储器中读或往存储器写的起始地址，在数据线上传送，并由 DMA 模块保存在其地址寄存器中；

4) 读或写的字数，也通过数据线传送，并由 DMA 模块保存在其数据寄存器中。

然后 CPU 继续执行其他工作，此时它已经把这个 IO 操作委托给 DMA 模块。DMA 模块直接从存储器或往存储器中传送整块数据，一次传送一个字，并且不需要再通过 CPU。传送结束后，DMA 模块给 CPU 发送一个中断信号。因此，只有在传送开始和结束时才会用到 CPU。

### 7.2.2. IO 子系统的层次结构

层次结构由：用户 IO 层软件、设备独立性软件、设备驱动程序、中断处理程序组成。

**用户 IO 层软件**的实现与用户交互的接口，用户可直接调用在用户层提供的、IO 操作有关的函数库。

**设备独立性软件**，用于实现用户程序与设备驱动器接口的统一，使得应用程序独立于具体的物理设备设备。主要功能又：执行所有设备的共有操作、想用户层提供接口。

**设备驱动程序**，直接与硬件相关，负责具体实现对设备发出的操作命令。通常每一类设备配备一个设备驱动程序，它是 IO 进程与设备 控制器之间的通信程序。

**中断处理程序**，用于保存被中断进程的 CPU 环境，转入响应的中断处理程序进行处理，处理完并恢复中断进程的现场。

## 7.3. IO 缓冲

假设用户要磁盘中读取多个数据块到用户进程地址空间中的一个区域，可以对磁盘单元执行一个 IO 命令，并等待数据传送完毕。这个等待可以是忙等待，也可以是进程被中断挂起。这种方法存在问题。首先，进程被挂起，等待相对更慢的 IO 完成。其次，这种挂起干扰了系统的交换策略，甚至可能导致单进程死锁。

缓冲：在输入请求发出前就开始执行输入传送，并且在输出请求发出一段时间后才开始执行输出传送。

### 7.3.1. 单缓冲

当用户进程发出 IO 请求时，操作系统给该操作分配一个位于内存中系统部分的缓冲区。输入传送的数据放入到系统缓冲区中，当传送完成时，进程把该块移到用户空间，并立即请求另一块。（预读）用户可以在下一块数据读取的同时，处理已读入的数据块。

### 7.3.2. 双缓冲

作为对单缓冲方案的改进，可以给操作分配两个缓冲区。在一个进程往一个缓冲区传送数据的同时，操作系统正在清空另一个缓冲区。

### 7.3.3. 循环缓冲

使用两个以上的缓冲区。

## 7.4. 磁盘调度

### 7.4.1. 磁盘性能参数

**寻道时间**：磁头定位到磁道所需要的时间；

**旋转延迟**：将磁盘的待访问地址区域旋转到读/写磁头可访问的位置所需要的时间；

**存取时间**：寻道时间+旋转延迟。

**传输时间**：一旦磁头定位完成，磁头就通过下面旋转的扇区开始执行读操作或者写操作，此即数据传送部分。该部分的时间即为传输时间。

除了存取时间和传输时间外，一次磁盘 IO 操作通常还会有一些排队延迟。当进程发出一个 IO 请求时，它必须在一个队列中等待该设备可用。

影响：从磁盘读扇区的顺序，对 IO 的性能有很大影响！

### 7.4.2. 磁盘调度策略

不同的扇区读取顺序造成的 IO 性能差异可以追溯到寻道时间。为提高 IO 性能，需要减少花费在寻道上的时间。

#### 7.4.2.1. 先进先出 FIFO

按顺序处理队列中的项目。该策略具有公平的优点，每个请求都会按接到的顺序得到处理。性能上接近于随机调度。

#### 7.4.2.2. 优先级

基于优先级的调度的控制并不会优化磁盘的利用率，不过可以满足系统的其他目标。比如较短作业和交互作业的优先级更高，而较长作业的优先级较低，使得大量的短作业可以迅速完成 IO 操作。但可能导致长时间作业饥饿。

#### 7.4.2.3. 最短服务时间优先

选择使磁头臂从当前位置开始移动最少的磁盘 IO 请求。最短服务时间优先策略总是选择导致最小寻道时间的请求。

#### 7.4.2.4. SCAN

电梯算法：要求磁头臂仅沿一个方向移动，并在途中满足所有未完成的请求，直到它到达这个方向上的最后一个磁道，或者在这个方向上没有别的请求为止，接着反转服务方向，

沿相反反向扫描。

### 7.5. 磁盘高速缓存

#### 7.5.1. 概述

磁盘高速缓存是内存中为磁盘扇区设置的一个缓冲区，它包含有磁盘中某些扇区的副本。当出现一个请求某一特定扇区的 IO 请求时，首先检测该扇区是否在磁盘高速缓存中。如果在，则该请求通过这个高速缓存来满足；否则，把被请求的扇区读到磁盘高速缓存中。由于访问的局部性现象的存在，当一块数据被取入高速缓存以满足一个 IO 请求时，很有可能将来还会访问到这一块数据。

#### 7.5.2. 置换策略

最常用的是最近最少使用算法（LRU）：置换在高速缓冲中未被访问的时间最长的块。

另一个方法使最不经常使用页面置换算法(LFU)：置换集合中被访问次数最少的块。

## 详细参考：

https://blog.csdn.net/qq_44853882/article/details/108284979?utm_medium=distribute.pc_relevant.none-task-blog-baidujs_baidulandingword-2&spm=1001.2101.3001.4242

## 8. 什么是用户态和内核态？

为了限制不同程序的访问能力，防止一些程序访问其它程序的内存数据，CPU 划分了用户态和内核态两个权限等级。

1) **用户态只能受限地访问内存，且不允许访问外围设备，没有占用 CPU 的能力，CPU 资源可以被其它程序获取**；
2) **内核态可以访问内存所有数据以及外围设备，也可以进行程序的切换**。

所有用户程序都运行在用户态，但有时需要进行一些内核态的操作，比如从硬盘或者键盘读数据，这时就需要进行系统调用，使用陷阱指令，CPU 切换到内核态，执行相应的服务，再切换为用户态并返回系统调用的结果。

## 为什么要分用户态和内核态？

1) 安全性：防止用户程序恶意或者不小心破坏系统/内存/硬件资源；
2) 封装性：用户程序不需要实现更加底层的代码；

3) 利于调度：如果多个用户程序都在等待键盘输入，这时就需要进行调度；统一交给操作系统调度更加方便。

## 如何从用户态切换到内核态？

1) **系统调用：**比如读取命令行输入。本质上还是通过中断实现
2) **用户程序发生异常时：**比如缺页异常
3) **外围设备的中断：**外围设备完成用户请求的操作之后，会向 CPU 发出中断信号，这时 CPU 会转去处理对应的中断处理程序