操作系统

[1. 基础概念 1](#_101)

[1.1. 共享 1](#_104)

[1.1.1. 互斥共享（临界资源） 1](#_113)

[1.1.2. 同时共享 1](#_114)

[1.2. 虚拟 1](#_115)

[1.2.1. 时分复用：多进程并发 1](#_116)

[1.2.2. 空间复用：虚拟内存 1](#_117)

[1.3. 同步/异步 1](#_118)

[1.3.1. 互斥量 1](#_232)

[1.3.2. 信号量P/V操作 1](#_233)

[2. 进程管理 1](#_102)

[2.1. 进程 1](#_105)

[2.1.1. 进程控制块PCB来描述其基本信息和运行状态 1](#_129)

[2.1.2. 资源分配基本单位：有独立的地址空间 1](#_107)

[2.1.3. 至少有1个线程 1](#_108)

[2.1.4. fork 1](#_112)

[2.1.5. 进程间通信IPC 2](#_131)

[2.1.6. 状态 2](#_138)

[2.2. 线程 2](#_106)

[2.2.1. CPU调度基本单位 2](#_130)

[2.2.2. 线程模型 2](#_145)

[2.3. 调度 3](#_150)

[2.3.1. 抢占和非抢占 3](#_151)

[2.3.2. 根据使用场景选择 3](#_152)

[2.4. 死锁 3](#_161)

[2.4.1. 必要条件 3](#_162)

[2.4.2. 处理方法 4](#_163)

[3. 文件管理 4](#_187)

[3.1. 磁盘 4](#_188)

[3.1.1. 盘面 4](#_189)

[3.1.2. 磁道 4](#_190)

[3.1.3. 扇区 4](#_191)

[3.1.4. 新的磁盘外磁道的数据更多 4](#_192)

[3.1.5. 读取磁盘内容时间 4](#_193)

[3.1.6. 调度 5](#_202)

[3.2. 文件 5](#_197)

[3.2.1. inode 5](#_198)

[4. 问题 5](#_228)

[4.1. 孤儿进程和僵尸进程 5](#_229)

[4.2. 守护进程 6](#_230)

[4.3. 静态链接和动态链接 6](#_231)

[5. 内存管理 6](#_103)

[5.1. 动态重定位 6](#_168)

[5.1.1. 多进程总内存大于物理内存 6](#_169)

[5.2. 空闲内存管理 8](#_172)

[5.2.1. 位图 8](#_173)

[5.2.2. 空闲链表 8](#_174)

[6. 大内核和微内核 8](#_119)

[6.1. 大内核 8](#_120)

[6.2. 微内核 9](#_121)

[7. 中断 9](#_122)

[7.1. 外中断 9](#_123)

[7.1.1. I/O中断 9](#_125)

[7.1.2. 时钟中断 9](#_126)

[7.2. 内中断 9](#_124)

[7.2.1. 异常：非法操作码、地址越界等 9](#_127)

[7.2.2. 陷入：系统调用 9](#_128)

[8. Linux 9](#_206)

[8.1. awk 9](#_207)

[8.2. uniq 9](#_208)

[8.3. sort 9](#_209)

[8.4. netstat 9](#_219)

[8.5. ss 9](#_234)

[8.6. pstree 10](#_235)

[9. I/O 10](#_210)

[9.1. 阻塞I/O 10](#_211)

[9.2. 非阻塞I/O 10](#_212)

[9.3. 多路复用 10](#_213)

[9.3.1. select 10](#_216)

[9.3.2. poll 11](#_218)

[9.3.3. epoll 11](#_217)

[9.4. 信号驱动 12](#_214)

[9.5. 异步I/O 12](#_215)

[10. 进程空间 12](#_223)

[10.1. 用户态 12](#_224)

[10.2. 内核态 12](#_225)

# 基础概念

## 共享

### 互斥共享（临界资源）

### 同时共享

## 虚拟

### 时分复用：多进程并发

### 空间复用：虚拟内存

## 同步/异步

### 互斥量

### 信号量P/V操作

# 进程管理

## 进程

### 进程控制块PCB来描述其基本信息和运行状态

### 资源分配基本单位：有独立的地址空间

#### 进程管理：寄存器，程序计数器，优先级等

#### 存储管理：正文段，数据段和堆栈段

#### 文件管理：文件描述符等

### 至少有1个线程

### fork

父子进程仅共享正文段，子进程拥有父进程地址空间的副本

### 进程间通信IPC

#### 管道：匿名和有名

#### 消息队列

#### 共享内存

#### 信号量

#### 套接字socket

#### 信号

### 状态

#### 运行

#### 就绪

#### 阻塞

## 线程

### CPU调度基本单位

### 线程模型

#### 内核线程

##### 调用接口-轻量级进程LWP

###### 最小执行上下文

###### 调度的统计信息

##### Hotspot采用1:1模型

#### 用户线程

#### 混合模型

## 调度

### 抢占和非抢占

### 根据使用场景选择

#### 批处理

##### 最短作业优先

##### 最短剩余时间优先

#### 交互

##### 时间片轮转

需要选择合理的时间片，一般设为20~50ms。

时间片太短，会导致过多的进程切换，降低CPU效率；时间片太长，会使得短交互请求的响应时间变长。

并且时间片轮转不考虑进程等待时间和执行时间，平均等待时间较长，上下文切换开销大。

##### 优先级调度

为了防止高优先级进程无休止运行，主要有两种做法：

在每个始终滴答降低当前进程的优先级

给每个进程赋予一个允许运行的最大时间片。

##### 多级队列

最高优先级进程运行一个时间片，次高优先级进程运行两个时间片。当进程用完分配的时间片后，移到下一类。

#### 实时

软/硬实时

## 死锁

在两个或多个并发进程中，如果每个进程持有某种资源而又都等待别的进程释放它或它们现在保持着的资源，在未改变这种状态之前都不能向前推进，称这一组进程产生了死锁。

### 必要条件

**互斥**：每个资源只能分配给一个进程。

**占有和等待**：每个进程在申请新资源的时候能够保持已经申请到的资源。

**不可抢占**：已经分配给一个进程的资源不能被强制性地抢占。

**循环等待**：若干进程之间形成一种头尾相接的循环等待资源关系。

### 处理方法

#### 鸵鸟策略

就什么都不干，主要是根据死锁发生产生的代价和频率。

#### 死锁预防

主要是破坏各个必要条件

**破坏占有和等待**：一次性获取所有资源

**破坏不可抢占**：请求资源失败时，需释放已有资源

**循坏等待**：顺序资源分配法

#### 死锁避免

银行家算法：当一个进程申请使用资源的时候，通过先试探分配给该进程资源，然后通过安全性算法判断分配后的系统是否处于安全状态，若不安全则试探分配作废，让该进程继续等待。

#### 死锁检测与恢复

**检测**：利用环路进行检测

**解除**：利用抢占、回滚、杀死进程恢复

# 文件管理

## 磁盘

### 盘面

### 磁道

### 扇区

### 新的磁盘外磁道的数据更多

### 读取磁盘内容时间

#### 寻道时间

#### 旋转延迟

#### 传输时间

### 调度

#### 先来先服务

#### 最短寻道时间优先（距离当前磁头位置最近）

#### 电梯调度

## 文件

### inode

#### 硬链接inode一样

#### 软链接inode不一样

#### 包含文件数据所在地址

# 问题

## 孤儿进程和僵尸进程

**孤儿进程**

当一个父进程退出，而它的一个或多个子进程还在运行，那么这些子进程将会成为孤儿进程。孤儿进程被init进程（进程id为1的进程）收养，并由init进程对它们完成状态手机工作。因为孤儿进程会被init进程收养，所以不会对系统造成危害。

**僵尸进程**

僵尸进程就是一个子进程的进程描述符在子进程退出时不会释放，只有当父进程通过wait()或waitpid()获取子进程信息后才会释放。如果子进程退出，而父进程没有调用wait()或waitpid()，那么子进程的进程描述符仍然保存在系统中，这种进程称为僵尸进程（通过ps -ef显示出来的状态为Z）

系统能使用的进程号是有限的，如果产生大量僵尸进程，可能会因为无可用的进程号而导致系统不能产生新的进程。如果要消灭系统中大量的僵尸进程，只需要将其父进程杀死，此时僵尸进程就会变成孤儿进程，从而被init进程收养，并释放所有僵尸进程所占有的资源，从而结束僵尸进程。

## 守护进程

运行在后台的一种特殊进程，独立于控制终端的，并周期性地执行某些任务。

## 静态链接和动态链接

**静态链接**

在编译期间，由编译器和连接器将静态库集成到应用程序内，并制成目标文件以及可以独立运作的可执行文件。静态库一般是一些外部函数和变量的集合。

**动态链接**

可以在首次载入的时候执行，也可以在程序开始执行时完成，比如标准C库（libc.so）就是动态链接，这样所有的程序就可以共享一个库，而不用分别封装（静态链接）。

# 内存管理

## 动态重定位

通过基址寄存器和界限寄存器

### 多进程总内存大于物理内存

#### 交换

#### 虚拟内存

##### MMU

程序生成的地址都是虚拟地址，并形成虚拟地址空间。

不会直接将地址发送到内存总线上，而是交给MMU（Memory Management Unit）内存管理单元把虚拟地址映射为物理内存地址。

###### 页表

主要是页到页框的映射，等同于：页框=f(页)

每一条页表项包括：

高速缓存禁止位：可以禁止页面被高速缓存。

修改位：跟踪页面使用情况，如果页面被写入时就会自动设置修改位。在换页时，如果一个页面被修改过，就会写回磁盘。

访问位：帮助操作系统在发生缺页中断时选择要淘汰的页。

保护位：表示读写权限

在/不在：表明页表项是否有效

###### 快表（硬件）

主要是为了加快虚拟地址到物理地址的映射。

切换进程时，会把进程页表的一部分加载进来。

在寻页时，先查询快表，命中则直接返回，未命中则查页表（如果页表在磁盘中，则读入），并且置换TLB中的表项。

###### 多级页表

主要为了解决页表过大的问题。

避免将全部页表都保存在内存中，不需要的页表就不应该保留。

###### 页表工作流程

1. 进程创建或切换，刷新MMU和TLB，并装载部分页表

2. 进程执行，访问的虚拟地址并未加载到内存，产生缺页

操作系统选择空闲页框，根据文件描述符加载到指定物理内存（发生在正文段或数据段）；从交换区中加载到指定物理内存（发生在堆栈段）从物理内存中加载或从磁盘中加载。

如果TLB页表空间足够，不考虑页表项置换，如果不够，则置换到内存。

如果物理内存不够，则对部分页框置换到交换区。

3. 恢复中断发生时的状态，继续执行

##### 段页式

地址空间划分为多个段，每个段都构成了自己独立的地址空间，并且划分成大小相同的页。

每个段的长度可以不同，并且可以在运行期间动态改变（如堆栈段）而不会影响到其他的段。

段可划分为代码段、数据段、堆栈段等。

分段的好处在于：使程序和数据可以被划分为逻辑上独立的地址空间并有助于共享和保护。

##### 置换算法

###### LRU（最近最久）

###### LFU（最不经常访问算法）

###### 先进先出

###### 第二次机会算法

###### 时钟算法

第二次机会算法的优化，使用了环形链表

##### 缺页中断

###### soft miss

访问的内存不在虚拟地址空间，但是在物理内存中，只需要MMU建立物理内存和虚拟地址空间的映射。

通常是多个进程访问同一个共享内存中的数据，可能某些进程还没有建立起映射关系。

###### hard miss

访问的内存不在虚拟地址空间中，也不在物理内存中，需要从磁盘中载入。

## 空闲内存管理

### 位图

0空闲，1占用

### 空闲链表

空闲区H和进程区P

# 大内核和微内核

## 大内核

将操作系统作为一个紧密结合的整体放到内核

## 微内核

仅有微内核这一模块运行在内核态，其余模块运行在用户态

# 中断

## 外中断

### I/O中断

### 时钟中断

## 内中断

### 异常：非法操作码、地址越界等

### 陷入：系统调用

# Linux

## awk

普通用法：last -n 5 | awk '{print $1}'

复杂用法：awk -F ':' 'BEGIN {count=0;} {name[count] = $1;count++;}; END{for (i = 0; i < NR; i++) print i, name[i]}' /etc/passwd

## uniq

## sort

## netstat

netstat用于显示各种网络相关信息。

1. netstat -anp | grep port\_id

根据port\_id查看端口占用情况

## ss

Socket Statistics。ss命令可以用来获取socket统计信息，它可以显示和netstat类似的内容。但ss的优势在于它能够显示更多更详细的有关TCP和连接状态的信息，而且比netstat更快速更高效。

ss -pl # 查看进程使用的socket（监听）

## pstree

pstree [-p] <pid>显示进程树

# I/O

## 阻塞I/O

## 非阻塞I/O

## 多路复用

### select

1. 使用copy\_from\_user将用户空间中的fd\_set拷贝到内核空间。

2. 注册回调函数\_\_pollwait

3. 遍历所有的fd，调用其对应的poll方法。以tcp\_poll为例，其核心实现就是\_\_pollwait。

4. pollwait的主要工作是把当前进程挂到设备的等待队列中（不同的设备有不同的等待队列）。在设备收到一条消息或填写完文件数据后，会唤醒设备等待队列上的进程。

5. poll方法会返回一个描述读写操作是否就绪的mask掩码，根据这个掩码给fd\_set赋值。

6. 如果遍历完所有的fd，还没有一个可读写的掩码，则会调用schedule\_timeou使调用select的进程进入睡眠。当设备驱动发生自身资源可读可写后，会唤醒等待队列上的睡眠的进程。如果超过一定的时间还没人唤醒，则调用select的进程会重新被唤醒获得CPU，进而重新遍历fd，判断有没有就绪的fd。

7. 把活跃的fd\_set从内核空间拷贝到用户空间。

缺点：

1. 每次调用select都需要拷贝fd\_set至内核空间，返回时select也会把保留了活跃事件的fd\_set返回。当fd\_set数量大的时候，这个过程消耗是很大的。

2. select需要逐个遍历fd\_set集合，然后去检查对应fd的可读写状态。如果fd\_set数据量较多，那么遍历fd\_set就是一个比较耗时的过程。

3. fd\_set是个集合类型的数据结构，有长度限制。32位系统支持1024个，64位系统支持2048个。

另外，每次调用select前都要重新设置文件描述符和时间，因为事件发生后，文件描述符和事件都被内核修改了。

### poll

早期计算机网络不发达，所以并发网络请求并不会很高，select模型也足够使用了，但是随着网络的高速发展，高并发的网络请求程序越来越多，而select模式下的fd\_set长度限制就成为了缺陷，随之而来的poll模型通过链表形式来保存监控的fd信息。

poll采用pollfd保存对应fd需要监控的event集合，也保存了一个当返回激活事件的event集合。

struct pollfd {

int fd;

short events;

short revents;

};

还存在的问题应该就是遍历的问题。

### epoll

epoll采用的是一组方法调用的方式，工作流程大致如下：

1. 创建内核事件表：int epoll\_create(int size)

向内核申请创建一个fd的文件描述符作为内核事件表（红黑树）

2. 添加或移除监控的fd和事件类型：int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event)

epfd：epoll\_create的返回值

op：表示动作，包括注册新的fd、修改已经注册的fd的监听事件和从epfd中删除一个fd。

fd：需要监听的fd

events：内核需要监听什么事

向事件监听表中添加或删除fd事件。

3. 等待事件的产生：int epoll\_wait(int epfd, struct epoll\_event\* events, int maxevents, int timeout)

内核向事件表的fd绑定一个回调函数，当监控的fd活跃时，会调用callback函数把事件加入到一个活跃事件队列里。最后在epoll\_wait返回的时候内核会把活跃事件队列里的fd和事件类型返回给应用程序。

4.遍历events（里面有可能有没有发生事件的fd）

相对于select和poll都需要在内核中遍历监控的fd集合，并且在返回时需要遍历事件列表（select是遍历所有fd，并比较fd\_set；poll是遍历返回的pollfd链表），epoll返回的是活跃的事件列表，直接处理就行。

#### ET

边缘触发，效率非常高，在并发，大流量情况下，会比LT少很多epoll的系统调用，因此效率高。但是对编程要求高，需要细致处理每个请求，否则容易发生丢失事件的情况。

#### LT

水平触发，效率低于ET触发，尤其在大并发，大流量情况下。但是LT对代码编写要求比较低，不容易出现问题。

LT模式服务器编写上的表现是：只要有数据没被获取，内核就不断通知你，因此不用担心事件丢失的情况。

## 信号驱动

## 异步I/O

# 进程空间

## 用户态

只能受限地访问内存，且不允许访问外围设备，占用CPU的能力被剥夺，CPU资源可以被其他程序获取。

## 内核态

CPU可以访问内存的所有数据，包括外围设备，例如硬盘，网卡，CPU也可以将自己从一个程序切换到另一个程序。