概述

系统调用

用户态

核心态 访问资源不受限 占用cpu不抢占

用户态 trap->核心态

中断

如何使cpu感知中断 硬件中断 （内外中断）+软件中断 INT 指令

An interrupt is an asynchronous event that is typically triggered by an I/O device.

中断也就是外中断 io中断

An exception is a synchronous event that is generated when the processor detects one or more predefined conditions while executing an instruction.

异常也就是内中断 trap 系统调用 fault 缺页中断 abort 除0

关中断

保护现场

请求中断

恢复线现场

开中断

进程管理

进程和线程的区别

1.资源

进程是资源分配的基本单位，线程访问进程的资源

2.调度

线程是独立调度的基本单位，进程间线程切换不会引起进程上下文切换

3.开销

一大一小

五种状态

Create-------->Ready<---------->Runnable------>terminated

<----waiting<---

进程通信方式

socket 多机

共享内存 速度快 进程安全问题

mq 容量以及大小受限

管道 速度慢 容量有限 匿名管道 父子进程和命名管道 任意进程

信号

semaphore 只能同步

线程同步机制

临界区

信号量

互斥量

事件 callback

用户级线程和内核级线程

多对一和一对一

进程调度策略

先来先服务 短作业优先 时间片轮转 优先级调度

临界资源和临界区

一次只有一个进程使用的资源， 访问临界资源的代码是临界区

死锁

请求保持 互斥访问 循环等待 不抢占

内存管理

各种地址

物理地址 内存的绝对地址

偏移地址(有效地址) offset

逻辑地址 段地址和偏移量

线性地址 保护模式的段地址和偏移量 段地址先去GDT（全局描述表）找真正的段地址

虚拟地址 如果线性地址开启了分页 那么还不能直接把找到的段地址+偏移量作为物理地址，所以叫为虚拟地址

连续分配

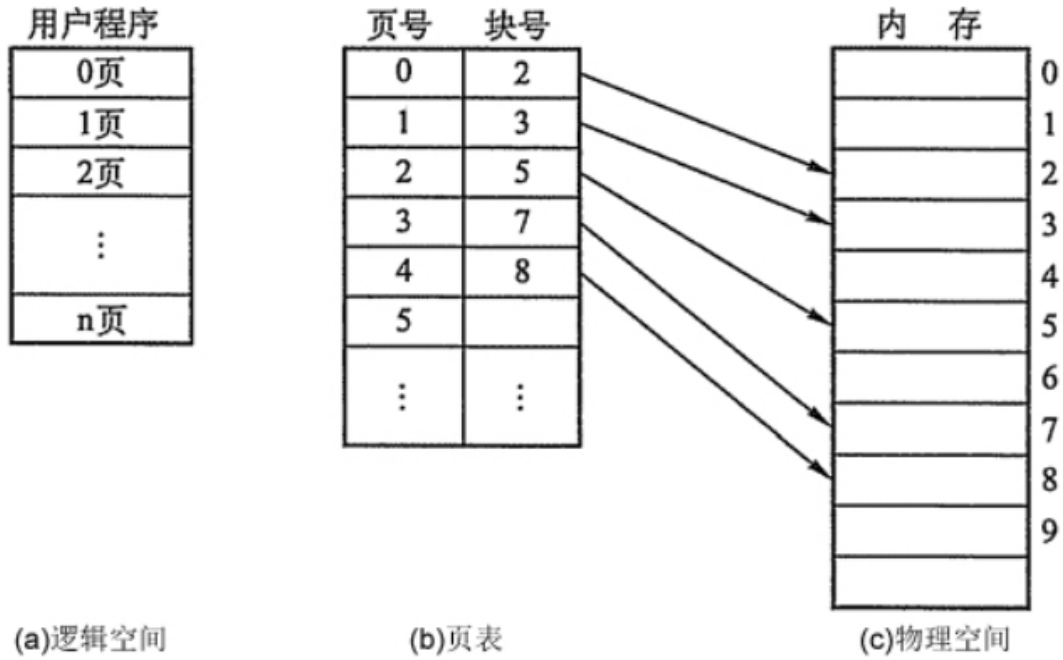
动态分区算法

最佳适应 最坏适应 邻近适应

非连续分配

分页

提高内存空间利用率



每个进程一个页表，页表项记录页号和块号，知道物理内存和页面大小即可知道页表项数量，知道页表在内存的起始位置就可以知道页表项的内存位置,页号也就隐藏了

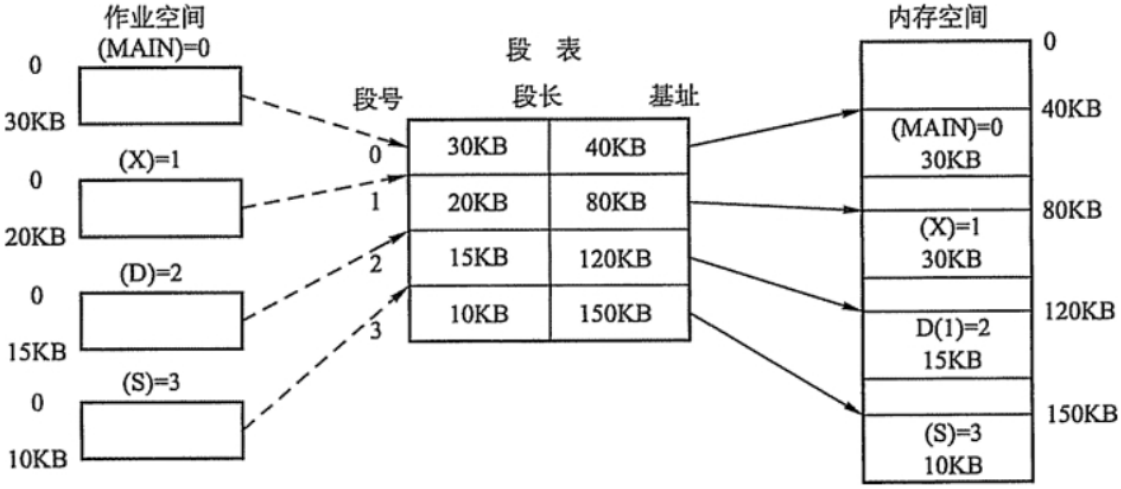
通过逻辑地址x可以算出页号 x/页面大小，偏移地址 x%页面大小

两级分页

因为页表需要连续存放，当页表项数量较多时，仍要占用较大连续的内存，可以将页表项再放到离散的地方，通过再建立一张页表进行映射查找

分段

满足逻辑需求



和页表不一样的是段长不固定，所以是二维的基址和段长(偏移量)

请求分页

换进和换出的功能

交换空间

物理内存和硬盘进行交换以释放空间，硬盘上的空间就叫交换空间，物理内存+交换空间就是虚拟内存的容量

页面置换算法

最佳置换(需要知道后序访问页面不可实现)

FIFO

LRU

NRO Not Recently Used 每个页面两个状态R(读) M(写) R定时置0并按照其大小00 01 10 11 越小越被换出

设备管理

磁盘调度算法

FIFO

最短寻道 贪心

电梯 先左后右或者先右后左

IO管理

IO模型

同步，异步：**访问数据的方式，同步需要主动读写数据，在读写数据的过程中还是会阻塞；异步只需要I/O操作完成的通知，并不主动读写数据，由操作系统内核完成数据的读写。**

概念指用户态和内核的交互方式，传输层以上为用户，反之为内核

同步指用户发起io请求后需要等待或者轮询

异步指用户发起io请求直接继续执行，等待kernel io完成后通知用户，或者调用用户注册的回调函数

阻塞，非阻塞：**进程/线程要访问的数据是否就绪，进程/线程是否需要等待；**

阻塞指用户发起io请求需要彻底完成才能返回用户空间

非阻塞指用户发起io请求后直接会得到状态值

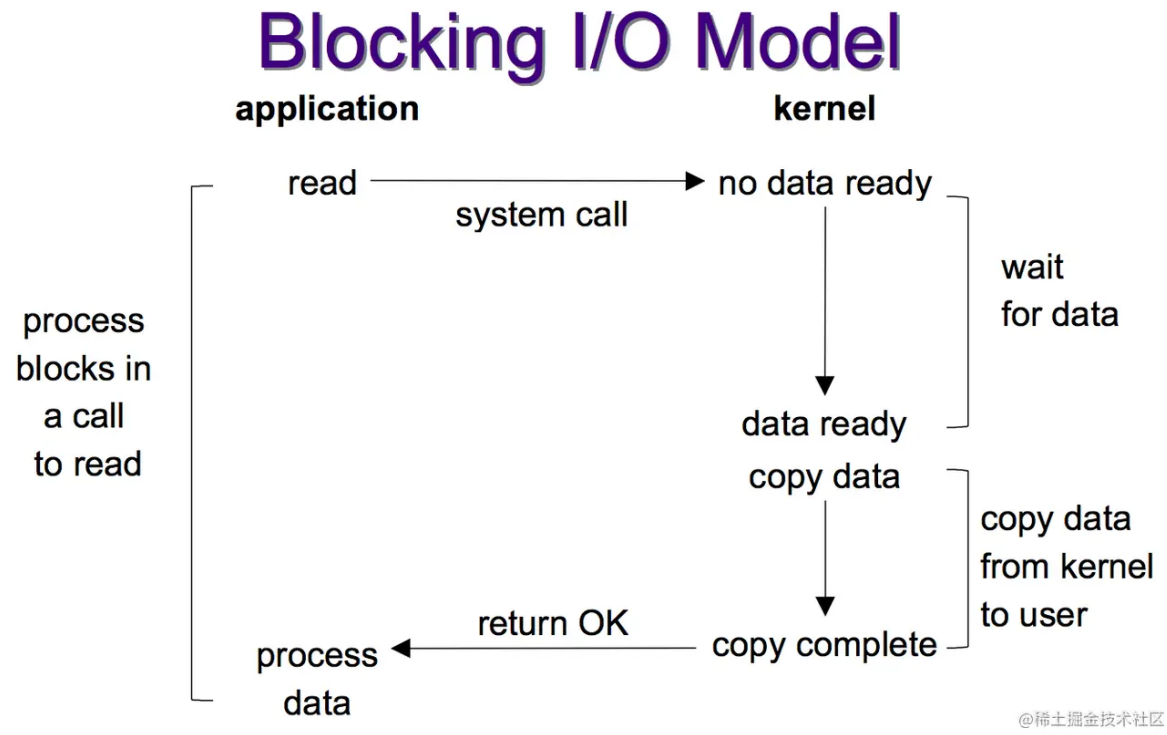
阻塞跟非阻塞，是在网卡到内核缓冲区；同步跟非同步，指的是需不需要等待CPU进行数据的拷贝。

BIO

Blocking IO （同步阻塞）

用户需要等待kernel等待数据和拷贝到用户空间

此时用户线程从runnable变为block，cpu利用率不够



如何去改进呢

```
while(1) {
  connfd = accept(listenfd);  // 阻塞建立连接
  pthread_create（doWork);  // 创建一个新的线程
}
```

小trick 服务器新建线程去接受阻塞这样服务器主线程不会阻塞仍然可以监听客户端

用OS实现即为NIO **为我们提供一个非阻塞的 read 函数。**

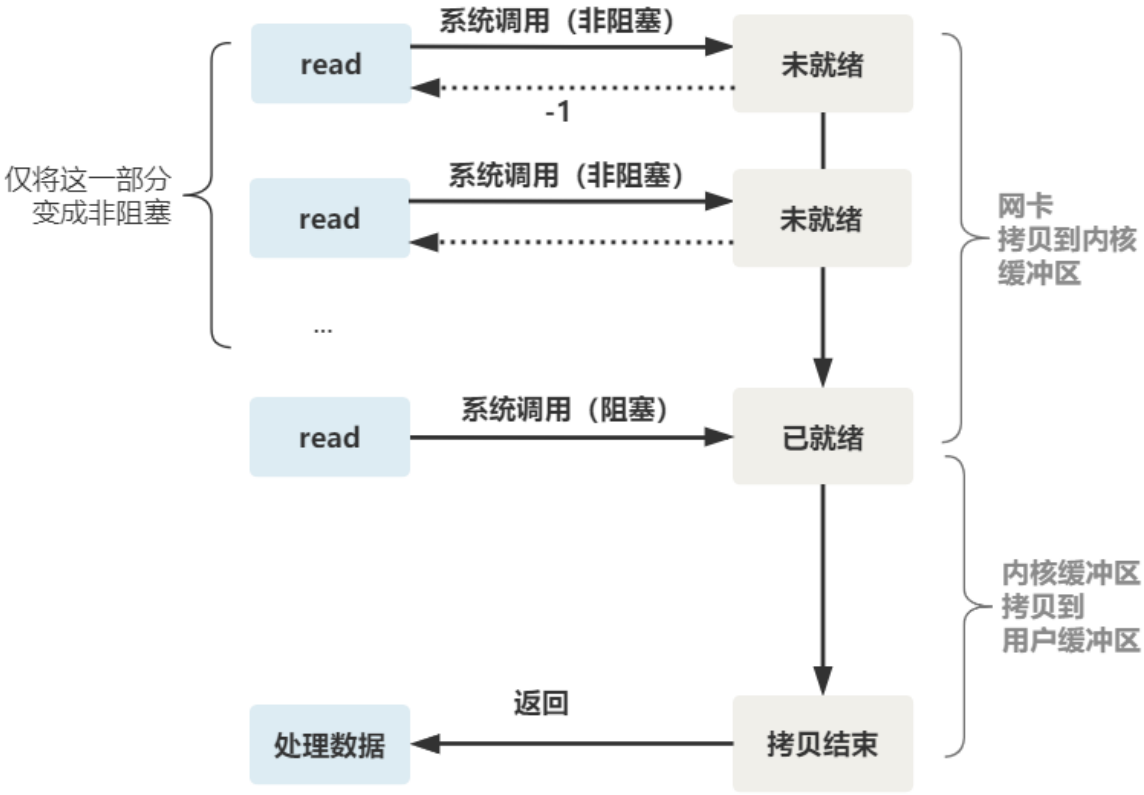
```
int n = read(connfd, buffer) != SUCCESS); //文件描述符
```

NIO

Non-blocking IO（同步非阻塞）

用户可以在发起io请求后返回，但是并未读到真正数据，用户线程需要轮询，耗费大量cpu

当网卡拷贝到内核缓冲区前都属于非阻塞，需要轮询read，而拷贝后进行系统调用仍然是阻塞



小trick 我们把文件描述符放到数组中开一个线程，轮询read

是不是有点多路复用的感觉呢

用OS实现即为select 但是它的数组有上限1024而poll没有

而epoll优化更猛

1. select 调用需要传入 fd 数组，需要拷贝一份到内核，高并发场景下这样的拷贝消耗的资源是惊人的。（可优化为不复制）
2. select 在内核层仍然是通过遍历的方式检查文件描述符的就绪状态，是个同步过程，只不过无系统调用切换上下文的开销。（内核层可优化为异步事件通知）
3. select 仅仅返回可读文件描述符的个数，具体哪个可读还是要用户自己遍历。（可优化为只返回给用户就绪的文件描述符，无需用户做无效的遍历）

所以 epoll 主要就是针对这三点进行了改进。

1. 内核中保存一份文件描述符集合，无需用户每次都重新传入，只需告诉内核修改的部分即可。
2. 内核不再通过轮询的方式找到就绪的文件描述符，而是通过异步 IO 事件唤醒。
3. 内核仅会将有 IO 事件的文件描述符返回给用户，用户也无需遍历整个文件描述符集合。

IO多路复用

同步阻塞