操作系统与系统编程面试高频问题

Linux内存模型

- 从高地址到低地址

1. 环境变量和命令行参数
2. 栈区
3. 共享区
4. 堆区
5. 未初始化数据段.bss
6. 初始化数据段.data
7. 代码段.text

代码生成可执行文件的过程

- 主要分为四个步骤

1. 预编译阶段：对g++编译器指定-E参数，生成.i文件。这个阶段的主要工作是将所有的宏展开，去掉所有的条件预编译指令，将所有的头文件包含进来，删除注释等。
2. 编译阶段：对g++编译器指定-S参数，生成.s汇编文件。这个阶段的主要工作是对代码的语法，语义和词法等进行分析。
3. 汇编阶段: 对g++编译器指定-c参数，生成.o二进制文件。
4. 链接阶段：将各个模块之间的相互引用处理好。把所有的静态库用到的目标文件装入程序中，并进行统一编址，然后进行重定位，即逻辑地址到物理地址的转换。

静态库与动态库

1. 静态库：命名方式为lib开头加上自定义的静态库名，然后以.a结尾。静态库实际上是一组目标文件的集合，再链接阶段与调用的程序生成可执行文件。 优点：代码加载速度快，发布程序时，不需要提供对应的库； 缺点：可执行文件体积大，同时如果静态库有修改，调用的程序需要重新编译，而编译的耗时比较久。

2. 动态库：命名方式为lib开头加上自定义的动态库名，然后以.so结尾。动态库首先生成与位置无关的目标文件，然后在运行时加载到内存。 优点：动态库可以共享，节省了系统资源，动态库进行修改后，无需重新编译。 缺点：加载速度比静态链接慢，发布程序时，需要单独提供动态库。

内存对齐

- union最大成员所占的整数倍，同时能容纳其他的成员。union中变量共用内存，应以最长的为准。
- struct按照成员的声明顺序，依次安排内存，偏移量为成员大小的整数倍，最后结构体的大小为最大成员所占大小的整数倍。在C++中，空结构体和空类的内存所占大小为1个字节。C中空结构体所占大小为0。
- 为什么要有内存对齐：
    1. 硬件原因：加速CPU的访问速度。因为CPU和内存数据交换的基本单位是块，块的大小为2的n次方字节。内存未对齐可能需要多次访问内存。2. 平台原因：不是所有的平台都支持任意地址的数据访问。

```cpp
#include <iostream>
using namespace std;
typedef union{
    long long i; //8 bytes
    int k[5]; //4 bytes 最长的成员不是20
    char c; // 1 byte
}UDATE;
//联合体共用内存 最长成员为8字节 结果要为8的倍数 同时要能容纳其他成员，即大于等于20字节所以为24字节
struct data{
    int cat; // 4 bytes
    UDATE cow; //24 bytes 但是需要先拆开来 最长成员为8字节
    double dog; //8 bytes
}too;
//结构体顺序考虑，结果为最大成员的整数倍，如果后一个成员的长度的开始位置不是整数倍需要填充字节
//cat占4个字节 填充4个字节
//起始位置为8 满足整数倍 cow占用24字节
//起始位置为32 满足整数倍 doule占用4字节
//所以结构体总共占用40字节，同时40也是8的倍数。
UDATE temp;
int main(){
    cout<<sizeof(temp)<<" "<< sizeof(struct data)<<endl; //24 40
    return 0;
}
```

new和malloc的区别

- malloc只分配内存不初始化；new不仅分配内存也初始化，new分配内存以后自动调用构造函数。
- malloc分配内存时必须指定内存大小，而new可以自动计算。malloc分配完成后返回的是void*类型，需要强转，而new返回的是对应类型的指针。
- malloc分配内存失败时返回NULL，而new分配内存失败时抛出bad_alloc异常。

volatile

- 用来告诉编译器不要对该变量做任何优化，编译器每次操作该变量时，一定要从内存中取出，而不是使用寄存器中与存在的值，因为值可能已经发生了改变。 应用场景：

1. 并行设备的硬件寄存器（如状态寄存器）。
2. 多线程中共享变量。

虚拟内存的作用

1. 主存容量有限
2. 分隔进程，保证进程空间彼此不受干扰
3. 基于局部性原理进行页面替换 虚拟内存的大小由计算机的地址总线决定 cache名字和TLB命中没有必然联系，是两种独立的机制。 CPU和Cache之间交换的单位是字节，Cache和内存之间交换的单位是块。

coredump文件

- gdb可以用于分析coredump文件。coredump文件含有进程被终止时内存/CPU寄存器和各种函数调用栈的信息。
- 产生coredump文件的原因：

1. 内存访问越界
2. 多线程使用了线程不安全的函数
3. 多线程读写的数据未加锁保护
4. 栈溢出

- core文件没有符号表信息，必须结合可执行文件才可调试

内核态和用户态

- 系统调用
- 中断
- 异常
- 内核态用于执行一些特权指令，比如中断机制，原语，进程管理等，目的是保护系统程序。
- 内核态通常包括三个方面：系统调用，中断和异常
- 中断和异常都可以实现用户态到内核态的切换，是通过硬件实现的。
- 中断：IO完成，定时器时钟中断。
- 异常：越界，溢出，缺页，异常不能被屏蔽，一旦出现应立即处理。

写文件的流程

- fwrite可以把数据写到用户空间缓冲区，但不会立即写到内核
- write内核函数
- fflush把用户缓冲区的数据立即写到内核中的页缓冲区
- 当我们写文件的时候，内核的文件系统模块把数据保存在页缓冲区中，不会立即写到存储设备。我们可以使用fsync把文件修改过的属性和数据立即写到存储设备，或者使用fsyncdata把文件修改过的数据立即写到存储设备。
- 应用程序可以使用glibc库封装的标准IO流函数访问文件。
- 标准IO流提供了缓冲区，目的是尽可能减少调用read和write的次数，提高性能。

数据写入磁盘的过程

- fwrite---application data---clib buffer-----page cache----disk
- fwrite返回的时候，数据还在clib buffer中，调用fclose可以触发文件写入到disk中。
- clib buffer刷新到page cache内核缓冲区可以调用fflush函数(非系统调用)
- page cache内核缓冲区强制刷新到磁盘可以调用fsync函数(系统调用)
- write函数直接从application data拷贝到page cache内核缓冲区

进程

linux文件类型

- -普通文件
- d目录
- l符号连接
- s套接字

- b块文件
- c字符文件
- p管道

什么是进程

- 进程是系统进行资源分配的基本单位，是程序加载到内存后的执行过程。进程一般由数据段，代码段和进程控制块三部分组成。系统通过进程控制块感知进程的存在并对进程进行控制。由于进程之间虚拟地址空间相互独立，多进程比多线程更安全，一个进程基本上不会影响另外一个进程。

进程三种状态

1. 就绪
2. 运行
3. 阻塞

IPC进程间通信

1. 管道
2. 共享内存
3. 消息队列
4. 信号(开销小)
5. 信号量
6. socket

僵尸进程

- 子进程先结束，而父进程没有回收子进程，释放子进程占用的资源，此时子进程将成为一个僵尸进程。

孤儿进程

- 父进程先结束，而子进程仍然存活，此时子进程称为孤儿进程，将由系统的init进程负责回收相关资源。

什么是线程

- 线程是CPU调度的基本单位。一个进程可以包含多个线程，线程占有的系统资源很少，线程可以和同属于一个进程的其他线程共享进程所拥有的全部资源。多线程之间对内存共享，线程间通信可以直接基于共享内存来实现，比多进程之间通信更方便。多线程之间切换不需要切换虚拟内存空间、文件描述符等，所以线程的上下文切换也比多进程轻量。

线程间同步

1. 互斥锁
2. 读写锁(读时共享，写时互斥)
3. 条件变量
4. 信号量(互斥锁的升级版)
5. 自旋锁(可以避免进程或线程上下文的开销)

线程共享资源

1. 文件描述符表（打开的文件）
2. 进程用户ID和进程组ID
3. 进程的内存地址空间
    - .text代码段
    - .data数据段
    - .bss
    - 堆区
    - 全局变量
    - 静态变量
4. 每种信号的处理方式
5. 进程的当前目录

线程独享资源

1. 线程栈
2. 寄存器的值
3. 线程ID
4. 错误返回码errno变量
5. 线程信号屏蔽字
6. 线程优先级

多进程和多线程的应用场景

1. 一般不同任务间需要大量的通信，使用多线程的场景比多进程多。IO密集型。
2. 但是多进程有更高的容错性，一个进程的崩溃不会导致整个系统的崩溃，在任务安全性较高的情况下，采用多进程。CPU密集型。

进程线程的本质区别

1. 进程更安全，一个进程完全不会影响另外的进程。
2. 进程间通信比线程间通信的性能差很多，线程切换开销更低。

进程和线程的区别

- 进程是资源分配的基本单位。一般由代码段，数据段和进程控制块组成。进程控制块记录了进程的ID，打开的文件描述符表，对信号的处理方式等。
- 线程是进程的一个实例。一个进程中可以有多个线程。多个线程间共享进程的内存地址空间。所以线程间的同步就很方便，如条件变量，信号量，套接字。线程独占的资源很少，包括寄存器、栈、错误返回码、线程优先级等。
- 总的来说：进程拥有独立的地址空间，所以一个进程的崩溃不会影响另一个进程。安全性较好。同时进程的通信方式开销比较大，包括共享内存，消息队列，管道，信号量，信号，套接字。大量通信用线程。安全性较高用进程。IO密集用协程。
- 协程是轻量级线程。传统的进程和线程由操作系统去管理，而协程是用户态的。在golang中的GMP模型中，每个goroutine对应一个内核级线程，通过processor去进行调度，当本地队列不足时，到全局队列中获取goroutine。当其中一个goroutine阻塞时，goroutine将让出CPU。

IO

阻塞IO和非阻塞IO

> 阻塞与非阻塞是从IO请求的发起者角度出发，是等待IO（阻塞）执行完成还是立马返回（非阻塞）。

- 阻塞的文件描述符为阻塞IO
- 非阻塞的文件描述符为非阻塞IO

同步IO和异步IO

> 同步与异步是从IO请求的执行者角度出发，执行完成后是应用程序主动询问（同步），还是内核通知应用程序 （异步）。

- 同步IO向应用程序通知的是IO就绪事件。要求用户代码自行执行读写操作，将数据从内核缓冲区读入用户缓冲区。
- 异步IO向应用程序通知的是IO完成事件。由内核来执行IO读写操作。

Reactor模式的工作流程

1. 主线程往epoll内核事件表中注册socket上的就绪事件。
2. 主线程调用epoll_wait等待socket上有数据可读。
3. 当socket上有数据可读时，epoll_wait通知主线程。主线程将socket可读事件放入请求队列。
4. 睡眠在请求队列上的某个工作线程被唤醒，它从socket读取数据，并处理客户请求，然后往epoll内核事件表中注册该socket上的写就绪事件。
5. 主线程调用epoll_wait等待socket可写。
6. 当socket可写时，epoll_wait通知主线程。主线程将socket可写事件放入请求队列。
7. 睡眠在请求队列上的某个工作线程被唤醒，它往socket上写入服务器处理客户请求的结果。

虚拟地址空间

- 虚拟地址是操作系统管理内存的一种方式。
- 方便不同进程使用的虚拟地址彼此隔离。
- 方便物理内存中不相邻的内存在虚拟地址上视为连续的来使用。
- 虚拟地址和物理地址的映射是通过MMU页表进行的。
- 虚拟内存对实际内存有保护作用。

内核态和用户态

- 内核态用于执行一些特权指令，比如中断机制，原语，进程管理等，目的是保护系统程序。
- 陷入内核态的三种情况：
    1. 系统调用
    2. 中断
    3. 异常
- 中断和异常都可以实现用户态到内核态的切换，是通过硬件实现的。
- 中断：IO完成，定时器时钟中断。

- 异常：越界，溢出，缺页，异常不能被屏蔽，一旦出现应立即处理。

进程调度方式

1. 抢占式：立马停止。
2. 非抢占式：时间片用完或者等待资源时，再调用另一个进程。

进程调度算法

1. 先来先服务
2. 短作业优先
3. 优先级调度
4. 时间片轮转
5. 高响应比优先

管道

- 管道是一种伪文件，实质为内核缓冲区大小为4K，内核借用环形队列实现。
- 管道是半双工的，数据只能单向流动，不可重复读取，只能用于有血缘关系的进程。

大端字节序和小端字节序

1. 大端字节序：网络字节序（高位存低位）
2. 小端字节序：主机字节序，现代PC机采用小端字节序（低位存低位，高位存高位）

> 比如0x1f3f5f7f 地址0x1000 0x1001 0x1002 0x1003
>
> 大端法：7f存在0x1003 5f存0x1002 3f存0x1001 1f存0x1000 低存高
>
> 小端法：7f存在0x1000 5f存0x1001 3f存0x1002 1f存0x1003 低存低

socket server

> socket 创建socket文件描述符 bind 绑定IP和端口号 listen 设置监听连接数的上限
> accept 阻塞等待数据到来 read/write 处理客户端的业务 close 关闭文件描述符

socket client

> socket 创建套接字文件描述符 bind 绑定IP和端口号（也可以隐式绑定） connect 尝试连接服务器（建立TCP连接） write/read 处理服务器端的业务

五种网络IO模型

1. 同步阻塞IO
2. 同步非阻塞IO
3. IO多路复用
4. 信号驱动IO
5. 异步IO

协程

- 协程是一种用户态的轻量级线程。
- 协程的开销远远小于线程的开销。
- 协程是一种比线程更加轻量级的存在，一个线程可以拥有多个协程。
- 无论是进程还是线程，都是由操作系统所管理的。而协程不是被OS所管理，而完全是由程序所控制(也就是在用户态执行)。

信号

- 信号是一种不精确通信。
- 常用的信号有SIGKILL 9 无条件终止信号
- SIGSEGV 11 无效存储访问
- SIGPOLL 8 轮询事件信号。
- 信号有三种处理方式：忽略，捕获，默认。
- kill命令向进程发送信号

什么是死锁

- 因为资源调度的方式不合理或者资源的稀缺性，导致进程间的相互等待。

- 死锁的四个必要条件：

1. 互斥条件
2. 请求和保持条件
3. 环路等待条件
4. 不可剥夺条件

- 死锁的预防只要破坏死锁产生的四个必要条件。通常采用预先静态分配方法，可以破坏请求和保持条件。
- 死锁的避免：采用银行家算法，只要系统处于安全状态，系统便可避免死锁。
- 死锁的解决：撤销进程，剥夺资源。
- 死锁检测：依赖关系图是否有环。

fork函数

- fork函数用来创建子进程 一次调用，两次返回。在父进程中返回子进程的PID，在子进程中返回0

exec族

- 在程序中调用另一个可执行程序，但是进程ID不改变。

mmap存储映射

- 将磁盘空间映射到进程空间，使进程可以采用指针的方式操作这段内存，而不用调用read和write函数。
- 提高了读写的效率，同时也可以实现进程间的通信。

异步IO原理

- 底层将数据准备好后，内核会给进程发送一个异步通知信号SIGIO29通知进程，然后进程调用信号处理函数去读数据，没准备好，数据就忙自己的事情。

文件

- open/read/write操作文件描述符三连

```
ret = read(fds[0], buf, sizeof(buf)); //返回读取到的字节数
write(fds[1], str, strlen(str));
```

fork

```
int pid = fork()
if(pid > 0){} 父进程
else if(pid == 0){}子进程 -1表示失败
```

pipe

- 本质是内核缓冲区，使用环形队列实现，默认大小是4k。
- 半双工，单方向流动数据
- 一读一写

```
itn fds[2];
int ret = pipe(fds);
0表示成功 -1表示失败
```

exec

- fork出子进程后，希望子进程执行另外一个可执行程序。
- 调用exec函数以后，进程的用户空间数据（.text和.data）被新程序所替换,进程id不改变。

```
execlp("ls","随便什么都可以","-l","-a",NULL); //最后一定要传入NULL用于指示不定长参数的结尾
```

信号

- 信息量比较小，开销小，不可靠，有时延。每个进程收到的所有信号，都是由内核负责发送的，内核处理。
- 产生信号：kill raise abort 段错误 除0 按键
- 信号机制：信号屏蔽字（阻塞信号集），未决信号集。放在PCB进程控制块中。
- 未决信号集bitmap
- 处理方式：忽略/捕获/默认

- 注册信号捕捉函数
- SIGSEGV
- SEGKILL
- SIGALRM
- SIGINT

```
void my_handler(int sig){
    printf("hello world!\n");
    abort(); //终止进程
}
int main(){
    signal(SIGALRM, my_handler); //注册信号捕捉函数
    alarm(1); //定时
    for(int i = 0; ; i++){
        printf("%d\n",i);
    }
    return 0;
}
```

alarm

定时器，只能支持到秒级，与进程状态无关，进程挂起时也在计时中。

每个进程都有且只有一个唯一个定时器，二次调用覆盖原来的定时器，返回原来定时的剩余时间。

```
alarm(5) //5秒后程序终止
alarm(0) //取消闹钟
```

setitimer微秒级定时

动态分配内存

- 分配策略
- 首次适应：地址递增最先满足条件的内存
- 最佳适应：容量递增最先满足条件的内存
- 循环适应：首次适应的优化版，从上次分配的位置开始。

分页内存管理

- 固定分区会产生内部碎片，动态分区会产生外部碎片。分页使用相等的页面大小，内存和进程都进行划分，由于页面较小，所以每个进程平均产生半个页的内部碎片。分页管理是从计算机的角度考虑设计的，页面的大小对用户是透明的。分段管理是从用户的角度去设计的，为了满足编程的方便，信息保护和共享等多方面的需要。
- 逻辑地址：页号P + 页内偏移量W

- 物理地址：b*L + W
- 页表项：页号P + 物理内存的块号b
- 页面大小必须相等
- 页表：由页表项组成。
- 地址转换机构：逻辑地址---快表TLB---页表---物理地址
- 两次访问主存，如果使用了快表就可以实现只访问一次主存拿数据和指令
- 多级页表：减少页表所占的连续内存空间。

分段内存管理

- 逻辑地址：段号S + 段内偏移量W
- 段表：由段表项组成。
- 段表项：段号S + 段长L + 起始地址b
- 物理地址：b + W (需要判断b是否小于等于L)

段页内存管理

- 逻辑地址：段号S + 页号P + 页内偏移量W
- 每个进程有一个段表，每个段有一个页表。
- 段页式存储既有分页也有分段的优点，采用分段来分配和管理用户地址空间，用分页来管理物理存储空间，但它的开销最大。

虚拟内存

- 局部性原理：时间局部性和空间局部性。
- 部分页面装入内存
- 缺页中断机制
- 页面置换算法：

1. 最佳置换算法
2. 最近最少使用
3. 先进先出
4. 时钟算法

- 抖动：页面频繁调度
- Belady问题：FIFO中随着物理块的增加，缺页故障反而增加。

进程调度算法

- 先来先服务：属于不可剥夺算法，对长作业有利，对短作业不利，有利于CPU繁忙，不利于IO繁忙。
- 短作业优先：对于长作业会产生饥饿，即长期不被调度。
- 优先级调度：从就绪队列中选择优先级最高的进程，分配CPU
- 高响应比优先调度：主要用于作业调度，克服了饥饿状态，兼顾了长作业。
- 时间片轮转：主要适用于分时系统，剥夺式算法
- 多级反馈队列

实践篇

文件描述符限制

- 一个进程打开的文件描述符上限是1024
- ulimit -u 获取文件描述符个数
- im /etc/security/limits.conf 打开该文件修改进程的文件描述符上限
- 查看系统的文件描述符上限 cat /proc/sys/fs/file-max
- select的数组大小要修改只能重新编译内核

自旋锁

- 保护的临界数据处理的时间尽可能短，否则很浪费CPU
- 在多CPU的系统上，自旋锁才有价值。
- 执行自旋锁内部代码时不能主动让出CPU，否则会引起死锁等问题。
- 自旋锁内是不允许存在信号量操作的，反之，信号量保护的代码里面是可以有自旋锁操作的。

伙伴系统

- buddy system
- 基于2的幂次开辟内存。
- 释放时自动合并成更大的空间。

slab机制

- 是linux系统的一种内存分配机制。它是针对经常分配和释放的对象。slab分配器基于对象进行管理，每次从slab列表中分配一个同样大小的单元。释放时不直接返回给伙伴系统，而是保存在slab列表中。

僵尸进程

- 子进程死了，父进程没有进行回收。waitpid回收指定进程。

孤儿进程

- 父进程死了，子进程仍然存活。系统会让init进程领养孤儿进程。

内存空间分布

- 内核区
- 用户区
    - 环境变量
    - 命令行参数
    - 栈区
    - 共享区【加载动态链接库或者建立内存映射】
    - 堆区
    - .bss段未初始化变量
    - .data段初始化数据
    - .text段代码段
    - 0-4k的受保护区域NULL

零拷贝技术

- IO系统调用：read/write用户态没有缓冲区

- 标准IO库：fread/fwrite用户态有缓冲区
- 零拷贝：是指不用将数据从内核态到用户态的频繁拷贝，节省了CPU的周期和内存。
- sendfile发送文件时只需要一次系统调用
    - 将数据从磁盘读取到内核缓冲区
    - 在socket buffer中记录内核缓冲区的位置和偏移量
    - 根据socket buffer中的记录将数据copy到网卡设备中
- 避免了内核态到用户态的频繁拷贝，减少系统调用的次数，降低了上下文切换的开销。
- mmap：将磁盘上的物理空间映射到共享区，避免了read和write的系统调用，可以直接通过指针操作文件。

上下文切换开销

- 页表目录项更新
- 寄存器中的数据
- 切换内核态栈
- 刷新TLB
- L1到L3的缓存间接失效，直接访问内存。

.C文件到ELF文件的流程

- 预编译
- 编译
- 汇编
- 链接

ELF文件执行的流程

- 核心函数是load_elf_binary，它通过读存放在ELF文件中的信息为当前进程建立一个新的执行环境。
- elf是静态文件，程序执行时所需要的指令和数据必需在内存中才能够正常运行。load_elf_binary 就是将elf里的指令和数据加载到内存中。
- 进程的创建
    - 创建一个独立的虚拟地址空间（先共享父进程的页框，即COW机制)
    - 读取可执行文件头，并且建立虚拟空间与可执行文件的映射关系（在子进程需要加载新elf文件时）
    - 将CPU的指令寄存器设置成可执行文件的入口地址，启动运行
- 完整流程
    - shell首先fork一个子进程，子进程通过execve系统调用启动加载器
    - 读取elf文件头
    - 调用load_elf_binary函数加载代码段和动态链接库，分配虚拟地址空间。
    - 将CPU的指令寄存器设置成可执行文件的入口地址，启动运行。
    - 往磁盘上写入数据，调用write函数，触发系统调用sys_write，写入内核缓冲区，可以使用fsync强制刷新到磁盘。

文件系统的主要功能

- 创建
- 删除
- 读写

- 挂载

VFS 虚拟文件系统

- 为了屏蔽底层文件系统和驱动程序等细节，提供友好的统一用户接口
- EXT4 块文件系统
- FAT
- swap 虚拟内存的文件系统

日志文件系统

- 日志文件系统可以在系统发生断电或者其它系统故障时保证整体数据的完整性

打开文件描述符表

打开文件描述符表，简称为打开文件表。 当进程打开一个文件的时候，虚拟文件系统会创建文件的一个打开实例：file结构体，然后在进程的打开文件表中分配一个索引inode，这个索引称为文件描述符，最后把文件描述符和file结构体的映射添加到打开文件表中。 file结构体包括：

- inode对象指针
- 文件偏移量
- 访问模式

> 打开文件表是PCB进程控制块的一部分，进程控制块是个task_struct结构体包括：

- 进程号
- 打开文件描述符表
- 等等

文件系统

文件系统应具有以下功能

1. 完成文件存储系统的管理，即分配空间和回收空间。
2. 实现文件名到物理地址的映射。
3. 实现文件和目录的建立，读写管理。
4. 向用户提供有关文件和目录操作的接口。

文件系统层次结构

- 用户空间
- 内核空间
    - 虚拟文件系统VFS
    - 块设备文件系统、内存文件系统、闪存文件系统
    - 页缓存 Page Cache
    - 块设备层
    - 块缓存
    - IO调度器
    - 块设备驱动程序
- 硬件

    - 机械硬盘、固态硬盘
    - 闪存

IO调度算法

- NOOP：FIFO
- CFQ完全公平：该算法为每个进程分配一个时间窗口，在时间窗口内允许进程发出IO请求
- Deadline：每个IO请求有一个最后执行期限。

分页管理

- 从进程虚拟地址空间映射到内存地址空间。
- 页面大小相同，内存和进程都进行划分。由于页面较小，每个进程平均产生半个页的内部碎片。
- 逻辑地址：页号 + 页内偏移
- 地址转换：逻辑地址---快表TLB---页表----物理地址
- 两次访问主存，如果使用了快表就可以实现只访问一次主存拿数据和指令

虚拟文件系统VFS

- 对用户友好，屏蔽底层细节。
- 管理和存储文件
- 文件系统的主要功能：安装，卸载，创建，删除，读写。
- 文件系统的结构分为三层：用户层，内核层，硬件层。
- 用户层的函数：fopen fclose fwrite fflush fread fseek
- 内核层函数：mount unmount reade write open lseek fsync fdatasync
- 内核中有个页缓存：page cache

mount挂载

- 一个存储设备上的文件系统，只有挂载到内存中目录树的某个目录下，进程才能访问这个文件系统。

linux上运行可执行文件的过程

- a.out是ELF文件格式
- linux每个程序都会运行在一个进程上下文中，这个进程上下文有自己的虚拟地址空间。这个进程是由shell进程fork出来的子进程。

内存管理

- 分页内存管理
    - 内存和进程划分为大小相等的页面，通常是4k，页面大小是对用户透明的，分页是从计算机的角度去考虑的，为了减少内存碎片，由于页面较小，每个进程平均产生半个内存碎片。
- 分段
    - 分段是从用户的角度考虑，为了满足编程的方便，信息保护和共享等方面的要求。
- 段页
    - 段号+页号+页内偏移

虚拟地址空间

- 基于局部性原理，使用页面置换算法，是内存的可用空间更大。
- 进程隔离
- 内存的管理
- 不连续的空间

读写文件的方式

- 调用内核提供的系统调用read/write
- 调用glibc库封装的标准IO流函数fread/fwrite。用户空间有缓冲区，能减少系统调用的次数，提高性能。
- 创建基于文件的内存映射，把文件的一个区间映射到进程的虚拟地址空间，然后直接读内存。mmap避免系统调用，性能最高。

IO控制方式

- 轮询
- 中断 CPU字节干预IO
- 直接存储器访问DMA 一个块数据 干预一次
- 通道 一组块数据 干预一次
- 目的：减少CPU对IO的干预，提高其处理数据的效率。

中断机制

- 外设速度远远慢于CPU的速度，所以需要外设资源准备好以后，利用中断机制主动通知操作系统。
- 内部中断/异常：CPU执行指令期间检测到非法的条件（除零，地址访问越界）
- 硬中断/外中断/中断：IO中断，时钟中断，信号中断产生的时间不确定。
- 软中断：应用程序使用系统调用而引发的事件。
- 中断描述符表
    - os中预先设置一些中断处理函数，当CPU接收到中断时，会根据中断号去查对应的处理函数，中断向量表就是记录中断号与中断函数映射关系的表。
- 中断机制是为了弥补CPU速度和外设速度数量级差异的机制，它的核心是中断向量表。

VFS中的数据结构

- 超级块 super block
- 索引节点 iNode
- 目录项 dentry
- 文件 file

超级块Super Block

- 描述文件系统的总体信息，挂载文件系统时在内存中创建超级块的副本。
- 一个文件系统只有挂载到内存中目录树的一个目录下，进程才能访问这个文件系统。
- 每次挂载文件系统，VFS会创建一个挂载描述符：mount结构体，并读取文件系统的超级块，在内存中创建超级块的副本。
- 每种文件系统的超级块的格式不同，需要向VFS注册文件系统类型，并实现mount方法用来读取和解析超级块。
- ext4----register----mount-----super block

索引节点iNode

- 每个文件对应一个索引节点，每个索引节点有个唯一的编号。当内核访问存储设备上的一个文件时，会在内存中创建索引节点的副本：结构体inode
- 一个块由多个扇区组成4k=8sector
- inode文件的元信息：创建者，创建日期，文件的大小。
- 查看系统中iNode的用量df -i
- df -h 输出文件系统分区情况
- du -h 文件、目录

目录项

- 文件系统把目录看作文件的一种类型，目录的数据是由目录项组成，每个目录项存储一个子目录或文件的名称以及对应的索引节点号。

打开文件表

- 当进程打开一个文件的时候，VFS会创建文件的一个打开实例：file结构体，然后在进程的打开文件描述符表中分配一个索引，这个索引被称为文件描述符，最后把文件描述符和file结构体的映射添加到文件描述符表中。