1.进程，线程(27%)

基础概念解释(15%)

进程，线程，协程，IPC，中断，syscall的ABI

进程(process)： 内核可知

- 每个进程都有自己独立的地址空间和系统资源。
- 进程之间通过进程间通信（IPC）来进行数据交换和协调，例如管道、消息队列等。
- 进程间的切换代价相对较高，因为切换需要保存和恢复大量的上下文信息。

线程(thread)： 内核可知

- 线程是在进程内部执行的轻量级执行单位。一个进程可以包含多个线程，它们共享进程的内存和系统资源。
- 线程共享进程的地址空间和文件描述符等资源，因此线程之间的通信更加简单和高效。
- 线程之间的切换开销相对较低，因为它们共享相同的上下文和内存空间。

协程(coroutine)： 内核不可知，即它不可以被中断等操作被动切换，只能等它自己主动切换出去。主要用于一些轻量级的计算

IPC(Inter-Process Communication,进程间通信)： IPC的主要目标是允许不同进程之间进行有效的通信和同步，以便实现数据共享、协作和协调。

以下是几种常见的IPC机制：

- 管道（Pipe）：管道是一种最基本的IPC机制，它提供了单向的、字节流的通信方式。管道可以用于父子进程之间或者具有亲缘关系的进程之间进行通信。
- 命名管道（Named Pipe）：与普通管道类似，但命名管道可用于无关的进程间通信，即使它们没有亲缘关系。
- 信号量（Semaphore）：信号量是一种用于进程同步的计数器。它可以用于多个进程之间的互斥和同步操作。
- 共享内存（Shared Memory）：共享内存允许多个进程访问相同的物理内存区域，从而实现高效的数据共享。进程可以直接读写共享内存，而无需复制数据。
- 消息队列（Message Queue）：消息队列是一种在进程之间传递数据的方式，进程可以将消息放入队列并从中读取。消息队列可以实现独立的、异步的进程间通信。
- 套接字（Socket）：套接字提供了一种网络编程接口，允许不同主机上的进程进行通信。套接字可以在本地主机上的进程间进行通信，也可以在网络上的远程主机间进行通信。

中断：

中断是一种异步事件，它可以来自于外部设备的请求或其他系统事件，例如键盘输入、鼠标点击、时钟信号等。当中断事件发生时，处理器会暂停当前执行的程序，跳转到预先定义的中断处理程序执行相应的操作，处理完毕后再返回到被中断的程序继续执行。

异常：

异常是一种同步事件，它通常是由程序执行过程中的错误、异常情况或非法操作引发的。例如，除以零、无效的内存访问、越界访问等。当异常事件发生时，处理器会暂停当前执行的程序，转而执行与异常相关的异常处理程序来处理异常。异常处理程序通常用于错误处理、恢复程序状态或终止程序的执行。

syscall的ABI(Application Binary Interface)：

系统调用（syscall）的ABI定义了应用程序如何调用系统调用以及系统调用的参数传递和返回值处理方式。

- 调用约定：指定了应用程序如何将参数传递给系统调用并获取返回值。这包括参数传递的顺序、寄存器的使用方式、栈的使用方式等。
- 寄存器使用规则：定义了哪些寄存器用于传递系统调用的参数，以及哪些寄存器用于保存系统调用的返回值。
- 系统调用号的传递方式：规定了如何将系统调用的标识符（系统调用号）传递给操作系统内核，以便执行相应的系统调用操作。
- 系统调用的异常处理：定义了当系统调用执行过程中出现异常或错误时，应用程序和操作系统如何进行处理和返回相应的错误码。

A.2.1 调用约定 Linux AMD64 内核在内部使用与用户级应用程序相同的调用约定（详见第3.2.3节）。想要调用系统调用的用户级应用程序应该使用C库中的函数。C库与Linux内核之间的接口与用户级应用程序相同，但存在以下区别：

1. 用户级应用程序使用整数寄存器顺序为 %rdi、%rsi、%rdx、%rcx、%r8 和 %r9 来传递参数。内核接口使用 %rdi、%rsi、%rdx、%r10、%r8 和 %r9。
2. 通过 syscall 指令进行系统调用。内核会破坏寄存器 %rcx 和 %r11。
3. 系统调用的编号必须在寄存器 %rax 中传递。
4. 系统调用的参数限制为六个，不会直接在堆栈上传递参数。
5. 从系统调用返回时，寄存器 %rax 中包含系统调用的结果。在 -4095 到 -1 的范围内的值表示错误，它等于 -errno。
6. 只有 INTEGER 类型或 MEMORY 类型的值会传递给内核。

其它问题如：一个CPU是多少核？ 会结合linux的输出来问(输出是什么)

```
(base) zhongzero@zero:/mnt/c/Users/zhongzero/Desktop$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              2
Core(s) per socket:              4
Socket(s):                       1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           142
Model name:                      Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
Stepping:                        10
CPU MHz:                         1799.999
BogoMIPS:                        3599.99
Hypervisor vendor:               Microsoft
Virtualization type:             full
L1d cache:                       128 KiB
L1i cache:                       128 KiB
L2 cache:                        1 MiB
L3 cache:                        6 MiB
Vulnerability Itlb multihit:     KVM: Mitigation: VMX unsupported
Vulnerability L1tf:              Mitigation; PTE Inversion
Vulnerability Mds:               Vulnerable: Clear CPU buffers attempted, no microcode; SMT Host state unknown
Vulnerability Meltdown:          Mitigation; PTI
Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass disabled via prctl and seccomp
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2:        Mitigation; Full generic retpoline, IBPB conditional, IBRS_FW, STIBP conditional, RSB fill
                                 ing
Vulnerability Srbds:             Unknown: Dependent on hypervisor status
Vulnerability Tsx async abort:   Not affected
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr s
                                 se sse2 ss ht syscall nx pdpe1gb rdtscp lm constant_tsc rep_good nopl xtopology cpuid pni
                                 pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 movbe popcnt xsave avx f16c rdrand hypervisor
                                 lahf_lm abm 3dnowprefetch invpcid_single pti ssbd ibrs ibpb stibp fsgsbase bmi1 avx2 smep
                                 bmi2 erms invpcid rdseed adx smap clflushopt xsaveopt xsavec xgetbv1 xsaves flush_l1d arch
                                 _capabilities
```

`cat /proc/cpuinfo` （more detail）

stepping：步长，即CPU的更新版本

cpu family：CPU产品系列代号

model：CPU属于其系列中的哪一代的代号

model name：cpu型号

CPU(s)：逻辑cpu颗数

On-line CPU(s) list：正在运行的cpu逻辑内核

Thread(s) per core：每个核的线程数(每个 Core 的硬件线程数)

Core(s) per socket：多少核

Socket(s)：服务器面板上有几个cpu槽位

cpu MHz：CPU的时钟频率（主频）

CPU max MHz：cpu时钟最大频率

CPU min MHz：cpu时钟最小频率

bogomips：在系统内核启动时粗略测算的CPU速度（Million Instructions Per Second）

Address sizes：记录物理地址和虚拟地址的大小

Virtualization：cpu支持的虚拟化技术

frequency boost：是否支持频率提升

为什么物理地址/虚拟地址的大小不是64位?

当前的计算机系统中，64位地址空间的范围非常大（2^64），远远超出了实际的内存容量需求。因此，为了节省内存和提高效率，操作系统通常会限制地址空间的大小。(RAM的大小与物理地址空间有关，而不与虚拟地址空间有关)

小端和大端的区别

大端：低地址存放高字节数

小端：低地址存储低字节数

部分syscall的执行过程，系统的角色(信号绑定和调用syscall本身两部分)(4%)

syscall是用户态和内核交互的过程

信号绑定是用户态告诉内核我现在遇到了什么事情要做什么事情

syscall的整体流程：

- 应用程序发起系统调用(应用程序通过调用特定的系统调用函数发起系统调用请求)；
- 用户态到内核态切换；
- 内核处理系统调用(操作系统内核接收系统调用请求，并根据系统调用号识别请求的类型。内核会执行相应的系统调用处理程序来完成请求的操作)；
- 内核执行系统调用(内核执行所需的操作，这可能包括在文件系统中读写文件、进行网络通信、创建或销毁进程、管理内存等)；
- 返回结果给应用程序；
- 内核态到用户态切换

信号绑定的整体流程：

- 定义信号处理函数；
- 注册信号处理函数(通过调用操作系统提供的函数（如 signal() 或 sigaction()）将信号处理函数与特定的信号关联起来)；
- 信号发生时的处理(当注册的信号发生时，操作系统会中断当前的程序流程，并开始执行与该信号关联的信号处理函数。这个过程通常是在内核态中执行)；
- 信号处理函数执行(操作系统将控制权转移到信号处理函数，并执行该函数的代码)；
- 信号处理完成后的操作(信号处理函数执行完成后，控制权返回到原来的上下文中，即被中断的程序流程继续执行)；

系统调用用于应用程序通过系统接口请求操作系统提供的服务和功能。

信号绑定用于捕获和处理特定的信号事件，通过将信号与处理函数关联来实现。

(信号是一种软件中断，是由操作系统发出的中断信号，被程序接收后执行相应的操作。)

在syscall过程中，系统的任务：

操作系统在syscall过程中起着关键的作用，充当了应用程序和底层硬件之间的中介者。它提供了系统调用接口、处理系统调用请求、访问系统资源、调度和执行操作等功能，以实现应用程序与底层系统的交互和协调。

- 操作系统提供了系统调用接口
- 当应用程序发起系统调用请求时，操作系统的内核接收并处理这些请求
- 用户态到内核态切换(这个切换是由操作系统的内核管理)
- 访问系统资源时，操作系统确保对这些资源的访问和操作是安全和可控的
- 调度和执行(操作系统负责调度和执行相应的系统调用处理程序。它管理计算资源的分配，以确保系统调用得到及时响应，并且不会影响到其他正在运行的进程)
- 返回结果给应用程序

性能分析，如果线程过多的时候会怎么样(性能分析共8%)

real time先基本不变后增加，user time一直在增加(sys time基本为0就不考虑了)

real time变化的原因：增加线程数会导致更多的线程同时竞争处理器资源、内存等系统资源。当线程数较少时，这些资源可能没有明显的竞争，因此实际运行时间保持不变。但随着线程数的增加，资源的争用程度逐渐加剧，导致了更多的竞争条件和锁等待，从而增加了实际运行时间。

real time,user time,sys time区别

real time：是墙上时间(wall clock time)，也就是进程从开始到结束所用的实际时间。这个时间包括其他进程使用的时间片和进程阻塞的时间（比如等待I/O完成）。 user time：指进程执行用户态代码（核心之外）所使用的时间。这是执行此进程所消耗的实际CPU时间，其他进程和此进程阻塞的时间并不包括在内。 sys time：指进程在内核态消耗的CPU时间，即在内核执行系统调用所使用的CPU时间。

对于单核情况：real > user + sys

多核情况：可能出现 real < user + sys

举个例子，一个纯计算任务，没有系统调用（即没有耗费sys time，只有user time），采用单线程需要执行的时间为8s。那么在一个四核的机器上，采用并行算法，用4个核一起算，理想上如果完全并行，加速比为4，2s内就能完成，那么从开始到结束，real time是2s。user time呢？应该是2s * 4 = 8s. 所以此时会出现real < sys + user的情况。

关闭turbo和打开turbo的区别

turbo为涡轮增压器。打开turbo会导致：

- CPU利用率的提升导致了所需功耗的提升，在功耗没有触及到天花板（TDP），系统会尽可能的提升CPU频率，除非频率已经触及到天花板（max turbo频率）。
- 利用率和功耗继续提升触及天花板之后，系统会主动降频适应功耗或者说散热要求

而关闭turbo会使电脑保持在基频运行(也可能是最大频率)

所以打开turbo可能会导致性能测试发生抖动，且可能会大幅度提升性能

性能分析，swap的开销怎么样(性能分析共8%)

在计算机中，"swap" 是指操作系统将内存中的部分数据或进程移动到硬盘上的一种机制。它用于提供额外的虚拟内存空间，以满足系统中多个进程同时运行时对内存的需求。

- 硬盘I/O开销：Swap 是使用硬盘空间来模拟内存的一部分，因此需要进行硬盘的读写操作。相比于内存的快速访问，硬盘的读写速度较慢，会导致额外的延迟和开销。
- 上下文切换开销：当操作系统将进程的页从内存交换到硬盘上的 Swap 空间时，会涉及到上下文切换的开销。这包括保存和恢复进程的上下文信息，例如寄存器状态、页表等，以及切换进程的页表映射。
- 页面调度开销：操作系统需要决定哪些内存页面应该被交换到 Swap 中，以及何时进行页面的换入和换出操作。这涉及到页面调度算法的开销，例如页面置换算法（如LRU）的执行。
- 系统性能下降：当系统过度依赖 Swap 时，由于硬盘的较低速度和较高的访问延迟，可能导致系统整体性能下降。特别是对于访问频繁的应用程序或需要大量内存的任务，频繁的 Swap 操作会显著影响系统的响应速度和吞吐量。

切入内核（进入内核态）需要进行以下修改：

1. 特权级别：切入内核态时，处理器需要将当前运行的任务从用户态切换到内核态。这涉及修改处理器的特权级别，从用户态切换到内核态的特权级别。
2. 栈切换：由于内核态执行的代码和用户态执行的代码可能使用不同的栈空间，切入内核态时需要切换栈，以确保在内核态下能够正确保存和恢复上下文信息。
3. 上下文切换：切入内核态需要保存当前任务的上下文信息，包括通用寄存器、程序计数器和其他相关寄存器的值。这些上下文信息将被保存在任务的控制块（也称为内核栈）中，以便在切换回用户态时能够正确恢复执行状态。
4. 寄存器修改：切入内核态时，处理器需要修改寄存器的状态，以便在内核态下执行相应的内核代码。这可能涉及到修改特定寄存器的值，以提供正确的参数和状态信息。

2.优先级反转(6%)

三个线程A,B,C，优先级 $A > B > C$

A 和 C 共用一个锁，现在 C 占用了锁， A 只能yield，等在 C 结束

但是由于 C 的优先级很低，它和 B 相比 B 会占用更多资源，所以导致 A 虽然权限高却反而得等 B 的执行，由此发生了优先级反转

一个解决办法是：当很多线程同时申请同一把锁的且某一个线程占用了这把锁时，把其中线程最高的优先级继承给它，使得它就算是一个低优先级的线程，此时也可以享有高优先级，因而解决优先级反转

3.内存管理(14%)

基础概念解释(5%)

页表上的地址是什么地址?

对于普通页表，它就是从虚拟页到物理页的翻译

对于多级页表，它把虚拟地址切分成几块，依次寻找下一级。

多级页表中每一层中存储的都是物理地址，但不是我们要的最终的物理地址，我们通过对它存储的物理地址处的下一级页表进行进一步查表，不断找，之后最终找到我们要的物理地址(所以我们也可以称我们前几次找的是相对于最后一次找到的物理地址的虚拟地址)

页表基地址上面存的是什么地址，到底是物理地址还是虚拟地址?

页表基地址（Page Table Base Address）是指用于定位页表的起始地址。在虚拟内存管理中，页表用于将虚拟地址映射到物理地址，因此需要通过页表基地址找到页表的位置。

页表基地址上面存的是物理地址。

我malloc返回了一个地址p，这个地址是物理地址还是虚拟地址?

malloc返回的地址是虚拟地址。

后续对该虚拟地址的读写操作会通过MMU将其映射到物理地址，从而访问真正的物理内存。

物理内存管理(5%)

内核到底感知感知不到物理页的存在，它需不需要感知物理页的存在?（Buddy伙伴这个东西到底应该放在什么地方）

内核是操作系统的核心部分，负责管理计算机的硬件资源和提供各种系统服务。物理页是实际的物理内存单元，内核需要感知物理页的存在以便进行内存管理和操作。

内核需要通过物理页的存在来进行以下操作：

- 内存分配和回收：内核需要知道哪些物理页已经被分配和正在使用，以及哪些是空闲的可供分配的。内核通过物理页的状态来进行内存分配和回收的管理。
- 虚拟内存映射：内核维护页表，将虚拟地址映射到物理地址。为了进行映射，内核需要知道物理页的位置和状态。
- 内存保护和权限管理：内核通过物理页的存在和属性来管理内存的访问权限，以确保不同进程之间的内存隔离和保护。
- 页面置换：当物理内存不足时，内核可能需要进行页面置换，将部分物理页的内容写入磁盘交换空间。内核需要感知物理页的存在和状态，以决定哪些物理页需要置换出去。

实际上，就算是微内核，它仍然保留了Virtual Memory的模块，仍然需要保存物理页的存在

MMU(内存管理单元)用来管理页表翻译，它是由谁来配置的?

系统，或者说内核

KPTI,Meltdown(4%)

Meltdown是一种计算机安全漏洞，于2018年首次公开披露。它是一种侧信道攻击，利用了现代处理器的一种设计缺陷，可以绕过硬件和操作系统的内存访问权限限制，从而允许恶意程序访问操作系统内核内存中的敏感数据。

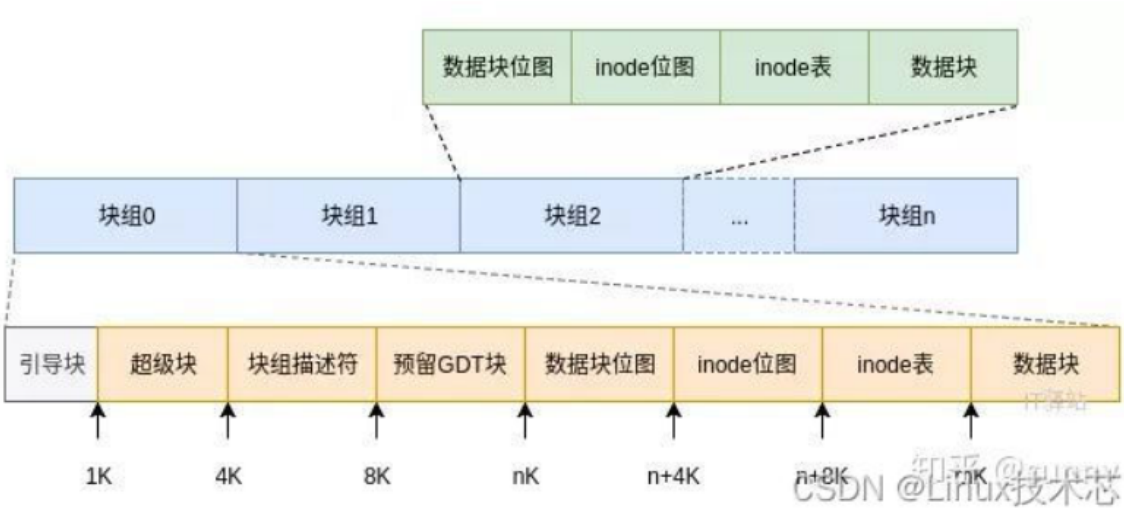
漏洞的根本原因是现代处理器在提高性能的同时，对指令的执行进行了优化，其中一项优化是在执行指令时预取并缓存可能需要的数据。利用现代处理器的乱序执行和高速缓存的特性，恶意程序可以通过执行特定的指令序列，绕过操作系统的内存访问权限限制，从而访问操作系统内核的敏感数据。

KPTI（Kernel Page Table Isolation）是一种用于提高操作系统内核安全性的技术，最初由Google提出。它旨在减轻针对操作系统内核的侧信道攻击风险，特别是针对Meltdown漏洞的攻击。

KPTI的核心思想是将操作系统内核的页表进行隔离。传统上，操作系统内核和用户空间共享同一组页表，这意味着用户程序可以直接访问内核的内存页。通过KPTI，操作系统会为内核和用户空间分别创建独立的页表，并在用户程序执行时切换页表。这样，用户程序就无法直接访问内核内存，从而阻止了Meltdown漏洞的利用。

4.文件系统 (21%)

基础概念解释(ext系列的inode,superblock,journal)(6%)



inode (index node，索引节点)：

在索引节点上存储的部分数据即为文件的属性元数据及其他少量信息。

- 文件的字节数
- 文件拥有者的User ID
- 文件的Group ID
- 文件的读、写、执行权限
- 文件的时间戳，共有三个：ctime表示inode上一次变动的时间，mtime表示文件内容上一次变动的时间，atime表示文件上一次打开的时间
- 链接数：指有多少文件名指向这个inode
- 文件数据的block号码（在文件的block数量很大时，通常会采用多级block来记录block号码，这里采用bmap标记未使用的inode号码。）

每一个文件都对应一个inode。

superblock：记录文件系统的整体信息，包含inode/block的大小、总量、使用量、剩余量，以及文件系统的格式，文件系统挂载时间，最近一次数据写入时间，最近一次校验磁盘的时间等。主要概括为 EXT文件系统的全局配置参数（如数据块大小、总块数和索引节点总数等，这些参数在文件系统创建时就确定下来了）和动态信息（如当前空闲块数和空闲索引节点数等，可以在运行中改变）。

superblock理论上只需要在第一个块组中记录一次即可，但是由于superblock太重要了，所以它实际在好多块组中都有记录，作为备份。

GDT(Group Descriptor Table,块组描述符表)：由很多块组描述符组成，整个分区分成多少个块组就对应有多少个块组描述符。每个块组描述符存储一个块组的描述信息，如在这个块组中从哪里开始是inode表、从哪里开始是数据块、空闲的inode和数据块还有多少个等

块组描述符表同样在好多块组中都有记录，作为备份，这些信息是非常重要的，一旦块组描述符表意外损坏就会丢失整个分区的数据

bmap(块位图)：它本身占一个块，其中的每位代表本块组中的一个块，该位为1表示该块已用，为0表示该块空闲可用。

imap(节点位图): 与块位图类似，本身占一个块，其中每位表示一个inode是否空闲可用。

inode table(节点表): 每个文件都有一个inode，一个块组中的所有inode组成了inode表。

inode表占多少个块在格式化时就要决定并写入块组描述符中，mke2fs格式化工具的默认策略是一个块组有多少个8KB就分配多少个inode。由于数据块占了整个块组的绝大部分，也可以近似认为数据块有多少个8KB就分配多少个inode

数据块： 根据不同的文件类型有以下几种情况：

- 对于常规文件，文件的数据存储在数据块中。
- 对于目录，该目录下的所有文件名和目录名存储在数据块中，注意文件名保存在它所在目录的数据块中，除文件名之外，ls -l命令看到的其他信息都保存在该文件的inode中。注意这个概念：目录也是一种文件，是一种特殊类型的文件。(**文件名和inode号不是存储在其自身的inode中，而是存储在其所在目录的data block中。**)
- 对于符号链接，如果目标路径名较短则直接保存在inode中以便更快地查找，如果目标路径名较长则分配一个数据块来保存。
- 设备文件、FIFO和socket等特殊文件没有数据块，设备文件的主设备号和次设备号保存在inode中。

如何根据inode号找到inode:

inode结构自身并没有保存inode号（同样，也没有保存文件名），那么inode号保存在哪里呢?

目录的data block中保存了该目录中每个文件的inode号。

另一个问题，既然inode中没有inode号，那么如何根据目录data block中的inode号找到inode table中对应的inode呢?

实际上，只要有了inode号，就可以计算出inode表中对应该inode号的inode结构。在创建文件系统的时候，每个块组中的起始inode号以及inode table的起始地址都已经确定了，所以只要知道inode号，就能知道这个inode号和该块组起始inode号的偏移数量，再根据每个inode结构的大小(256字节或其它大小)，就能计算出来对应的inode结构。

数据块寻址方法:

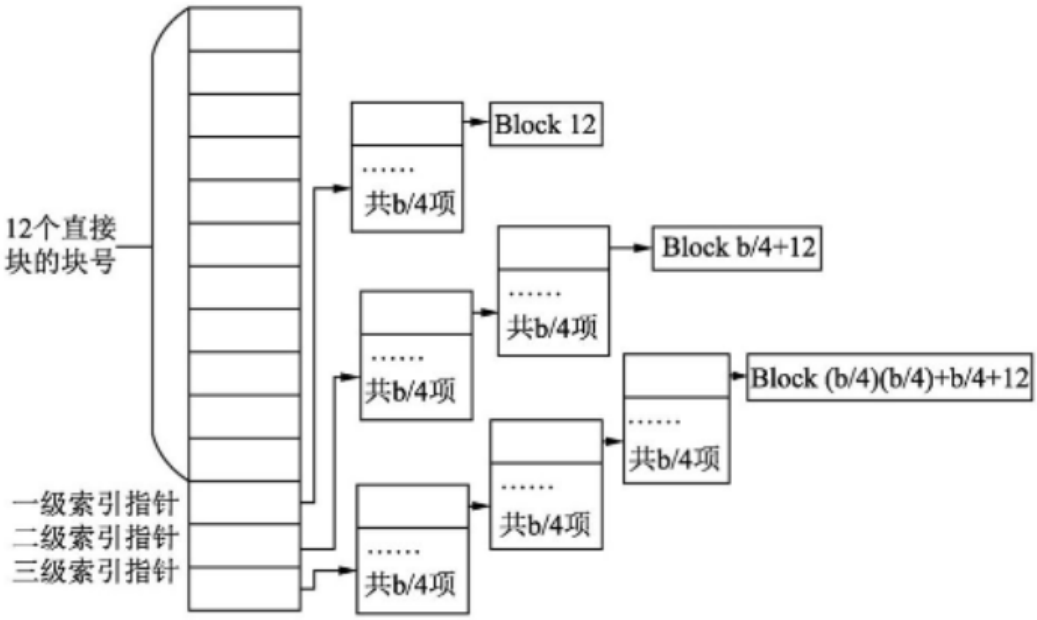


图2-7 EXT3文件数据块的组织示意图

如果一个文件有多个数据块，这些数据块很可能不是连续存放的，应该如何寻址到每个块呢？实际上，根目录的数据块是通过其inode中的索引项Blocks[0]找到的，事实上这样的索引项一共有15个，从Blocks[0]到Blocks[14]，每个索引项占4字节。前12个索引项都表示块编号，如Blocks[0]字段保存着24，就表示第24个块是该文件的数据块，如果块大小是1KB，这样可以表示从0～12KB的文件。

索引项Blocks[12]所指向的块并非数据块，而是称为间接寻址块（IndirectBlock），其中存放的都是类似Blocks[0]这种索引项，再由索引项指向数据块，设块大小为b，那么一个间接寻址块中可以存放b/4个索引项，指向b/4个数据块。所以如果把Blocks[0]到Blocks[12]都用上，最多可以表示b/4+12个数据块，对于块大小是1KB的情况，最大可表示268KB的文件。二级索引指针可以管理的文件大小为64MB，三级索引指针可以管理的文件大小为16GB。

journal(日志系统):

对文件的存储包括对文件数据（data）和文件属性（元数据，metadata）的存储。为了在意外情况（如系统崩溃）下，可以通过恢复程序保证数据的一致性和完整性，EXT3文件系统引入了日志的概念，EXT4还能对日志进行校验和，确保有效的数据变更能够在底层文件系统上正确完成。EXT3/4日志存放在journal文件中，并提供了三种日志的记录方式：

1）data=writeback方式：在这种方式下，EXT3文件系统只对元数据写日志。虽然这会让最近修改的文件在出现意外事件时损坏，但可以得到较高的速度。

2）data=ordered方式（默认方式）：在这种方式下，EXT3文件系统也只将元数据写入日志系统，但还把对数据的每一次更新都作为一个事务写入相关文件中，这样有效地解决了第一种方式中文件数据被损坏的问题。但是文件系统的运行速度要稍慢些，且不能解决文件数据被覆盖时系统崩溃而无法恢复的问题。

3）data=journal方式：在这种方式下，EXT3文件系统提供了完整的文件数据和元数据的日志记录，再写入它的最终位置。这样在系统崩溃时，就可以通过日志来完全修复，保证文件数据和元数据的一致性。

读取文件的过程:

当执行"cat /var/log/messages"命令在系统内部进行了什么样的步骤呢?

找到GDT-->找到"/"的inode-->找到/的数据块读取var的inode-->找到var的数据块读取log的inode-->找到log的数据块读取messages的inode-->找到messages的数据块并读取它们。

软链接和硬链接在inode上的区别是什么

软链接:

软链接的block指针存储的是目标文件名。软链接在功能上等价与Windows系统中的快捷方式，它指向原文件，原文件损坏或消失，软链接文件就损坏。可以认为软链接inode记录中的指针内容是目标路径的字符串。

软链接之所以也被称为特殊文件的原因是：它一般情况下不占用data block，仅仅通过它对应的inode记录就能将其信息描述完成；符号链接的大小是其指向目标路径占用的字符个数，例如某个符号链接的指向方式为"rmt --> ../sbin/rmt"，则其文件大小为11字节；只有当符号链接指向的目标的路径名较长(60个字节)时文件系统才会划分一个data block给它；它的权限如何也不重要，因它只是一个指向原文件的"工具"，最终决定是否能读写执行的权限由原文件决定，所以很可能ls -l查看到的符号链接权限为777。

硬链接:

inode相同的文件在Linux中被称为"硬链接"。

即这些文件所在目录的data block中的inode号都是一样的，只不过各inode号对应的文件名互不相同而已。

每个文件都有一个"硬链接数"的属性，使用ls -l的第二列就是被硬链接数，它表示的就是该文件有几个硬链接。

每创建一个文件的硬链接，实质上是多一个指向该inode记录的inode指针，并且硬链接数加1。

删除文件的实质是删除该文件所在目录data block中的对应的inode行，所以也是减少硬链接次数，由于block指针是存储在inode中的，所以不是真的删除数据，如果仍有其他inode号链接到该inode，那么该文件的block指针仍然是可用的。当硬链接次数为1时再删除文件就是真的删除文件了，此时inode记录中block指针也将被删除。

硬链接只能对文件创建，无法对目录创建硬链接。之所以无法对目录创建硬链接，是因为文件系统已经把每个目录的硬链接创建好了。一个包含子目录的目录文件，其硬链接数是2+子目录数。为什么文件系统自己创建好了目录的硬链接就不允许人为创建呢？从"."和".."的用法上考虑，如果当前目录为/usr，我们可以使用"./local"来表示/usr/local，但是如果我们人为创建了/usr目录的硬链接/tmp/husr，难道我们也要使用"/tmp/husr/local"来表示/usr/local吗？这其实已经是软链接的作用了。若要将其认为是硬链接的功能，这必将导致硬链接维护的混乱。

容错(4%)

为什么要写这么多superblock，为什么不只写一个就好了?

因为superblock文件系统而言是至关重要的，superblock丢失或损坏必将导致文件系统的损坏。所以需要进行备份，使得当一个superblock坏了之后可以通过备份来恢复，由此大大降低损坏的可能性。

类FAT文件系统的分析(6%)

给定的类FAT文件系统：最前面有一个FAT，后面直接跟着一个个文件

一共两个问题

Q1：这种文件系统有什么缺点?

不支持可变长度读写，它每一个文件大小都必须在一开始定死，即不能使用超过一开始确定的大小(超出的空间是下一个文件存储的空间)

Q2：如果我想要在这种文件系统中支持可变长度读写，我应该怎么做?

此时应该引入page的概念了，后面不应该是一块连续的存储空间，而应该是一个个页。

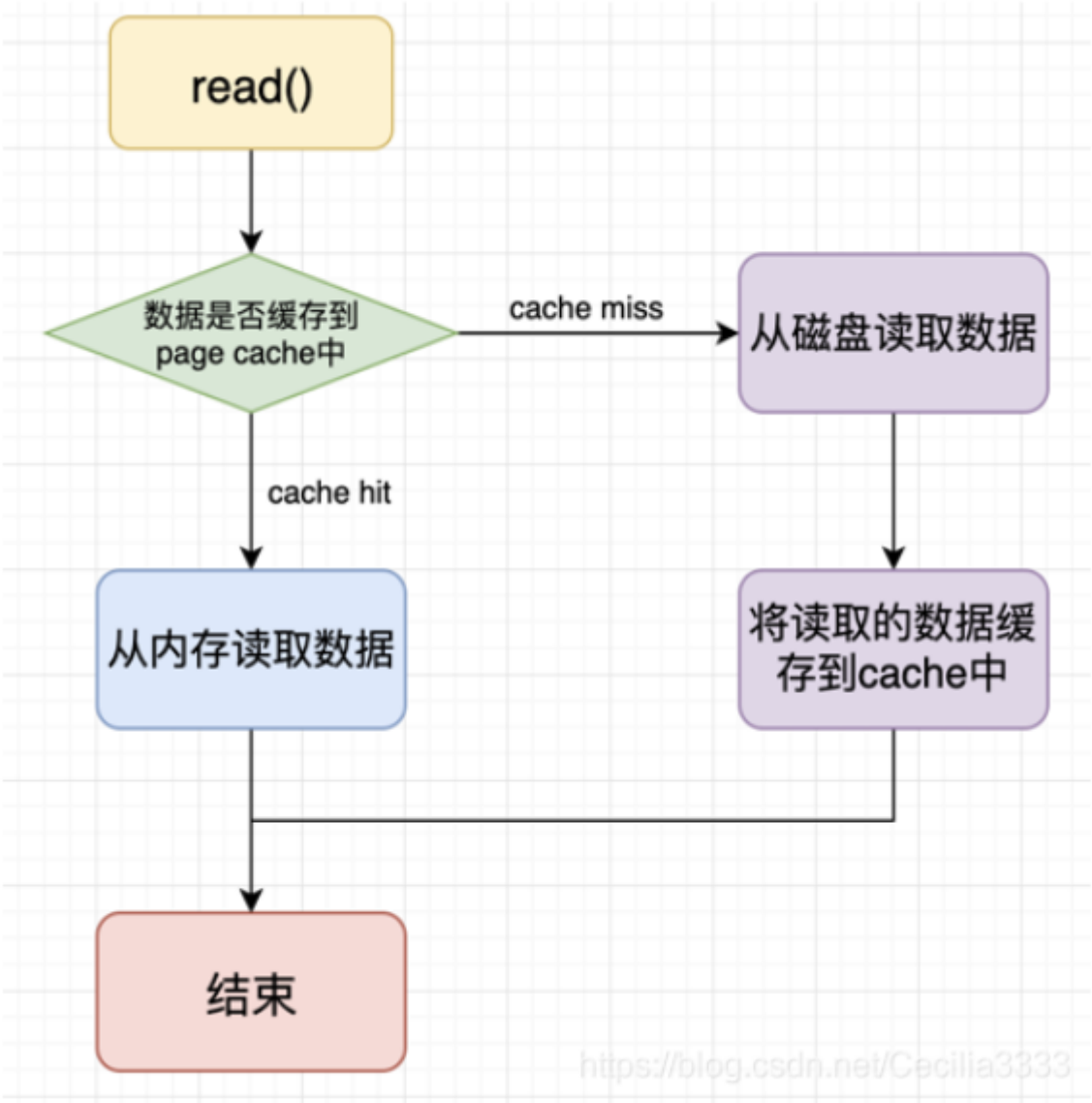
每个文件分配的是很多页，它们不一定连续，它们之间由链表连接起来，而链表信息抽出来放到一开始的file allocation table中。

Page Cache设计(5%)

(只考read，不考write)

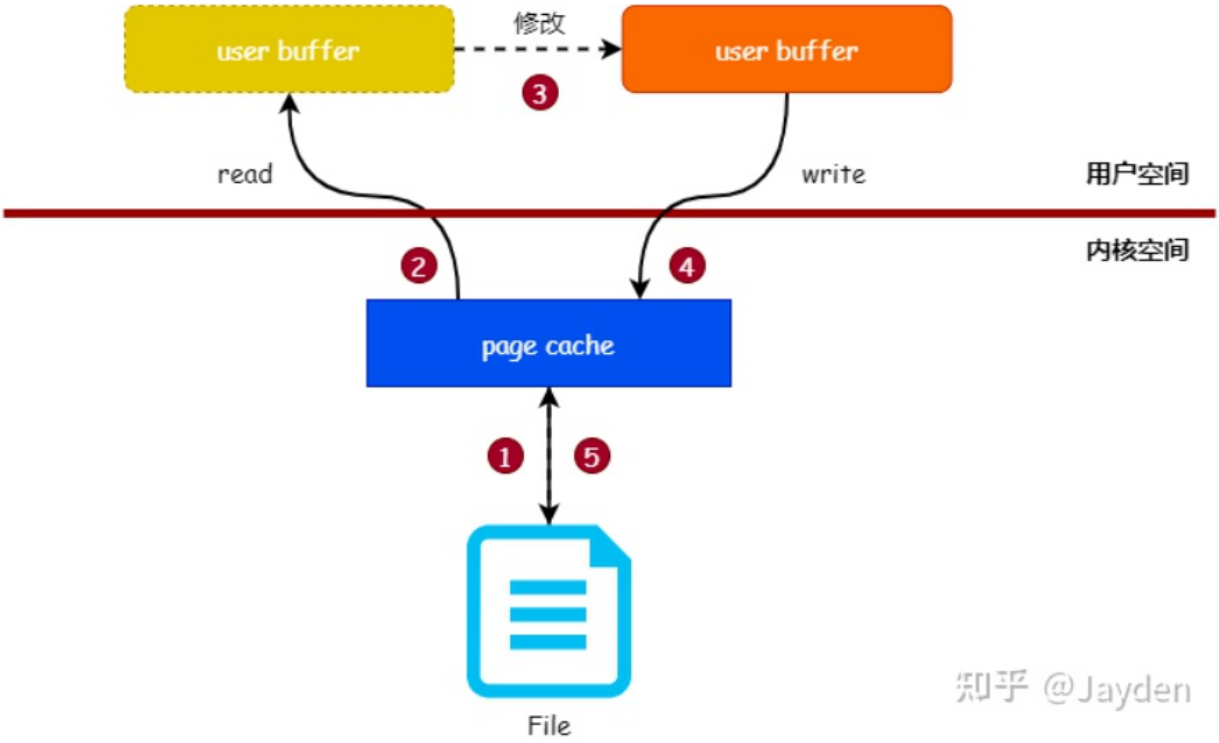
page cache，页高速缓冲存储器。page cache用于缓存文件的页数据，从磁盘中读取到的内容是存储在page cache里的。它的目的是用于加速文件系统的读取操作。

当内核发起一个读请求时（例如进程发起read()请求），首先会检查请求的数据是否缓存到了page cache中。如果有，那么直接从内存中读取，不需要访问磁盘，这被称为cache命中（cache hit）。如果cache中没有请求的数据，即cache未命中（cache miss），就必须从磁盘中读取数据。然后内核将读取的数据缓存到cache中，这样后续的读请求就可以命中cache了。



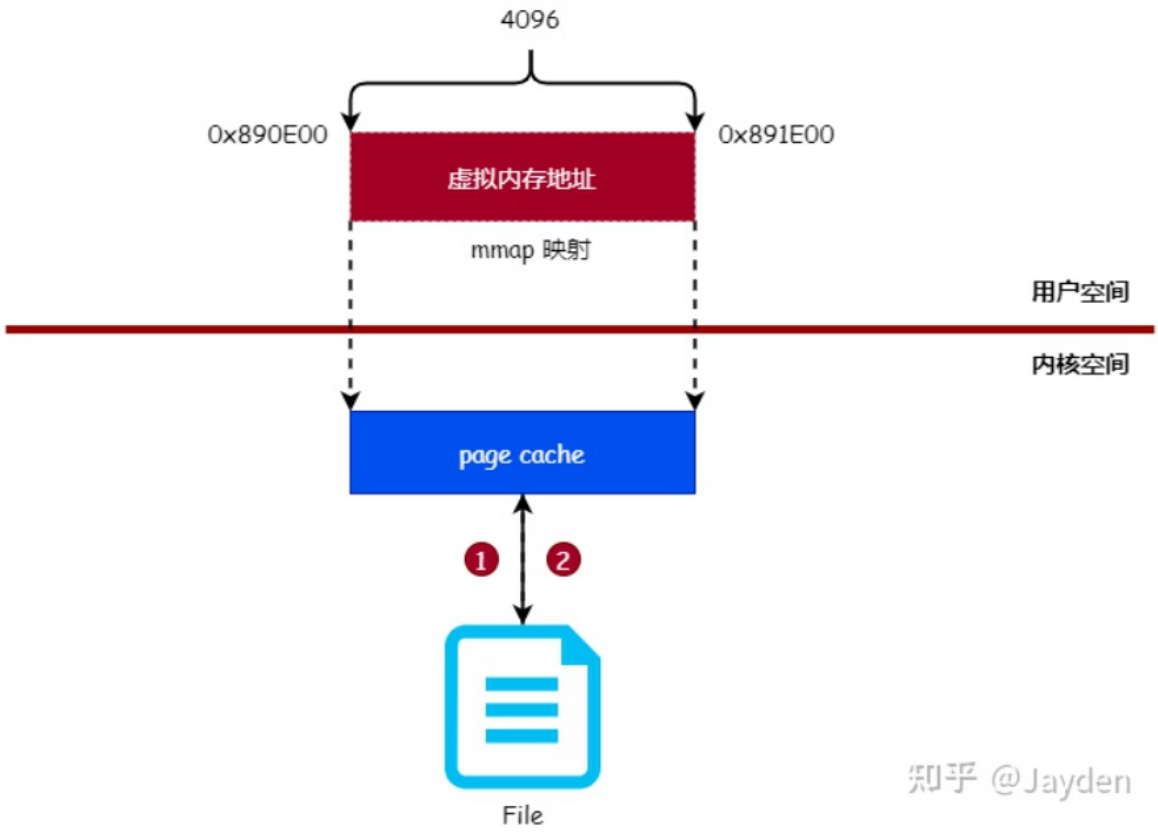
一般来说(传统的读写文件)，修改一个文件的内容需要如下3个步骤：

- 把文件内容读入到内存中。
- 修改内存中的内容。
- 把内存的数据写入到文件中。



从传统读写文件的过程中，我们可以发现有个地方可以优化：如果可以直接在用户空间读写 `页缓存`，那么就可以免去将 `页缓存` 的数据复制到用户空间缓冲区的过程。

由此引入了mmap



使用mmap进行映射时，并没有直接把对应存储的文件内容读入内存中，而是让虚拟内存地址和page cache产生了一个映射关系。

在每次read时候，会查询对应page cache中是否存在相应的数据，如果没有则引发缺页异常，会把对应的页(甚至包括相邻的页)加载到page cache中，再映射到内存中。

5.虚拟化与安全(12%)

每个进程都有独立的页表，CR3寄存器存储的是页表基地址，当需要切进程的时候只需要修改CR3即可完成页表切换，且CR3修改之后会flush TLB(快表)

Hypervisor，也称为虚拟机监视器或VMM，Virtual Machine Monitor。

Hypervisor充当虚拟化层，它在物理服务器或主机操作系统之上运行，并创建和管理多个虚拟机实例。它提供了一个虚拟化平台，使得多个虚拟机可以在同一台物理服务器上共享硬件资源，如处理器、内存、磁盘和网络。

KVM，即Kernel-based Virtual Machine，是一种基于Linux内核的虚拟化解决方案。

watchdog（看门狗）是一种硬件或软件机制，用于监控系统的运行状态并在发生故障或异常情况时采取相应的措施。

硬件看门狗（Hardware Watchdog）是一种专门的计时器设备，通常由计算机的主板或外部模块提供。它定期向计算机发送一个信号，以确保系统正常运行。如果系统在预定的时间内未能响应该信号，硬件看门狗会认为系统发生故障或死锁，并触发一个复位信号来重新启动系统或采取其他预定义的恢复措施。

软件看门狗（Software Watchdog）是一种由操作系统或应用程序实现的看门狗机制。它基于计时器或定时任务，在系统正常运行时定期进行重置或更新。如果系统出现故障、死锁、崩溃或其他异常情况，导致软件看门狗无法定期重置，它会认为系统出现问题，并触发相应的恢复动作，如重新启动系统或执行预定的故障处理程序。

EPT(Extended Page Table)，nestedPT(Nested Page Table)，shadow page table

在虚拟化环境中，由于VMM（Virtual Machine Monitor）可以同时管理、运行多个虚拟机，每个虚拟机看到的硬件资源都由VMM模拟或者分配，所以虚拟机看到的硬件资源可能不是真实的，比如内存，所以在虚拟化环境中就需要将虚拟机所认为的物理内存转换成实际的物理内存。即在虚拟机内部的客户机操作系统，它能感受到的是和普通的操作系统一样，即将虚拟地址转换为物理地址（客户机物理地址），而在实际的实现中，还需要在客户机操作系统无法感知的地方将客户机物理地址转换为真实的物理地址，即宿主机物理地址，这个过程即GVA（Guest Virtual Address）-> GPA（Guest Physical Address） -> HPA（Host Physical Address）。

具体流程是，为了让一个页表实现两层的地址转换（GVA->GPA是虚拟机操作系统可以见的，GPA->HPA是虚拟机操作系统不可见的），VMM将采用陷入再模拟的方式，让虚拟机操作系统对CR3和页表的访问都触发退出到VMM，然后VMM将两次地址转换的页表结合起来。

具体有两种方法对上述流程进行加速，可以通过纯软件的方式，也可以通过硬件辅助的方式。

纯软件的方法即模拟上述操作，然后将两次地址转换的页表结合起来形成一系列的页表，即影子页表(shadow page table)，

为了实现影子页表，大部分的CR3寄存器和页表访问都需要陷入到VMM中，让VMM在软件层面上进行模拟。同时VMM需要为每个虚拟机操作系统的每个进程维护一张地址转换页表，会占用大量的内存资源。为此带来了两个问题，一个是地址转换的效率不高，另一方面则是内存资源占用过多。

另一种方法就是直接使用上述两次翻译的流程，但是为了克服在虚拟化环境中地址转换的性能和资源问题，Intel x86 CPU的VMX功能集中引入了EPT（Extended Page Table）机制，即在原本基于CR3和地址转换页表实现分页机制的基础上，再在硬件上引入一层分页机制，该新的分页机制就叫做EPT。(NestedPT和EPT类似，它是AMD-V技术的一部分，在支持AMD-V的处理器上可用)

EPT分页机制用于将GPA转化为HPA，而这一过程全部由硬件自动完成

硬件的地址转换速度就比软件快多了，解决了性能的问题。

另外，由于EPT的作用是进行GPA->HPA的转换，每个虚拟机操作系统只有一份GPA，所以VMM只需要为每个虚拟机维护一份GPA->HPA的地址转换页表，这样就可以大大减少地址转换页表对内存的占用。

不用EPT，直接走GVA -> GPA -> HPA，两边都是4级页表，则一次地址翻译需要查询多少次Host OS的页表查询(GPA->HPA)?

第一步 GVA -> GPA 是由Guest OS(VMM)来做的，第二步 GPA -> HPA 时由Host OS来做的。

一共需要4*4+4=20次Host Os的页表查询

4次指的是查询Guest OS上的CR3的HPA时需要4次Host OS的页表查询

(有时候CR3的页表查询可能可以提前做掉，这种情况下只需要查4*4=16次Host Os的页表查询)

EPT有什么用

EPT通过引入额外的页表级别，扩展了虚拟机的页表结构，使得虚拟机可以直接映射和访问物理内存，而无需进行额外的软件转换。

它大大提高了内存访问的速度和效率，且相对于影子页表大大减少了内存资源占用(原来使用影子页表时VMM需要对每一个虚拟机中的线程都维护一个 GVA -> HPA 的影子页表，而使用EPT之后VMM只需要对于一个虚拟机维护 GPA -> HPA 的页表)

如果一个病毒，把所有中断包括时钟中断都关掉了，那这个核就死掉了。如果我们有一个特殊的core，它永远不会死掉，它可以向其它核发NMI，请设计一种方案来解救死掉的那个核

NMI(Non-Maskable Interrupt)是一种特殊类型的中断，它在计算机系统中具有最高的优先级，无法被屏蔽或忽略。NMI通常用于处理紧急和关键的系统事件，如硬件故障、系统崩溃、内存校验错误等。

如果一台机器上，一个进程死掉了，会不会影响其它进程的运行？ 不会。

如果病毒是在虚拟机中，hypervisor可以直接把虚拟中断重新打开。

如果病毒是在一个真实的CPU上，我们难以直接对硬件进行更改，此时需要使用NMI。

具体为，当发现某一个核长时间没有任何硬件中断时，可以使用NMI发核间中断。NMI通常由硬件设备或电路生成，并通过专用的NMI线路或引脚连接到处理器。

VMX(Virtual Machine Extension)是指Intel处理器的虚拟化扩展技术，也被称为硬件辅助虚拟化技术。它提供了一组处理器指令和硬件功能，用于增强虚拟化的性能和效率。

VMX技术的主要目标是提供硬件级别的虚拟化支持，以减少虚拟化的开销和性能损失。它引入了两种特权级别，分别为Root模式和Non-root模式。Root模式用于虚拟机监视器，它具有更高的特权级别，可以直接控制和管理物理资源。而Non-root模式用于虚拟机，它处于更低的特权级别，并且受到虚拟机监视器的管理和限制。

已知一种病毒能更改CR3，那如果病毒在虚拟机中，在有nestedPT的时候，它能不能发挥作用

不能，因为它修改的CR3是Guest OS中的CR3，它无法改到Host OS中的CR3

在虚拟机中，病毒对关键寄存器进行修改，如CR3，并不会直接修改宿主机的对应寄存器，简要说明虚拟机如何达成这一权限控制?

在使用nestedPT且不使用EPT进行硬件加速的时候，具体是由于VMM会对Guest OS中的所有GPA再做一次翻译到HPA，所以病毒对CR3进行修改时，它只能修改Guest OS中的CR3，它无法改到Host OS中的CR3。

如果使用EPT进行硬件加速，我们在Host OS的kernel中使用的是VMX Root Mode，而在Guest OS的kernel中使用的是VMX non-Root Mode，由此对权限进行了控制。

6.fork分析(10%)

fork创建好以后，除了pid是不是一模一样？是的

fork and printf

```
printf("a");
int x=fork();
if(x==0){//child
   printf("b");
}
else {//parent
   printf("b");
}
```

两个进程都输出 `ab`

因为printf没有刷新缓冲区，而fork会把缓冲区也复制一份。

fork and malloc

先malloc，再fork，问fork的那一段空间在物理地址上是否共享?

fork的时候页表拷贝了一份。

使用copy on write。如果没有write操作，它们的物理地址一开始是共享的，只有当write的时候，page fault，就copy一份再修改。

7.综合设计(10%)

- 虚拟化的加速
- NUMA场景下的同步原语
- 异构计算和可信执行环境

NUMA场景下的同步原语

现代操作系统P301，P302

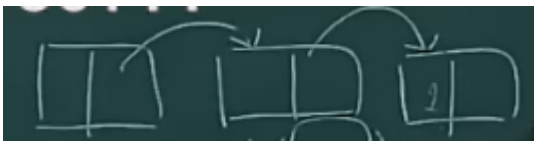
- UMA（Uniform Memory Access，统一存储器访问）：UMA是一种内存访问模式，指的是在计算机系统中，所有的处理器（或核心）对内存的访问具有相同的延迟和带宽。这意味着无论处理器位于系统中的哪个位置，访问内存的速度和性能都是相同的。在UMA系统中，所有的处理器共享同一个物理内存，并且可以通过共享总线或交叉连接网络进行通信。
- NUMA（Non-Uniform Memory Access，非一致存储器访问）：NUMA是一种内存访问模式，指的是在计算机系统中，不同的处理器（或核心）对内存的访问具有不同的延迟和带宽。这是由于系统中存在多个内存节点（memory node），每个节点包含一部分物理内存和与之关联的处理器。每个处理器可以更快地访问与其关联的本地内存节点，但访问其他处理器关联的远程内存节点时会有较高的延迟。NUMA系统通过在硬件和操作系统级别管理内存访问，以实现更好的性能和可扩展性。

同步原语是一种用于协调多个并发执行的线程或进程之间操作顺序和访问共享资源的机制。它提供了一组原子性操作，用于确保多个线程或进程在访问共享资源时的正确性和一致性。

常见的同步原语包括 互斥锁（Mutex），信号量（Semaphore）条件变量（Condition Variable），屏障（Barrier）

现在有这样一个问题，在具有超多CPU的NUMA多处理机中，当多个CPU都对同一个锁进行争抢，此时会不断撞cache line，发生cache冲突，反而导致处理器增多，反而性能下降，要如何设计使得性能增加?

可以让每个打算获得互斥信号量的CPU都拥有各自用于测试的私有锁变量，并且它们的cache尽量不会冲突。对于每一个未能获得锁的CPU分配一个锁变量并且把它附在等待该锁的CPU链表的末端，让链表前一个CPU占用它的私有锁，并且让它自身不断在该私有锁上自旋。每当锁的持有者退出临界区时，它释放该锁，并释放链表下一个CPU的私有锁，使得下一个CPU能进入尝试申请锁并进入临界区。如此不断做下去。



优势:

- 本来多个CPU对同一个锁进行争抢，cache冲突，每个CPU都要不断刷cache，造成大量性能损失。而使用链表的方式使得每个CPU都在不同的锁上自旋，它们不太会cache冲突
- 本来NUMA会导致不同CPU访问数据所需要的时间是不同的，可能同时争抢一个锁的时候，距离远访问慢的CPU一直抢不到锁，而使用链表之后访问锁的顺序只跟谁先申请锁有关，由此可以实现一定程度的公平