

cs-medium-03

step1

小明的问题：

- 16G电脑内存指的是RAM，是易失性存储器，用于临时存储正在运行的程序和数据，供cpu直接高速读取。手机中的内存一般指存储，用运行内存指代RAM。
- 硬盘756G，硬盘有多种（如HDD，SSD），是非易失性存储器，用于长期存放文件。而硬盘不仅影响了电脑总存储容量，硬盘的存储速度直接影响开机，软件加载和文件传输的快慢。而硬盘的常规容量是480G/512G或960G/1T，可能将一部分容量模拟为高速缓存或由不同容量固态硬盘组合。而756G并不算大，操作系统约200G，为实现小双的要求，一个3A游戏动辄上百G，因此756G并不算多。

问题1：

CPU是中央处理器，用来处理各种计算任务。讲解cpu需要引入核心的概念。核心是cpu内部的一个独立的，完整的计算单元。核心具有自己的算数逻辑单元，寄存器（极小但极快的内存，用于存放临时计算数据），控制单元（负责指挥和协调工作），一级缓存（核心专有的高速内存）。一个cpu中含有多个核心，多个核心可以同时并行处理多个任务，并且每个物理核心还可以模拟出多个逻辑核心。因此，cpu的性能主要有核心数量与核心质量两个指标构成。

而大型游戏对于单核性能非常敏感，因为它们有很多计算无法拆分，则为了满足小双游玩3A游戏的需求，cpu要更关注单核质量而不能只看核心数量。而“i几”指核心数量，i3一般为少量，i5多余i3，i7，i9拥有更多核心，这里10核心指说明核心多，但不能说明单核质量好

GPU是显卡，即图像处理器，专门用于处理图像和视频相关的并行计算。主要负责将电脑的数字信号转化为图像输出到显示器，对于游戏，视频渲染等至关重要，而ATi 7670是一张老旧的，低性能的显卡，无法带动3A游戏的画面。

主板是一块连接所有其他硬件的PCB电路板，为CPU，GPU，内存，硬盘等提供插槽和接口，负责各部件之间的数据传输和电力分配，继承了声卡，网卡等基础功能。

问题2：

对于内存/主存（RAM），电脑直接叫内存，手机也叫“内存”或“运行内存”

存储，电脑叫硬盘，或直接叫存储。而手机叫“存储空间”，“内存”，“闪存”

问题3：

对于台式电脑，不同于笔记本电脑的电源由“电池”和“充电器”两部分组成，台式电脑的电源只是一个“充电器”，有电电脑可以运行，充电器只传电，不存电，断电则电脑不能运行。电源的使用有以下需要注意的事项：

- 注意额定功率，估算电脑耗电的大致功率，不能超过额定功率，否则会导致输出不稳，波纹过大，可能会烧毁硬件
- 电源过热，短路等问题可能会引发火灾
- 电源如果供电不稳定，还会导致电脑频繁重启，系统不稳定，蓝屏等诸多问题。
- 要注意电源散热
- 确保电源有一个完善的保护电路

问题4：

CPU的性能要看具体型号,不能只看核心数量，从架构，核心/线程数，单核性能等多个方面选择/评估一个CPU。

CPU架构包括宏观架构（核心布局，缓存系统L1,L2,L3的大小，共享策略等以及互联总线（核心间，核心与外部如何通信））。和微观架构，主要包括指令集（CPU语言，有x86（电脑）和ARM（手机）），流水线（指令处理流程），分析预测，乱序执行，执行单元等等。

GPU的性能同样要看具体型号，可以直接查看显卡天梯图，比较显卡性能，再根据预算选择合适的显卡。

问题5：

- 整体平台老旧，E5 CPU + X79主板是一个早已淘汰的平台，所有部件都是二手或翻新的，没有保修，故障率高，没有升级空间
- 品牌模糊，内存，硬盘没有标出具体的型号，可能使用劣质颗粒的白牌或拆机货，稳定性差，容易蓝屏
- 散热器差，全铝双铜管散热器会导致CPU发热
- 整体性能差，不可能畅玩3A游戏

- 价格高，整体价格应在1000元以下

8 + 16G：

8GB是实际的运行内存，而16G是虚拟内存，从手机存储中画出16G来模拟成运行内存，当系统内存不够用时，系统会把后台不常用的app交换到存储空间划出的这个区域，为前台腾出实际内存。

而虚拟内存的与真实内存的速度差异极大，还会消耗存储寿命，因此8+16实际只是一个营销噱头。

“为什么不能呢”：

游戏内存是易失性的，在断电情况下来不及不做任何记录，断电后电脑直接关闭，内存的内容无法保存在存储中，来不及保存的进度是实时的，只保存在内存中，因此会失去。

CPU调动显存，即某一个游戏画面，内存为上一个保存到现在产生的全部游戏数据，存储中有你的游戏账号和这个游戏本身。如果要做到断电后游戏仍能存在，则需要一直对存储进行操作，这是不现实的。显存的计算速度远高于存储，如果使用存储实时保存游戏进度，则会导致游戏卡顿，无法运行。

在断电后，内存的数据大部分失去，只剩下小部分残留数据，无法回复成原先的结果，如果强行操作，会导致数据错误等诸多问题。

寄存器：

ESP为“扩展栈指针”，是CPU内部的特殊存储单元。用来存储一个指向栈顶部的内存地址。

每一个进程中，都会分配一个栈用来存放局部变量，传递函数参数，存放返回地址。而通过移动ESP指针的位置，可以直接改变栈顶，快速分配或释放栈的空间，效率极高。

而CPU通过ESP跟踪栈顶的位置，实现函数调用和返回。而如果ESP位置异常，CPU就会向一个错误的地址访问栈，触发严重的异常。而蓝屏的一个可能原因是：“一个坏的驱动程序或硬件故障意外覆盖ESP，导致操作系统内核陷入无法恢复的陷阱。通过蓝屏重启来保护系统。”

接下来学习什么是寄存器：寄存器是CPU内部数量稀少，速度极快的存储单元，用于临时存放正在被CPU直接处理的指令，数据和地址，速度极快，快于缓存。并且寄存器容量极小，通常只能存放一个数据（一个指针）。

常见寄存器包括：

- 数据寄存器：存放操作数与结果

- 地址寄存器：存放内存地址
- 控制寄存器：控制CPU工作模式
- 段寄存器：内存分段管理
- 特殊功能寄存器：指令指针，指向下一个指令地址 栈指针（ESP） 标志寄存器，进位/零值/溢出等状

下面的五个问题：

问题1：

寄存器存在于CPU内部。

高速缓存是位于CPU和主内存之间的快速存储器，缓存的L1和L2通常在每个核心的内部，L3缓存被所有核心共享，存在于CPU内部或紧邻CPU的封装中

只读存储器：是一种非易失性存储器，数据只能读取不能随意写入，用于存储固件和启动代码。存在于主板上，是一个独立芯片。

问题2：

- SARM：静态随机存取存储器，高速缓存，使用触发器存储数据，不需要刷新，访问速度快。主要用于缓存，以及需要高速访问的嵌入式系统。原因：由于每个存储单元需要6个晶体管，结构复杂，所以面积大，成本高，但无需刷新，速度快
- NAND：非易失性存储器，可读写，但写入慢，擦除次数有限，容量大，成本低，用于硬盘存储。原因：基于电荷trapping原理，数据在断电后不会丢失，但写入和擦除需要高电压，导致延迟和磨损。
- HDD：硬盘驱动器，使用磁性盘片和机械臂读写数据，速度慢，用于大容量数据存储，如台式机硬盘，服务器存储和备份。原因：机械结构导致寻道时间和旋转延迟，因此访问速度远低于电子存储器。
- DRAM：动态随机存取存储器，虚拟缓存，比高速缓存慢10倍左右，使用电容存储数据，需要定期刷新防止数据丢失，访问速度快。用于计算机内存。原因：每个存储单元只需一个晶体管和一个电容，结构简单，密度高，但电容漏电需要刷新。

问题3：

物理结构：

- 存储单元：每个单元由一个晶体管和一个电容组成，电容存储电荷（代表二进制0和1），晶体管作为开关控制访问（一个单元只能存储一个0或1）
- 阵列结构：多个单元组成一个存储bank，多个bank组成一个内存芯片，内存条由多个芯片封装在PCB上
- 外围电路：包括行解码器，列解码器，感应放大器和刷新电路

工作原理：

读取：

1. 预充电：位线被预充电道参考电压
2. 行选择：行地址阶码后，激活字线，打开该行所有晶体管
3. 电荷共享：电容上的电荷与位线共享，导致位线电压微小变化
4. 感应放大：感应放大器检测电压变化，并放大为逻辑电平（0或1）
5. 列选择：列地址阶码后，通过数据线输出数据
6. 刷新：读取后，电容电荷被破坏，所有数据写回

写入：

1. 行和列选择目标单元
2. 数据线提供新数据，通过写入电路强制位线电压改变
3. 激活字线，晶体管导通，新电荷写入电容

问题4：

9950HX X3d的缓存更大，访问延迟可能增加，因为寻址时间变长。而9800 X3d的缓存设计可能更优化，延迟更低

此外，还有热设计效率，缓存增加会导致功耗和发热增加，降低性能；另外，软件优化等原因也会导致延迟增高。

缓存不是越多越好，主要是因为L1，L2的效率明显高于L3,9950HX X3d的缓存更大主要是实现了L3的堆叠。缓存增多，延长了缓存的访问路径，拖慢了L1和L2的速度；此外，缓存增加会导致功耗增高，发热，同时会增加芯片成本。

此外，存在局部性原理：如果程序没有良好的局部性，大缓存可能无法提高命中率，反而增加管理成本。

还有一致性原理：多核环境下，缓存一致性协议更复杂，可能导致竞争和延迟。

问题5：

什么是cache? cache就是缓存。

最根本原理：局部性原理，表明：程序在执行时对数据的访问呈现出明显的规律性，而非完全随机。这是cache存在的基础。

- 时间局部性：最近访问的数据很可能再次被访问。例子：循环（简单编程）线上购物时购物车这个功能的调用（函数调用）
- 空间局部性：如果一个数据被访问，那么它地址附近的数据项很有可能在不久后被访问。例子：数组，加载图片（加载一块图片时，不仅需要加载当前的数据块，还需要其后续的所有数据块。）

由于程序的行为具有局部性的规律，计算机对接下来的行为进行预测和提前准备，实现缓存。

缓存实现的方法：用空间换时间

主动牺牲一部分昂贵的高速存储空间，用来存放那些根据“局部性原理”预测可能会被用到的数据副本。当真正需要这些数据时，我们就可以直接从高速缓存种获取，从而节省了慢速主存储中查找所需的时间。

这些事cache存在的根本原理。

接下来用一个代码实现矩阵乘法：这个代码比较简单，重点是从其中体悟cache的实现，再询问ai后，我了解到，这个函数的矩阵要相对较大，而不能是我在往常学习时见到的很小的数据

为了实现缓存的应用，我在这里学习了一种新的循环方法：引入block块，将矩阵一次处理一个模块，简化计算。

```
#include <stdio.h>
#include <stdlib.h>

#define N 128 // 矩阵大小
#define BLOCK_SIZE 32 // 块大小，通常选择使块能放入L1缓存：这里int占四字节，block的大小约为4kb
```

//整体的大小适合放入缓存中

```
void matrix_multiply(int A[N][N], int B[N][N], int C[N][N]) {  
    //外层块循环  
    for (int i = 0; i < N; i += BLOCK_SIZE) {  
        //中层循环，获得B的列  
        for (int j = 0; j < N; j += BLOCK_SIZE) {  
            //内层循环，获得A的列和B的行  
            for (int k = 0; k < N; k += BLOCK_SIZE) {  
                // 处理一个块，这里使用逐个循环：  
                for (int ii = i; ii < i + BLOCK_SIZE && ii < N; ii++) {  
                    //从这里开始，ii为定值，A的行不变  
                    for (int jj = j; jj < j + BLOCK_SIZE && jj < N; jj++) {  
                        int sum = 0; //从这里开始，B的列不在改变  
                        for (int kk = k; kk < k + BLOCK_SIZE && kk < N; kk++) {  
                            sum += A[ii][kk] * B[kk][jj];  
                        } //统一A和B共有的K变量，A的列改变，B的行改变，实现矩阵乘法的行x列  
                        C[ii][jj] += sum;  
                    }  
                } //截止到这里  
            }  
        }  
    }  
}  
//这个函数复杂的原因是因为在c语言中一个循环只能定义一个自增量，不能像数学中一样使用累加符号  
//但是这个函数整体的思路很简单，就是实现行x列再累加
```

```
int main()  
{  
    // 初始化矩阵A和B  
    //由于矩阵为128的方阵，一个一个填不现实，因此这里采用了伪随机数来一键填入  
    //这里最初的N为1024，但因为数据太大，溢出栈的范围，报错：“段错误”，因此改为128  
    //注：默认栈大小为8M，若为1024，三个矩阵的大小约为12M，所以不行。  
    int A[N][N], B[N][N], C[N][N];  
    for (int i = 0; i < N; i++) {  
        for (int j = 0; j < N; j++) {  
            A[i][j] = rand() % 100;  
            B[i][j] = rand() % 100;  
            C[i][j] = 0;  
        }  
    }  
}
```

```

    }
}
matrix_multiply(A, B, C);
return 0;
// 由于矩阵太大因此这里不进行打印
}

```

使用分块算法后，在每一块中，利用循环，反复调用这个块的元素，利用了时间局部性

而加块利用到了矩阵元素临近地址的元素，体现了空间局部性

step2

在上一个代码块中我已经出现了一次段错误，在答题前我先学习了段错误出现的原因：

程序试图访问其无权访问的内存区域时发生错误，如访问空指针/数组越界访问/栈溢出等。

问题1：虚拟内存是什么？

虚拟内存是操作系统提供的一种抽象，它让每个进程都认为自己拥有独立的，连续的内存空间，而实际上这些内存可能分散在物理内存的不同位置，甚至部分存储在磁盘上。虚拟地址对应虚拟内存，不适物理内存条上的真实地址，而是操作系统为每个进程创建的假地址空间。

问题2：虚拟内存的组织：无乱是32位或64位，基本结构是相同的

```

+-----+
| 内核空间 | ← 所有进程共享（但用户态不能访问）
+-----+
| 栈       | ← 向低地址增长
|   ↓     |
+-----+
| 内存映射区 | ← 共享库、文件映射等
+-----+
| 堆       | ← 向高地址增长
|   ↑     |
+-----+
| BSS段(.bss) | ← 未初始化全局变量

```



```

+-----+
| 数据段(.data) | ← 已初始化全局变量
+-----+
| 代码段(.text) | ← 程序指令（只读）
+-----+
| 保留区域      |
+-----+

```

虚拟地址空间可以划分为内核空间和用户空间。内核空间在所有进程间共享，但用户态程序无法直接访问内核空间，只有通过系统调用陷入内核态才能访问。内核空间中包含操作系统核心代码和数据结构。

```

+-----+
| 固定映射区   | ← 特殊用途的固定映射
+-----+
| vmalloc区    | ← 非连续内存分配
+-----+
| 内核模块     | ← 动态加载的内核模块
+-----+
| 内核数据     | ← 全局变量、数据结构
+-----+
| 内核代码     | ← 内核核心代码(text)
+-----+
| 直接映射区   | ← 线性映射物理内存
+-----+

```

如图为内核空间的各区域详解。接下来阐明映射的概念：映射指物理内存通过一定变化转化为虚拟内存的过程。

由于这一板块我不了解的知识太多，因此我们一个知识一个知识得看：

关于映射：

- 文件映射：将存储在硬盘中的文件直接映射到虚拟内存中，读取文件时，操作系统将相应部分从磁盘加载到内存。修改文件时，更改首先反映在内存中，稍后同步到磁盘
- 匿名映射，不与任何文件关联的内存映射，用于malloc申请内存空间

接下来解释用户空间的组成：

- 代码段：只读，可执行，在程序加载时从可执行文件中读取，多个进程可共享一个物理页
- 数据段：.data：已初始化的全局变量/静态变量（int a = 10;的a） .bss：未初始化的全局/静态变量（int b; 的b）
- 堆：用来存放malloc申请的内存块，通过brk调用，向高地址增长
- 内存映射区：包含共享库，文件映射和匿名映射。
- 栈：自动管理，存储函数或局部变量，向低地址增长，每个线程都有自己的栈

64位系统中虚拟内存空间的形式：

```

+-----+0xFFFFFFFFFFFFFFFF
| 内核空间 | ← 所有进程共享（但用户态不能访问）
+-----+0xFFFF800000000000
| 栈       | ← 向低地址增长
|   ↓     |
+-----+
| 内存映射区 | ← 共享库、文件映射等
+-----+
| 堆        | ← 向高地址增长
|   ↑     |
+-----+
| BSS段(.bss) | ← 未初始化全局变量
+-----+
| 数据段(.data) | ← 已初始化全局变量
+-----+
| 代码段(.text) | ← 程序指令（只读）
+-----+
| 保留区域     |
+-----+0x0000000000000000

```

接下来介绍页表管理这个概念：

页表是操作系统用来实现虚拟内存到物理内存映射的核心数据结构，记录了每个虚拟页面对应的物理页面位置。

- 页面：虚拟内存和物理内存被划分为固定大小的块，通常为4kb；
- 页框：物理内存中的页面成为页框
- 页表项：页表中的每个项目，包括虚拟页面到物理页框的映射信息。

页表的基本作用：当CPU发出一个虚拟地址时，内存管理单元（MMU）会查询当前进程的页表，完成以下转换：虚拟页号（VPN） + 页内偏移（Offset） → 物理页帧号（PFN） + 页内偏移（Offset）

好的，我们来深入探讨操作系统内存管理的核心——页表管理。我会先阐述其专业原理，然后用一个详细的比喻帮助你彻底理解。

页表由大量页表项组成。每个PTE不仅包含物理页帧号，还包含重要的控制位：

控制位	功能说明
存在位	核心中的核心！表明该页是否已加载到物理内存（1表示在内存，0表示在磁盘交换区）。
读/写位	标记页的权限。例如，代码段通常被标记为只读，以防止程序意外修改指令。
用户/超级用户位	决定用户态程序是否可以访问该页，是实现内核空间保护的关键。
脏位	当程序向该页写入数据后，硬件会自动置位。帮助操作系统在置换该页时，判断是否需要写回磁盘。
访问位	当页被读取或写入时置位。用于页面置换算法（如LRU的近似实现）。

而通过`cat /proc/[PID]/maps`这个linux代码访问的结果

```
00400000-00401000 r-xp 00000000 08:01 123456 /home/xiaoshuang/my_program # 代码段（只读、可执行）
00600000-00601000 r--p 00000000 08:01 123456 /home/xiaoshuang/my_program # 数据段（只读）
00601000-00602000 rw-p 00001000 08:01 123456 /home/xiaoshuang/my_program # 数据段（可读写）
7ffce0000000-7ffce0021000 rw-p 00000000 00:00 0 # 堆（匿名映射，可读写）
7ffce1000000-7ffce1021000 r-xp 00000000 08:01 1234567 /lib/x86_64-linux-gnu/libc.so.6 # C库（共享库映射）
7ffce1220000-7ffce1221000 r--p 00000000 00:00 0 # 栈（匿名映射，可读写）
...
```

如图，显示了控制位于虚拟内存的类型

利用c语言对虚拟内存进行模拟

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdbool.h>

#define PAGE_DIR_BITS    10 // 页目录索引位数
#define PAGE_TABLE_BITS 10 // 页表索引位数
#define OFFSET_BITS     12 // 页内偏移位数

#define PAGE_TABLE_SIZE (1 << PAGE_TABLE_BITS) // 1024
#define PAGE_DIR_SIZE   (1 << PAGE_DIR_BITS)   // 1024
#define PAGE_SIZE       (1 << OFFSET_BITS)     // 4096 (4KB)

// 页表项 (PTE - Page Table Entry)
// 包含物理帧号和权限位
typedef struct {
    bool    present_bit; // 存在位: 1表示该页在物理内存中, 0表示不在
    bool    rw_bit;      // 读写位: 1表示可读可写, 0表示只读
    uint32_t frame_number; // 对应的物理帧号
} PTE;

// 页目录项 (PDE - Page Directory Entry)
// 包含指向页表的指针和存在位
typedef struct {
    bool present_bit; // 存在位: 1表示该页目录项指向一个有效的页表, 0表示无效
    PTE* page_table_base; // 指向一个页表 (PTE数组) 的指针
} PDE;

// 模拟的页目录表 (作为我们模拟的 "CR3寄存器" 指向的地址)
PDE page_directory[PAGE_DIR_SIZE];

/**
 * @brief 模拟MMU进行地址翻译和权限检查
 * * @param virtual_address 要翻译的32位虚拟地址
 * * @param is_write_access 访问类型, true表示写操作, false表示读操作
 */
```

```

*/
void translate_address(uint32_t virtual_address, bool is_write_access) {//请你完成这个
尚未完成的函数

    printf("-----\n");
    printf("Translating Virtual Address: 0x%08X (%s access)\n", virtual_address,
is_write_access ? "WRITE" : "READ");

    // --- 步骤1: 从虚拟地址中提取索引和偏移 ---

    // 提取页目录索引（高10位）：将虚拟地址右移22位（12位偏移+10位页表索引）
    uint32_t page_dir_index = (virtual_address >> (OFFSET_BITS + PAGE_TABLE_BITS)) &
((1 << PAGE_DIR_BITS) - 1);

    // 提取页表索引（中间10位）：将虚拟地址右移12位（偏移位数），然后取低10位
    uint32_t page_table_index = (virtual_address >> OFFSET_BITS) & ((1 <<
PAGE_TABLE_BITS) - 1);

    // 提取页内偏移（低12位）：直接取虚拟地址的低12位
    uint32_t offset = virtual_address & ((1 << OFFSET_BITS) - 1);

    printf(" -> Page Dir Index: %u (0x%X)\n", page_dir_index, page_dir_index);
    printf(" -> Page Table Index: %u (0x%X)\n", page_table_index, page_table_index);
    printf(" -> Offset: %u (0x%X)\n", offset, offset);

    // --- 步骤2: 查询页目录表 ---
    printf(" [*] Checking Page Directory Entry %u...\n", page_dir_index);
    // 获取对应的页目录项(PDE)
    PDE* pde = &page_directory[page_dir_index];

    // 检查页目录项的存在位
    if (!pde->present_bit) {
        // 如果不存在，触发段错误 (Segmentation Fault)
        printf(" [!] FAULT: Page Directory Entry not present. (Segmentation
Fault)\n");
        return;
    }

    printf("      -> PDE is present. Page table base address: %p\n", (void*)pde-
>page_table_base);

```

```

// --- 步骤3: 查询页表 ---
printf("  [*] Checking Page Table Entry %u...\n", page_table_index);
// 获取对应的页表项(PTE)
PTE* pte = &pde->page_table_base[page_table_index];

// 检查页表项的存在位
if (!pte->present_bit) {
    // 如果不存在, 触发缺页错误 (Page Fault)
    printf("  [!] FAULT: Page Table Entry not present. (Page Fault)\n");
    return;
}
printf("      -> PTE is present. Frame number: %u (0x%X)\n", pte->frame_number,
pte->frame_number);

// --- 步骤4: 检查访问权限 ---
printf("  [*] Checking access permissions...\n");
// 检查写权限: 如果是写操作且页面是只读的, 触发保护错误
if (is_write_access && !pte->rw_bit) {
    printf("  [!] FAULT: Write attempt on a read-only page. (Protection
Fault)\n");
    return;
}
printf("      -> Access granted.\n");

// --- 步骤5: 计算最终的物理地址 ---
uint32_t physical_address = (pte->frame_number << OFFSET_BITS) | offset;

printf("  [SUCCESS] Translation complete.\n");
printf("  Virtual Address 0x%08X => Physical Address 0x%08X\n", virtual_address,
physical_address);
}

/**
 * @brief 初始化模拟环境, 预设一些页表和页目录项
 */
void initialize_simulation() {

```

```
printf("Initializing MMU simulation environment...\n");

// 初始化整个页目录表
for (int i = 0; i < PAGE_DIR_SIZE; ++i) {
    page_directory[i].present_bit = false;
    page_directory[i].page_table_base = NULL;
}

// 2. 创建并填充第一个页表 (用于虚拟地址 0x00000000 - 0x003FFFFFF)
//     假设页目录索引为0
PTE* page_table_1 = (PTE*)malloc(sizeof(PTE) * PAGE_TABLE_SIZE);

page_directory[0].present_bit = true;
page_directory[0].page_table_base = page_table_1;

for (int i = 0; i < PAGE_TABLE_SIZE; ++i) {
    page_table_1[i].present_bit = false; // 默认所有PTE无效
}

// 设置几个有效的PTE
// VA 0x00001xxx -> PA 0x0001Axxx (可读可写)
page_table_1[1].present_bit = true;
page_table_1[1].rw_bit = true;
page_table_1[1].frame_number = 26; // 物理帧号 0x1A

// VA 0x00002xxx -> PA 0x0008Fxxx (只读)
page_table_1[2].present_bit = true;
page_table_1[2].rw_bit = false; // 只读页面
page_table_1[2].frame_number = 143; // 物理帧号 0x8F

// 3. 创建并填充第二个页表 (用于虚拟地址 0x00400000 - 0x007FFFFFF)
//     假设页目录索引为1
PTE* page_table_2 = (PTE*)malloc(sizeof(PTE) * PAGE_TABLE_SIZE);
page_directory[1].present_bit = true;
page_directory[1].page_table_base = page_table_2;

for (int i = 0; i < PAGE_TABLE_SIZE; ++i) {
    page_table_2[i].present_bit = false;
}
```

```

// VA 0x00400xxx -> PA 0x00033xxx
page_table_2[0].present_bit = true;
page_table_2[0].rw_bit = true;
page_table_2[0].frame_number = 51; // 物理帧号 0x33

printf("Initialization complete.\n\n");
}

// --- 4. 主函数，运行测试用例 ---

int main() {
    initialize_simulation();

    // --- 测试用例 ---

    // 1. 成功读取：访问一个有效的、可读写的地址
    // 虚拟地址：0x00001A2B
    // -> 页目录索引：0，页表索引：1，偏移：0xA2B
    // -> 查找 PDE[0] -> PTE[1] -> 物理帧号 26 (0x1A)
    // -> 物理地址：(26 << 12) | 0xA2B = 0x1A000 | 0xA2B = 0x1AA2B
    translate_address(0x00001A2B, false);

    // 2. 成功写入：访问一个有效的、可读写的地址
    // 与上面相同，但请求是写操作
    translate_address(0x00001A2B, true);

    // 3. 保护错误：尝试写入一个只读页面
    // 虚拟地址：0x00002048
    // -> 页目录索引：0，页表索引：2，偏移：0x048
    // -> 查找 PDE[0] -> PTE[2] -> rw_bit = 0，触发保护错误
    translate_address(0x00002048, true);

    // 4. 缺页错误：访问一个页表项(PTE)无效的地址
    // 虚拟地址：0x00003555
    // -> 页目录索引：0，页表索引：3，偏移：0x555
    // -> 查找 PDE[0] -> PTE[3] -> present_bit = 0，触发缺页错误
    translate_address(0x00003555, false);
}

```



```

// 5. 段错误：访问一个页目录项(PDE)无效的地址
// 虚拟地址：0x00804000 (页目录索引=2)
// -> 页目录索引：2
// -> 查找 PDE[2] -> present_bit = 0, 触发段错误
translate_address(0x00804000, false);

// --- 释放动态分配的内存 ---
// 在真实OS中，这部分内存管理会更复杂
free(page_directory[0].page_table_base);
free(page_directory[1].page_table_base);

return 0;
}

```

问题1:

主要寄存器分类:

地址转换相关寄存器:

- CR3 (控制寄存器3) : 存储当前进程的页目录基地址, 是虚拟地址转换的起点
- GDTR (全局描述符表寄存器) : 指向GDT的基地址, 用于段式内存管理
- LDTR (局部描述符表寄存器) : 指向当前进程的LDT

系统表指针寄存器:

- IDTR (中断描述符表寄存器) : 指向中断描述符表, 处理异常和中断
- TR (任务寄存器) : 指向当前任务状态段TSS

控制寄存器:

- CR0: 包含PE (保护模式使能)、PG (分页使能) 等关键位
- CR2: 存储引发页错误的线性地址
- CR4: 包含PAE (物理地址扩展)、PSE (页大小扩展) 等

问题2:

核心概念层次结构：

虚拟地址空间 → 页目录(PDE) → 页表(PTE) → 物理页帧

MMU（内存管理单元）

作用：硬件实现的地址翻译器，负责虚拟地址到物理地址的实时转换

工作原理：通过查询页表完成地址映射，同时进行权限检查

页表：

核心数据结构：存储虚拟页号到物理页帧号的映射关系

页表项包含：物理帧号、存在位、读写权限、用户/超级用户位、访问位、脏位

TLB（转换后备缓冲区），作用：缓存最近使用的页表项，避免每次地址转换都访问内存中的页表

工作流程：虚拟地址→查TLB→命中则直接获取物理地址，未命中则查询页表

由于页面要求内存地址连续，一个内存块可能有400M，无法在物理内存中找到一个足够大的连续空间，因此提出多级页表的概念。

多级页表：

解决单级页表空间浪费问题：只为实际使用的地址区域分配页表空间

典型结构：二级页表（PDE→PTE）、四级页表（PGD→PUD→PMD→PTE）

虚拟到物理内存转换过程示例：

虚拟地址 0x08048000 转换过程：

1. 从CR3获取页目录基地址
2. 提取虚拟地址高10位作为页目录索引，查询页目录
3. 从页目录获取页表基地址
4. 提取中间10位作为页表索引，查询页表
5. 从页表获取物理页帧号
6. 组合低12位偏移得到物理地址

问题3:

从内存读取数据:

CPU发出虚拟地址 → MMU查询TLB →

[TLB命中] → 直接获取物理地址 → 访问内存控制器 →

[Cache命中] → 直接从缓存返回数据

[Cache未命中] → 访问DRAM返回数据并更新缓存

[TLB未命中] → 查询多级页表 → 更新TLB → 获取物理地址 → 后续流程相同

从存储设备读取数据（缺页异常）:

CPU发出虚拟地址 → MMU发现PTE存在位为0 → 触发缺页异常 →

操作系统异常处理程序:

1. 检查地址合法性
2. 分配空闲物理页帧
3. 从交换空间或文件系统加载数据
4. 更新页表，设置存在位
5. 重新执行引发异常的指令

问题4: 缺页异常类型:

1.硬缺页（Major Page Fault）：页面从未被加载到内存，需要从磁盘读取数据，开销较大（程序首次访问某个代码段或数据段）

2.软缺页（Minor Page Fault）：页面已在内存但未映射到当前进程，需要建立页表映射即可，无需磁盘I/O（共享库被多个进程使用时的首次映射）

3.无效缺页（Invalid Page Fault）：访问非法地址或权限不足，导致向进程发送SIGSEGV信号终止进程（访问空指针或只读内存的写操作）

问题5:

页面替换的必要性：当物理内存不足时，需要将某些页面换出到磁盘，为新的页面腾出空间

主要替换算法:

最优算法（OPT）：替换未来最长时间不会被访问的页面，理论最优但无法实际实现（需要预知未来）

最近最少使用（LRU）：基于时间局部性原理，替换最久未被访问的页面。

实现：维护访问时间链表或使用访问位近似

时钟算法（Clock）：LRU的近似实现，使用引用位循环检查，可以平衡实现复杂度和替换效果

先进先出（FIFO）：简单但效果较差，可能置换掉常用页面

替换策略考虑因素：优先换出未被修改的页面（避免写回磁盘），考虑页面访问频率和重要性，避免系统抖动（频繁页面置换）

问题6：

中断的本质：中断是处理器响应异步事件的一种机制，使CPU能够暂停当前任务，处理更紧急的事件，然后返回原任务继续执行

中断分类：

硬件中断：

- 外部设备触发（键盘、鼠标、定时器等）
- 可屏蔽中断（INTR）和不可屏蔽中断（NMI）

异常（内部中断）：

- 由CPU执行指令时检测到异常条件引发
- 故障（Faults）：可修复的异常，如缺页异常
- 陷阱（Traps）：有意引发的异常，如系统调用
- 终止（Aborts）：严重错误，通常导致进程终止

软件中断：由程序指令主动触发，如INT指令

中断处理过程：1. 中断发生 → 2. 保存当前上下文 → 3. 查询IDT获取处理程序地址

4. 执行中断处理程序 → 5. 恢复上下文 → 6. 返回原程序继续执行

缺页异常作为一种特殊的中断，使操作系统能够介入内存管理，实现虚拟内存的透明扩展

step3

在此前我没有学习过linux有关的知识，为了实现这个代码的正确运行，我先实现了vscode与wsl的关联，使得vs可以使用针对linux系统的函数。

成功配置好了环境后，我得到了正确的运行结果：

```
--> Hello from exec_demo! My PID is 73343.  
--> I am about to call execlp() to transform into 'ls -l'...  
total 64  
-rwxr-xr-x 1 likaishuo likaishuo 31816 Oct 15 17:37 helloworld  
-rw-r--r-- 1 likaishuo likaishuo   139 Oct 15 17:30 helloworld.cpp  
-rw-r--r-x 1 likaishuo likaishuo  21528 Oct 15 17:41 kfm1
```

```
fflush(stdout); // 刷新标准输出
```

```
//这里这行代码是为了防止execlp将原进程覆盖掉，导致printf无法执行
```

```
//这里由于程序简单，即使注释掉也无影响
```

```
//保证输出完整性：在进程被 exec系列函数替换之前，强制刷新标准输出缓冲区，确保所有提示信息都能被用户看到，避免数据因缓冲区被覆盖而丢失。
```

```
//应对不确定性：虽然换行符 \n通常会触发行缓冲刷新，但明确使用 fflush是一种更严谨、可移植性更好的编程习惯。
```

并对这个不熟悉的函数进行解释：

exec的最重要的特性是进程替换，在一个ID下，将原有进程替换为一个新的进程

而fork-exec模型实现了创建一个新的模型（子进程），再将子进程用exec替换，执行新的程序

还用fork后，会得到一个与父进程除ID外完全相同的子进程，另外，在这个过程中存在写时复制 (Copy-On-Write, COW)：现代操作系统为了效率，通常不会在 fork() 时立即复制父进程的全部内存。父子进程最初共享物理内存页。只有当其中一个进程试图修改某个内存页时，操作系统才会为该进程复制该页。这大大减少了 fork() 的开销。

`fork()` 的主要目的是创建一个新的执行上下文（进程），这个新进程在创建之初和父进程一模一样。它为后续加载新程序（`exec`）提供了基础。

而在新的子进程中再调用 `exec` 族函数，让子进程变成一个新的程序，并且 `exec()` 函数族允许父进程（或子进程自身）通过参数列表（`argv`）和环境变量（`envp`）向新程序传递信息。

这样的结构带了灵活性：

- 在 `fork()` 之后、`exec()` 之前，子进程（因为它拥有父进程的副本）可以执行一些“准备工作”，而这些工作不需要或不应该在父进程中做。例如：关闭不需要继承的文件描述符。修改环境变量。改变信号处理方式。设置用户/组 ID（如果父进程有特权）。重定向标准输入/输出/错误（通过关闭旧描述符并打开新文件，或者使用 `dup2`）。
- 父进程可以在 `fork()` 后继续执行自己的任务，同时子进程去加载并运行新程序。

同时使得每一步工作的功能清晰，将 windows 系统中直接创建一个新的程序的工作分成两步，是 linux 系统的核心。

在写这个 shell 之前，先学习 cow 机制：

如果有多个调用者（如进程、线程）同时请求相同的资源（如内存区域），他们会共同获取相同的指针，指向这份资源。只有当某个调用者试图修改资源的内容时，系统才会真正复制一份专用副本给该调用者，而其他调用者所见到的依然是原来的、未修改的资源。

根据这个原理，子进程不会占用多个父进程等大的存储空间，直接 `exec` 后会直接占用新的内存空间。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/mman.h>

#define MAX_INPUT 1024
#define SHARED_MEM_SIZE 4096

// 演示 Copy-On-Write 的函数
void demonstrate_cow() {
    printf("\n=== 开始演示 Copy-On-Write 机制 ===\n");
```



```

    printf("父进程：子进程修改后，我的变量值 = %d (未被改变!)\n", *shared_var);
    printf("父进程：这就是 Copy-On-Write：写时才复制内存页\n");

    wait(NULL); // 等待子进程结束
    free(shared_var);
}

printf("=== Copy-On-Write 演示结束 ===\n\n");
}

// 执行命令的函数
void execute_command(char *command) {
    pid_t pid = fork();

    if (pid < 0) {
        perror("minishell: fork 失败");
        return;
    } else if (pid == 0) {
        // 子进程：执行命令
        char *args[] = {command, NULL};

        if (execvp(command, args) == -1) { //通过字符串比较，确定子进程的内容是否修改
            printf("minishell: 命令未找到: %s\n", command);
            exit(1);
        }
    } else {
        // 父进程：等待子进程结束
        int status;
        waitpid(pid, &status, 0);

        if (WIFEXITED(status)) {
            printf("minishell: 命令执行完成，退出码: %d\n", WEXITSTATUS(status));
        }
    }
}

// 主函数: minishell 循环
int main() {

```



```
char input[MAX_INPUT];

printf("=== 欢迎使用 MiniShell ===\n");
printf("支持的命令: ls, pwd, date, whoami, echo 等系统命令\n");
printf("特殊命令: cow (演示Copy-On-Write), exit (退出)\n");
printf("=====\n");

// 演示一次 Copy-On-Write
demonstrate_cow();

//执行minishell:
while (1) {
    printf("minishell> ");
    fflush(stdout); //这个函数是刷新缓冲区，避免缓冲区被复制到子进程中，导致重复输出

    // 读取用户输入
    if (fgets(input, MAX_INPUT, stdin) == NULL) { //读取失败
        printf("\n");
        break;
    }

    // 去除换行符
    input[strcspn(input, "\n")] = 0;

    // 处理空输入
    if (strlen(input) == 0) {
        continue;
    }

    // 退出命令
    if (strcmp(input, "exit") == 0) {
        printf("再见! \n");
        break;
    }

    // 特殊命令: 演示 Copy-On-Write
    if (strcmp(input, "cow") == 0) {
        demonstrate_cow();
    }
}
```

```

        continue;
    }

    // 执行普通命令
    execute_command(input);
}

return 0;
}

```

这个函数只是写了一个外壳，让用户可以在终端不断输入指令，实质上shell功能的实现是利用了linux系统本身的特性，只是把shell的功能套进了这个无限循环中

运行结果如下：

```

=== 欢迎使用 MiniShell ===
支持的命令：ls, pwd, date, whoami, echo 等系统命令
特殊命令：cow (演示Copy-On-Write), exit (退出)
=====

=== 开始演示 Copy-On-Write 机制 ===
父进程：分配内存，初始值 = 100，地址 = 0x5555555596b0
父进程：变量值 = 100
子进程：fork() 后，变量值 = 100 (与父进程相同)
子进程：修改变量值为 200
子进程：修改后，变量值 = 200，地址 = 0x5555555596b0
子进程：注意！地址相同，但物理内存页已被复制
父进程：子进程修改后，我的变量值 = 100 (未被改变！)
父进程：这就是 Copy-On-Write：写时才复制内存页
=== Copy-On-Write 演示结束 ===

minishell> ls
helloworld  helloworld.cpp  rfm1  rfm1.c  rfm2  rfm2.c
minishell: 命令执行完成，退出码：0
minishell> 

```

写一个fork:

```

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// --- 预设的数据结构和模拟环境 ---

#define NUM_PAGES 16    // 为了简化输出，我们假设一个进程最多16个虚拟页
#define NUM_FRAMES 64   // 系统总共有64个物理帧

// 模拟物理内存帧
typedef struct {
    bool in_use;          // 此物理帧是否已被分配
    int share_count;       // 有多少个PTE指向此帧（用于COW）
} PhysicalFrame;

// 模拟页表项（PTE）
typedef struct {
    bool present;          // 存在位
    bool writable;         // 可写位
    int frame_index;       // 指向的物理帧的索引
} PTE;

// 模拟进程控制块（PCB）
typedef struct {
    int pid;
    PTE page_table[NUM_PAGES]; // 每个进程有一个页表
} Process;

// 全局模拟环境
PhysicalFrame G_physical_memory[NUM_FRAMES]; // G_ 前缀表示全局变量
Process* G_process_list[10];                // 最多10个进程
int G_next_pid = 100;                        // 下一个要分配的PID
int G_process_count = 0;

// --- 框架提供的辅助函数（无需修改） ---

// 初始化模拟环境
void init_simulation() {

```

```

    for (int i = 0; i < NUM_FRAMES; ++i) {
        G_physical_memory[i].in_use = false;
        G_physical_memory[i].share_count = 0;
    }
    printf("Simulation environment initialized.\n");
}

// 创建初始的父进程以供测试
Process* create_initial_process() {
    Process* parent = (Process*)malloc(sizeof(Process));
    parent->pid = G_next_pid++;
    G_process_list[G_process_count++] = parent;

    // 初始化页表
    for (int i = 0; i < NUM_PAGES; ++i) parent->page_table[i].present = false;

    // 分配几个页面
    // 页面0: 代码页 (只读)
    parent->page_table[0] = (PTE){.present=true, .writable=false, .frame_index=10};
    G_physical_memory[10] = (PhysicalFrame){.in_use=true, .share_count=1};
    // 页面1: 数据页 (可写)
    parent->page_table[1] = (PTE){.present=true, .writable=true, .frame_index=25};
    G_physical_memory[25] = (PhysicalFrame){.in_use=true, .share_count=1};
    // 页面2: 堆页面 (可写)
    parent->page_table[2] = (PTE){.present=true, .writable=true, .frame_index=30};
    G_physical_memory[30] = (PhysicalFrame){.in_use=true, .share_count=1};

    printf("Initial parent process (PID %d) created.\n", parent->pid);
    return parent;
}

// 打印一个进程的页表状态, 用于验证
void print_process_pagetable(Process* proc) {
    if (!proc) return;
    printf("\n--- Page Table for PID: %d ---\n", proc->pid);
    printf("V.Page | Present | Writable | P.Frame | Frame Share Count\n");
    printf("-----\n");
    for (int i = 0; i < NUM_PAGES; ++i) {

```

```

        if (proc->page_table[i].present) {
            int frame_idx = proc->page_table[i].frame_index;
            printf("  %-4d |   %-3s |   %-5s |   %-5d | %-d\n",
                i,
                proc->page_table[i].present ? "Yes" : "No",
                proc->page_table[i].writable ? "Yes" : "No",
                frame_idx,
                G_physical_memory[frame_idx].share_count);
        }
    }
    printf("-----\n");
}

/**
 * @brief 模拟fork()系统调用，创建一个子进程，并实现写时复制(COW)。
 * @param parent 指向父进程的指针。
 * @return 指向新创建的子进程的指针。
 */
Process* my_fork(Process* parent) {

    printf("\n>>> Calling my_fork() on parent PID %d...\n", parent->pid);

    // TODO: 步骤 1: 创建一个新的子进程结构体(Process)，并为其分配一个新的PID。
    Process* child = (Process*)malloc(sizeof(Process));
    if (child == NULL) {
        printf("Error: Failed to allocate memory for child process\n");
        return NULL;
    }
    child->pid = G_next_pid++;

    // 初始化子进程页表为全无效
    for (int i = 0; i < NUM_PAGES; ++i) {
        child->page_table[i].present = false;
    }

    // TODO: 步骤 2: 遍历父进程的页表 (从 i=0 到 NUM_PAGES-1)。
    for (int i = 0; i < NUM_PAGES; ++i) { //这里写一个大循环，内层进行条件判断，实现遍历

```

```

// TODO: 步骤 3: 对于父进程中每一个有效的页表项 (即 present_bit 为 true 的PTE):
if (parent->page_table[i].present) {

    //      a. 将父进程的PTE完整地复制给子进程的对应PTE。
    child->page_table[i] = parent->page_table[i];

    //      b. 检查这个页面是否是可写的 (writable)。如果是, 则需要触发COW机制。
    if (parent->page_table[i].writable) {
        //      c. COW处理:
        //          i. 将父进程中该页面的PTE权限设置为只读 (writable = false)。
        //          ii. 将子进程中该页面的PTE权限也设置为只读 (writable = false)。
        parent->page_table[i].writable = false;
        child->page_table[i].writable = false;

        printf("  Page %d: COW enabled (both processes set to read-only)\n",
i);

        } else {
            printf("  Page %d: Read-only page, directly shared\n", i);
        }

    //      d. 无论是只读页还是被设置为只读的可写页, 现在它们都被共享了。
    //          因此, 需要增加其对应的物理帧的共享计数 (share_count)。
    //          提示: 物理帧的索引是 pte.frame_index。
    int frame_idx = parent->page_table[i].frame_index;
    G_physical_memory[frame_idx].share_count++;

    printf("  Frame %d share count increased to %d\n",
        frame_idx, G_physical_memory[frame_idx].share_count);
}
}

// TODO: 步骤 4: 将新创建的子进程添加到全局进程列表 G_process_list 中, 并更新
G_process_count。

```

```

if (G_process_count < 10) {
    G_process_list[G_process_count] = child;
    G_process_count++;
    printf("Child process (PID %d) created successfully.\n", child->pid);
} else {
    printf("Error: Process list is full!\n");
    free(child);
    return NULL;
}

// TODO: 步骤 5: 返回指向子进程的指针。
return child;
}

// --- 用于测试你的实现的 main 函数 (无需修改) ---
int main() {
    init_simulation();

    Process* parent = create_initial_process();
    printf("\n--- State BEFORE fork ---\n");
    print_process_pagetable(parent);

    Process* child = my_fork(parent);

    printf("\n--- State AFTER fork ---\n");
    printf("Parent process state after fork:\n");
    print_process_pagetable(parent);

    printf("Child process state after fork:\n");
    print_process_pagetable(child);

    // 释放内存 (在真实OS中, 这是由进程退出时完成的)
    free(parent);
    free(child);

    return 0;
}

```

```

Initial parent process (PID 100) created.

--- State BEFORE fork ---

--- Page Table for PID: 100 ---
V.Page | Present | Writable | P.Frame | Frame Share Count
-----|-----|-----|-----|-----
0      | Yes     | No       | 10      | 1
1      | Yes     | Yes      | 25      | 1
2      | Yes     | Yes      | 30      | 1
-----|-----|-----|-----|-----

>>> Calling my_fork() on parent PID 100...
Page 0: Read-only page, directly shared
Frame 10 share count increased to 2
Page 1: COW enabled (both processes set to read-only)
Frame 25 share count increased to 2
Page 2: COW enabled (both processes set to read-only)
Frame 30 share count increased to 2
Child process (PID 101) created successfully.

--- State AFTER fork ---
Parent process state after fork:

--- Page Table for PID: 100 ---
V.Page | Present | Writable | P.Frame | Frame Share Count
-----|-----|-----|-----|-----
0      | Yes     | No       | 10      | 2
1      | Yes     | No       | 25      | 2
2      | Yes     | No       | 30      | 2
-----|-----|-----|-----|-----

Child process state after fork:

--- Page Table for PID: 101 ---
V.Page | Present | Writable | P.Frame | Frame Share Count
-----|-----|-----|-----|-----
0      | Yes     | No       | 10      | 2
1      | Yes     | No       | 25      | 2
2      | Yes     | No       | 30      | 2
-----|-----|-----|-----|-----

[1] + Done                                "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/M
cn.sj5"
likaishuo@lks:~/projects/helloworld$ 
调试活动文件 (helloworld)

```

这是运行结果