lab2

为什么要进行内存管理

隔离与保护：防止在运行多个程序时会访问同一个地址导致冲突。 **易于使用**：我们使用内存管理对内存进行虚拟化，让每个程序觉得，它有一个很大的连续地址空间来放入其代码和数据。这样程序员在编程的时候就可以不需要考虑在哪里存储变量的问题。在内存少的时候，实现对有限的内存更加有效的利用，使之能够支撑更多的任务 内存多的时候，尽可能的不浪费内存资源。

内存管理机制

“翻译”机制：使不同的程序在访问相同的逻辑地址时经过翻译指向不同的物理地址，从而实现隔离与保护的目的。 把连续的很多字节合在一起翻译，让他们翻译前后的数值之差相同，这就是“页”。 **动态分区分配**：处理连续的内存分配，比较典型的是伙伴系统（Buddy System）。 **分区交换技术（swapping in/out）**：抢占并回收处于等待状态进程的分区增大可用内存空间。 **内存覆盖技术**：把程序分割成许多片段，称为覆盖（overlay）。 **虚拟内存——分页管理机制**：每一页都有连续的地址范围，并被映射到物理内存

sv39页表机制

每个页的大小是4KB，也就是4096个字节。

sv39的一个页表项占据8字节（64位）。其中第 53-10 位共44位为一个物理页号，表示这个虚拟页号映射到的物理页号。后面的第 9-0 位共10位则描述映射的状态信息。

sv39中，定义物理地址有 56位，而虚拟地址有 39位。 实际使用时：一个虚拟地址要占用 64位，只有低 39位有效，规定 63–39 位的值必须等于第 38 位的值，否则会认为该虚拟地址不合法，在访问时会产生异常。

物理地址、虚拟地址的最后12位表示的是页内偏移（这个地址在它所在页帧的什么位置（同一个位置的物理地址和虚拟地址的页内偏移相同）），除了最后12位，前面的部分表示的是物理页号或者虚拟页号。

并不是所有的虚拟页都有对应的物理页，虚拟页可能的数目远大于物理页的数目。

多级页表

对页表进行“分级”，把很多页表项组合成一个“大页”，如果这些页表项都非法（没有对应的物 理页），那么只需要用一个非法的页表项来覆盖这个大页，而不需要分别建立一大堆非法页表 项。很多个大页(megapage)还可以组合起来变成大大页(gigapage!)，继而可以有更大的页。

在本次实验中，sv39使用三级页表，有4KiB=4096字节的页，大小为2MiB= 2^21 字节的大 页，和大小为1 GiB 的大大页。

原先的一个39位虚拟地址，被我们看成27位的页号和12位的页内偏移。那么在三级页表下，我们可以把它看成9位的“大大页页号”，9位的“大页页号”（也是大大页内的页内偏移），9位的“页号”（大页的页内偏移），还有12位的页内偏移。

整个Sv39的虚拟内存空间里，有512（2的9次方）个大大页，每个大大页里有512个大页，每个大页里有512个页，每个页里有4096个字节，整个虚拟内存空间里就有512*512*512*4096个字节，是512GiB的地址空间。

建立快表以加快访问效率

为什么要建立

物理内存的访问速度要比 CPU 的运行速度慢很多，按照页表机制一步步走，共需访问4次物理内存（3+1）才能读到我们想要的数据。

根据什么建立

虚拟地址的访问具有时间局部性和空间局部性。 被访问过一次的地址很有可能不远的将来再次被访问、被访问地址附近的地址很有可能在不远的将来被访问。

怎么使用

在 CPU 内部，我们使用快表 (TLB, Translation Lookaside Buffer) 来记录近期已完成的虚拟页号到物理页号的映射。要做一个映射时，先到 TLB 里面去查一下，有的话我们就可以直接完成映射，而不用访问那么多次内存了。 有时需要 sfence.vma 指令刷新 TLB （切换到了一个与先前映射方式完全不同的页表、手动修改一个页表项之后）（不加参数的or后面加一虚拟地址，只刷新这个虚拟地址的映射）

练习1：理解first-fit连续物理内存分配算法（思考题）

最先匹配（first-fit）策略的实现思路是找到第一个足够大的块，将请求的空间返还给用户，将剩余的空间放入空闲列表。具体实现的原理如下：

我们会维护一个空闲列表，在收到内存请求的时候沿着空闲列表进行扫描，寻找第一个足够大的空间区 块分配给用户，如果选中的区块比请求的内存大得多，则进行拆分返回用户所需要的空间并将剩余空间添加到空闲列表中。该内存分配算法实现相对简单，并且效率相对较高。由于他在分配过程中是找到第一个比请求内存大的分区，因此无需遍历整个空闲列表，这就减小了时间开销。但是这个算法容易产生外部碎片，同时如果请求的内存较大时，时间开销较大

练习1：实现Best-fit连续物理内存分配算法

具体实现

数据结构：

free_area_t：用于管理空闲内存块，包括一个双向链表free_list和一个记录空闲块数量的nr_free。 Page：表示内存页的结构体，包含标志位flags、属性property和引用计数ref等。

初始化：

best_fit_init：初始化空闲链表并将空闲块数量设为0。 best_fit_init_memmap：初始化一段连续的内存页，将其标记为空闲并插入空闲链表中。

内存分配：

best_fit_alloc_pages：遍历空闲链表，找到一个大小最接近但不小于所需大小的块。如果找到合适的块，则从中分配所需的页，并更新剩余块的大小和空闲链表。

内存释放：

best_fit_free_pages：将释放的页重新插入空闲链表，并尝试与相邻的空闲块合并，以减少内存碎片。

辅助函数：

best_fit_nr_free_pages：返回当前空闲页的数量。 basic_check和best_fit_check：用于测试内存管理器的正确性。

管理器结构体：

best_fit_pmm_manager：定义了内存管理器的接口，包括初始化、内存分配、内存释放和检查函数。

算法思路

best-fit算法的思路就是在遍历所有空闲块，选择一个大小最接近但不小于所需大小的块， **优点：**

1. 减少内存碎片：最佳适配算法通过选择最小的适合块来满足内存请求，尽量减少了内存碎片的产生。
2. 高效利用内存：由于每次分配都选择最小的适合块，内存利用率较高，能够更好地利用可用内存。
3. 适合小内存请求：对于频繁的小内存请求，最佳适配算法能够有效地找到合适的内存块，减少浪费。

缺点：

1. 搜索时间长：最佳适配算法需要遍历整个空闲链表来找到最适合的内存块，搜索时间较长，尤其是在空闲块较多时，性能会受到影响。
2. 容易产生小碎片：虽然最佳适配算法减少了大块内存的碎片，但容易产生很多小碎片，这些小碎片可能无法满足后续的内存请求。
3. 复杂性高：实现最佳适配算法相对复杂，需要维护空闲链表并进行频繁的插入和删除操作。

可以改进的方面

1. 使用链表进行增删改查都会有较大的时间开销，实现略微简单但是会降低性能，如果使用一些高级的数据结构例如红黑树，AVL树等更高级的数据结构，能过够有效的提高性能。
2. 存储空闲块时按照地址进行排序，这样可能会降低性能，如果能够按照大小排序，分配时就可以采用二分查找迅速找到需要的空闲块，但是排序也需要时间开销，综合考虑第一条可以使用优先队列来方便分配和查找
3. 或许可以对分配的内存块进行限制，也就是**内存块大小-需要的内存块大小**不能过小导致产生特别小的内存块无法使用。

扩展练习Challenge：buddy system（伙伴系统）分配算法

该部分见buddy system的设计文档buddy.md

扩展练习Challenge：硬件的可用物理内存范围的获取方法。

如果 OS 无法提前知道当前硬件的可用物理内存范围，请问你有何办法让 OS 获取可用物理内存范围?

1. BIOS/UEFI 内存映射表： 操作系统在启动时可以通过 BIOS 或 UEFI 提供的内存映射表来获取物理内存的布局和可用范围。这些表格通常包含了系统中所有内存块的起始地址和长度，以及它们的类型（如可用、保留、ACPI 等）。

2. ACPI 表： 高级配置与电源接口（ACPI）提供了一组标准化的表格，操作系统可以通过这些表格获取系统硬件信息，包括内存布局。

3. EFI 固件接口： 在使用 UEFI 的系统中，操作系统可以通过 EFI 固件接口获取内存映射信息。EFI 提供了一组 API，操作系统可以调用这些 API 来获取内存信息。

4. 内存检测算法： 操作系统可以在启动时通过内存检测算法来探测可用的物理内存范围。这些算法通常通过访问特定的内存地址并检查响应来确定哪些内存是可用的。

5. 硬件抽象层（HAL）： 一些操作系统使用硬件抽象层（HAL）来与底层硬件交互。HAL 可以提供获取物理内存范围的功能。