Lab 5: RV64 动态内存分配与缺页异常处理

1 实验目的

在充分理解前序实验中 RISC-V 地址空间映射机制与任务调度管理的前提下，进一步了解与**动态**内存管理相关的重要结构，实现基本的内存管理算法，并了解缺页异常的处理流程。

2 实验内容及要求

- 了解 **Buddy System** 和 **Slub Allocator** 物理内存管理机制的实现原理，并使用 **Buddy System** 和 **Slub Allocator** 管理和分配物理内存。
- 修改进程信息结构，补充虚拟内存相关信息字段 **vm_area_struct**数据结构。
- 实现 **mmap munmap** 系统调用。
- 添加缺页异常处理 **Page Fault Handler**。

请各小组独立完成实验，任何抄袭行为都将使本次实验判为0分。

请跟随实验步骤完成实验并根据文档中的要求记录实验过程，最后删除文档末尾的附录部分，并命名为"**学号1*姓名1*学号2_姓名2_lab5.pdf**"，你的代码请打包并命名为"**学号1*姓名1*学号2_姓名2_lab5**"，文件上传至学在浙大平台。

本实验以双人组队的方式进行，**仅需一人提交实验，**默认平均分配两人的得分（若原始打分为X，则可分配分数为2X，平均分配后每人得到X分）。如果有特殊情况请单独向助教反应，出示两人对于分数的钉钉聊天记录截图。单人完成实验者的得分为原始打分。

姓名	学号	分工	分配分数
徐铭	3210102037	后60%	50%
王思雨	3210106039	前40%	50%

3 实验步骤

3.1 搭建实验环境

你可以从 lab5.zip 下载本实验提供好的代码。

```
.
├── Makefile
├── arch
│   └── riscv
│       ├── Makefile
│       ├── kernel
│       │   ├── ...
│       │   ├── mm.c（新增）
│       │   └── slub.c（新增）
│       └── user
│           ├── Makefile
│           ├── lib
│           │   ├── Makefile
│           │   ├── include
│           │   │   ├── ...
│           │   │   └── mm.h（新增）
│           │   └── src
│           │       ├── ...
│           │       └── mm.c（新增）
│           ├── src
│           │   ├── Makefile
│           │   ├── test1.c
│           │   ├── test2.c
│           │   ├── test3.c
│           │   ├── test4.c
│           │   └── test5.c
│           └── users.S
└── include
    ├── ...
    ├── list.h（新增）
    ├── mm.h（新增）
    └── slub.h（新增）
```

本次实验新增了如上文件，其中`kernel/mm.c`与`kernel/slub.c`负责内核内存管理，`user/lib/src/mm.c`负责用户态内存管理。

我们在`list.h`中提供了linux list的实现，方便大家使用。代码实现部分给出了本实验涉及到的用法提示，足够大家完成本次实验，如果感兴趣可以参看List 链表的使用教程学习相关用法。

3.2 实现 buddy system （30%）

在 Lab 4 中，我们为每个进程分配的内存都是直接从当前内存的末尾位置直接追加一个页实现的。这样虽然实现简单，但难以回收不再使用的页，也无法有效的按需分配页的大小。从本节开始，我们将实现一个页级的动态内存分配器——Buddy System。

3.2.1 buddy system算法

如果只是简单的分配页，那还有更加简单的方法，比如用一个队列来存储当前空闲的页，需要分配时就从队列中取出，不使用后再入队即可。然而，内核经常需要和硬件交互，有些硬件要求数据存放在内存中的位置是连续的，即连续的物理内存，因此内核需要一种能够高效分配连续内存的方法。

Buddy System 的思路十分简单：

假设我们系统一共有8个可分配的页面（可分配的页面数需保证是2^n）：

```
memory: （这里的 page 0～7 计做 page X）
+--------+--------+--------+--------+--------+--------+--------+--------+
| page 0 | page 1 | page 2 | page 3 | page 4 | page 5 | page 6 | page 7 |
+--------+--------+--------+--------+--------+--------+--------+--------+
```

3.2.1.1 分配过程：

当一个内存请求到来时，Buddy System 将连续的内存不断二等分，直到匹配内存申请要求为止（注意buddy system只能分配2的幂次的内存）。例如我们申请一个1 page大小的内存，buddy system会进行如下的分配操作：

```
 0-7  +---> 0-3 +---> 0-1 +---> Page 0 --> 分配
(root)|         |         |
      |         |         +---> Page 1
      |         |
      |         +---> 2-3
      |
      +---> 4-7
```

Buddy system首先把 page 0-7 拆分成 page 0-3 和 page 4-7 两个节点，然后对左子节点继续二分，直到拆分成 page 0、page 1、page 2-3 和 page 4-7 四个节点，最后把 page 0 分配出去。

> 注意在上图中我们把树横过来画，在上方的是左子节点。

经过上述分配过程后，Buddy system 可以分配的节点变成了page 1、page 2-3 和 page 4-7。如果我们此时再申请一个2 page的内存，Buddy system 会搜索大小为2的内存区域，然后将 page 2-3 分配出去：

```
 0-7  +---> 0-3 +---> 0-1 +---> Page 0 --> 分配
(root)|         |         |
      |         |         +---> Page 1
      |         |
      |         +---> 2-3 --> 分配
      |
      +---> 4-7
```

总结一下Buddy system的分配过程，就是**如果有大小匹配的节点，则分配，否则拆分节点（左子节点优先）或报错无法分配。**为了方便描述，我们把一次申请得到的内存区域称为**“内存块”。**

3.2.1.2 回收过程：

> 注意我们要求只能以内存块（分配过的一块）为单位进行回收，例如上次分配了Page2~3，那么回收的时候不能只回收2，必须是2～3一起回收。

继续上面的例子，现在我们回收了page 0，那么buddy system需要对 page 0 和 page 1进行合并，如下图所示：

```
 1:
 0-7  +---> 0-3 +---> 0-1 +---> Page 0 --> 回收
(root)|         |         |
      |         |         +---> Page 1
      |         |
      |         +---> 2-3 --> 分配
      |
      +---> 4-7

 2:
 0-7  +---> 0-3 +---> 0-1 +---> Page 0 --> 现在都是空的，合并
(root)|         |         |
      |         |         +---> Page 1 --> 现在都是空的，合并
      |         |
      |         +---> 2-3 --> 分配
      |
      +---> 4-7

 3:
  0-7  +---> 0-3 +---> 0-1
(root) |         |
       |         +---> 2-3 --> 分配
       |
       +---> 4-7
```

Buddy system的回收过程，就是回收目标节点，并判断能否与兄弟节点合并，如果能则合并，然后递归判断父节点能否合并。

3.2.2 buddy system实现

> 本节仅提供了一种比较直观的实现思路（但和实际上有区别），你可以按照自己的想法实现。
>
> 这里提供一个巧妙的实现讲解 https://coolshell.cn/articles/10427.html，且性能较高（无需递归调用函数）你可以做参考并编写此版本，但请不要直接复制代码。

> Linux 内核编写规范中并不限制使用递归函数，但是由于内核栈是有限大小的。所以编写递归函数一定要注意调用层数避免栈溢出。另一方面，递归函数虽然简单易懂，但迭代版本有时会性能更佳（有些递归函数会被编译器优化）。

Buddy system 有很多种实现方式，在本实验中我们采用满二叉树的方法来实现：

对于每一个节点，我们需要有两个属性值：

- 该节点是否已经被拆分成子节点
- 该节点可以分配的最大连续物理内存大小（已拆分则是左右子节点的可分配的最大连续物理内存最大值，未拆分则是两者之和）

n 个 page 的分配过程：

```
1. 从根节点开始搜索

2. 如果当前节点可分配最大连续内存大小大于 n，则递归搜索子节点（左子节点优先）

3. 如果当前节点可分配最大连续内存大小等于n，判断节点是否被拆分
   3.1 如果当前节点已经被拆分，则递归搜索子节点（左子节点优先）
   3.2 如果当前节点未被拆分，则分配该节点，更新可分配内存大小，返回对应物理内存区域的首地址

4. 如果当前节点可分配最大连续内存大小小于n，则该节点无法被分配，返回 0

（注意递归回去时更新相关节点的拆分信息，最大连续物理内存大小信息）
```

例子如下：

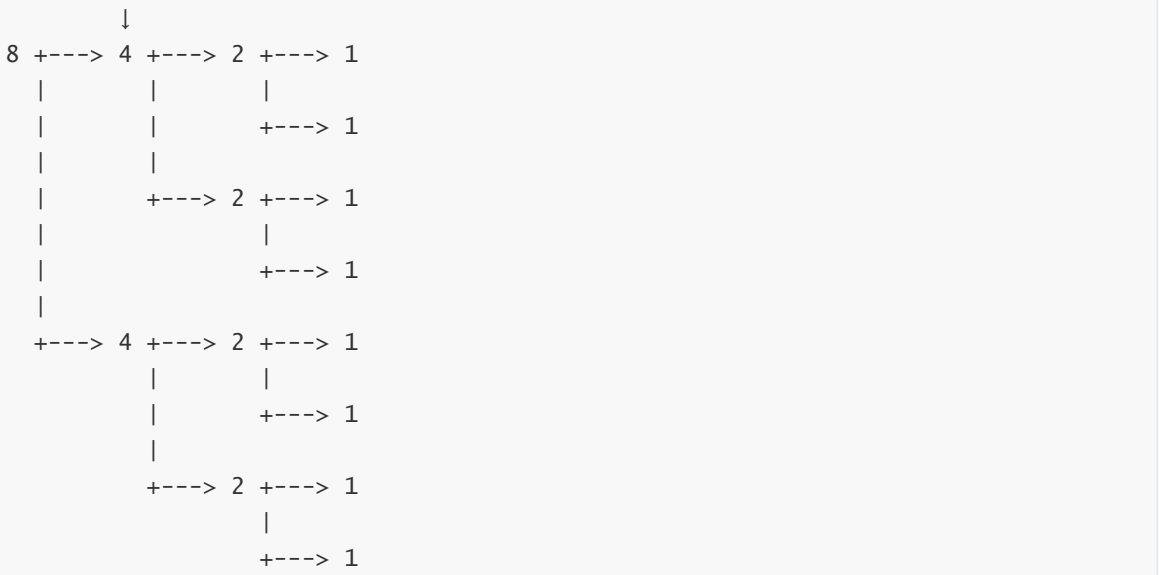
```
memory: （这里的 page 0~7 计做 page X）

+--------+--------+--------+--------+--------+--------+--------+--------+
| page 0 | page 1 | page 2 | page 3 | page 4 | page 5 | page 6 | page 7 |
+--------+--------+--------+--------+--------+--------+--------+--------+
```

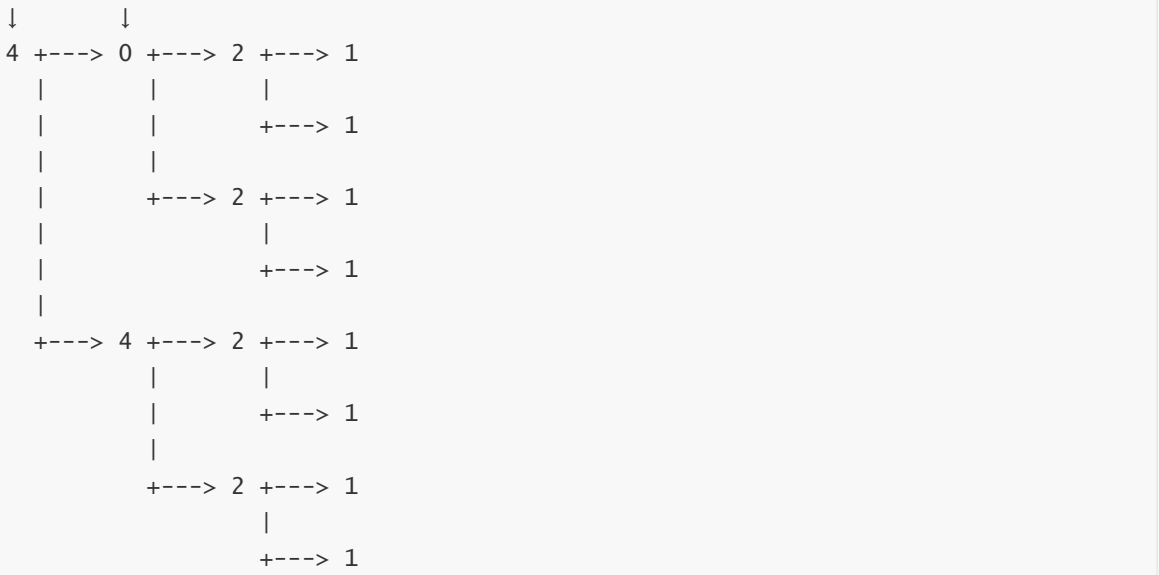
alloc_pages(3) 请求分配3个页面

由于buddy system每次必须分配2^n个页面，所以我们将 3 个页面向上扩展至 4 个页面。

我们从树根出发查找到恰好满足当前大小需求的节点：



找到了 0~3 这个对应连续长 4 的节点，由于分配出去了，所以现在该节点能分配长度为 0。



由于树根所对应的 8 个物理地址连续页，其中的 4 个被分配出去了

所以树根对应的可分配物理地址页数为 4 个。

对应的，回收内存的时候，只需要把对应节点的连续内存大小标回原始大小即可。但需要注意，如果回收之后，左右儿子都处于空闲状态，那么还需要更新其父亲节点的信息。（相当于有两块区域连起来了）

回收过程请大家自己思考应该如何实现。

3.2.2 实现对应数据结构以及接口

数据结构和接口定义在 `mm.h` 文件当中，代码实现在 `mm.c` 中。

我们定义buddy system的数据结构如下所示，并实例化了一个buddy system对象叫做 `buddy_system`。

```
// 定义buddy system可分配的内存大小为16MB
#define MEMORY_SIZE 0x1000000

typedef struct {
  bool initialized;
  uint64_t base_addr;
  unsigned int bitmap[8192];
} buddy;

static buddy buddy_system;
```

其中`initialized`表示是否完成初始化，`base_addr`表示`buddy_system`管理的全部内存区域的起始地址，`bitmap`是我们的满二叉树，`bitmap`的元素类型为`unsigned int`，我们用它来表示可分配的连续内存大小以及是否已经被拆分（详见后文）。

> 这里我们用了一个数组来实现满二叉树，bitmap[1]表示根节点，bitmap[2*x*] *和* *bitmap[2*x +1]分别表示bitmap[x]的左子节点和右子节点。
>
> 我们定义buddy system可以管理的内存大小为16MB，也就是4096个 page，我们规定buddy system可以管理的最小内存大小为一个 page，因此我们的 bitmap 需要8191个节点，由于不使用bitmap[0]，所以bitmap的大小为8192。

在`mm.h`中我们定义了 buddy system 的接口，如下所示：

```
int alloced_page_num(); // 已停用

void init_buddy_system();

uint64_t alloc_pages(unsigned int num);

uint64_t alloc_page();

void free_pages(uint64_t pa);
```

需要大家实现的各接口的含义如下：

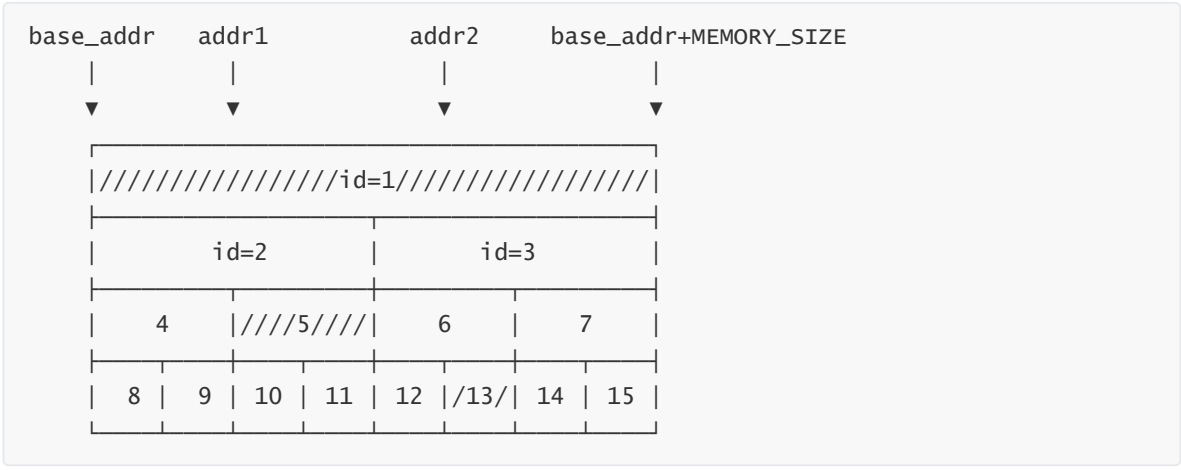
1. `init_buddy_system()` 负责初始化buddy system

2. `alloc_pages(num)` 负责分配num个page，并返回分配的物理内存的首地址
3. `free_pages` 负责回收以 `pa` 为起始地址的内存块

> 为了适配过去的lab，我们保留了 `alloc_page` 和 `alloced_page_num` 两个接口，但是 `alloced_page_num` 已经停用。`alloc_page` 实际上就是`alloc_pages(1)`。

在实现过程中同学们可能会用到的函数与宏包括：

1. `uint64_t get_index(uint64_t pa)`：获取bitmap中，以物理地址 `pa` 起始的最上层的节点的index



1. 在上图中，`get_index(base_addr)==1`，`get_index(addr1)==5`，`get_index(addr2)==13`。
2. `uint64_t get_addr(int index)`：获取节点 `index` 的起始物理地址
3. `uint32_t get_block_size(int index)`：获取节点 `index` 的初始大小
4. `set_split(x)`：`bitmap[index] = set_split(bitmap[index])` 表示将节点 `index` 标记为 `splitted`
5. `set_unsplit(x)`：与 `set_split(x)` 相反
6. `get_size(x)`：`get_size(bitmap[index])` 表示获取节点 `index` 的可分配连续内存大小
7. `check_split(x)`：`check_split(bitmap[index])` 表示节点 `index` 是否已经被拆分

请在下方代码框中补充完整你的代码：

```
void init_buddy_system() {
  // TODO: 初始化buddy system
  // 1. 将buddy system的每个节点初始化为可分配的内存大小，单位为PAGE_SIZE
  // 注意我们用buddy_system.bitmap实现一个满二叉树，其下标变化规律如下：如果当前节点的下标是X，那么左儿子就是 `X * 2` ，右儿子就是 `X * 2 + 1` ，X 从1开始。
  // 那么，下标为 X 的节点可分配的内存为多少呢？
  // 2. 将buddy system的base_addr设置为&_end的物理地址
  // 3. 将buddy system的initialized设置为true
  for(int i=8191;i>8191/2;i--){
    buddy_system.bitmap[i]=1;
  }
```

```
  for(int i=8191/2;i>0;i--){
    buddy_system.bitmap[i]=buddy_system.bitmap[i*2]+buddy_system.bitmap[i*2+1];
  }

  buddy_system.base_addr=PHYSICAL_ADDR(&_end);
  buddy_system.initialized=1;

};
```

```
uint64_t alloc_buddy(int index, unsigned int num) {
  // TODO: 找到可分配的节点并完成分配，返回分配的物理页面的首地址（通过get_addr函数可以获取节点index的起始地址）
  // 1. 如果当前节点大于num，则查看当前节点的左右儿子节点
  // 提示：通过get_size函数可以获取节点index的可分配连续内存大小
  // 2. 如果当前节点的可分配连续内存等于num，且当前节点没有被拆分，则分配当前节点
  // 提示：通过check_split函数可以获取节点index的状态
  // 3. 如果当前节点的可分配连续内存等于num且已经被拆分，则查看当前节点的左右儿子节点
  // 4. 如果当前节点的可分配连续内存小于num，则分配失败返回上层节点
  // 如果完成分配，则要递归更新节点信息，将涉及到的节点的可分配连续内存更新，
  // 已拆分则是左右子节点的可分配的最大连续物理内存最大值，未拆分则是两者之和，并使用set_split更新节点的状态
  if(get_size(buddy_system.bitmap[index]) > num ){
    uint64_t res=alloc_buddy(index*2,num);
    if(res==0){
      res=alloc_buddy(index*2+1,num);
    }
    if(res!=0){

 buddy_system.bitmap[index]=get_size(get_size(buddy_system.bitmap[index*2])>get_size(buddy_system.bitmap[index*2+1])?
buddy_system.bitmap[index*2]:buddy_system.bitmap[index*2+1]);
      buddy_system.bitmap[index]=set_split(buddy_system.bitmap[index]);
    }
    return res;
  }
  else if(get_size(buddy_system.bitmap[index]) == num ){
    if(check_split(buddy_system.bitmap[index])==0){//没有分裂
      buddy_system.bitmap[index]=0;//分配
      return get_addr(index);
    }
    else{//分裂了
      uint64_t res=alloc_buddy(index*2,num);
      if(res==0){
        res=alloc_buddy(index*2+1,num);
      }
      if(res!=0){

 buddy_system.bitmap[index]=get_size(get_size(buddy_system.bitmap[index*2])>get_size(buddy_system.bitmap[index*2+1])?
buddy_system.bitmap[index*2]:buddy_system.bitmap[index*2+1]);
        buddy_system.bitmap[index]=set_split(buddy_system.bitmap[index]);
      }
      return res;
    }
  }
```

```
  else{
    return 0;
  }
}
```

```
uint64_t alloc_pages(unsigned int num) {
  // 分配num个页面，返回分配到的页面的首地址，如果没有足够的空闲页面，返回0
  if (!buddy_system.initialized) {
    init_buddy_system();
  }
  // TODO:
  // 1. 将num向上对齐到2的幂次
  // 2. 调用alloc_buddy函数完成分配
  unsigned int new_num=1;
  while(new_num<num) new_num<<=1;
  return alloc_buddy(1,new_num);
}
```

```
void free_buddy(int index) {
  // TODO: 释放节点index的页面
  // 1. 首先判断节点index的状态，如果已经被拆分，则不能直接释放。异常状态，报错并进入死循环。
  // 2. 如果没有被拆分，则恢复节点index的状态为初始状态
  // 3. 如果该节点与其兄弟节点都没有被使用，则合并这两个节点，并递归处理父节点
  // 提示：使用check_split函数可以获取节点index的状态
  // 提示：使用set_unsplit函数可以将节点index的状态恢复为初始状态
  // 提示：使用get_block_size函数可以获取节点index的初始可分配内存大小
  if(check_split(buddy_system.bitmap[index])==1){
    printf("error: can't free\n");
    while (1){};
  }
  buddy_system.bitmap[index]=get_block_size(index);
  buddy_system.bitmap[index]=set_unsplit(buddy_system.bitmap[index]);

  if(index!=1&&get_block_size(index^1)==get_size(buddy_system.bitmap[index^1])){
    free_buddy(index/2);
  }
  return;
}
```

```
void free_pages(uint64_t pa) {
  // TODO: find the buddy system node according to pa, and free it.
  // 注意，如果该节点的状态为已经被拆分，则应该释放其左子节点
  // 提示：使用get_index函数可以获取pa对应的最上层节点的下标
  // check that the pa is in the buddy system
  if (pa < buddy_system.base_addr || pa >= buddy_system.base_addr + MEMORY_SIZE)
{
    printf("error: pa out of range\n");
    return;
  }
  // TODO: find the buddy system node according to pa, and free it.
```

```
  // 注意，如果该节点的状态为已经被拆分，则应该释放其左子节点
  // 提示：使用get_index函数可以获取pa对应的最上层节点的下标
  int index=get_index(pa);
  if(check_split(buddy_system.bitmap[index])==1){
    index=index/2;
  }
  free_buddy(index);

}
```

3.2.3 测试

同学们可以执行`make buddy && make run`来测试`buddy system`的正确性。如果能正常运行调度过程，则通过测试。注意此处只检测了`alloc_pages()`的正确性，不能保证buddy system完全正确。

> 我们在lab5中修改了调度测例，现在会按照1 ~ 5的顺序每个进程执行一次

3.3 实现 kmalloc 和 kfree （10%）

3.3.1 SLUB分配器

Buddy System 管理的是页级别，还不能满足内核需要动态分配小内存的需求（比如动态分配一个比较小的`struct`），管理小内存使用的是 SLUB 分配器，由于 SLUB 分配器代码比较复杂，本次实验直接给出，其功能依赖 Buddy System 的功能，因此你需要正确的实现 Buddy System 的接口后，才能使用 SLUB 分配器。

> SLUB 不要求做过多了解，这里仅仅简单介绍，更多信息可以搜索学习。

SLUB 如何分配小内存？首先从 Buddy System 申请一个页 （4096字节），分成512 个长度为 8 字节的对象，每当申请一个小于 8 字节的空间的时候，就分配一个出去，回收的时候直接收回即可，【具体实现中使用了链表】

同理还可以创建 分配粒度为 16，32，64， ...，2048 Byte 的 SLUB 分配器，这样就完成了小内存动态分配的功能。

以下是两者的关系。

- Buddy System 是以**页 (Page)** 为基本单位来分配空间的，其提供两个接口 **alloc_pages** 和 **free_pages** 。

- SLUB 在初始化时，需要预先申请一定的空间来做数据结构和 Cache 的初始化，依赖于 Buddy System提供的上述接口。
- 实验所用的 SLUB 提供8, 16, 32, 64, 128, 256, 512, 1024, 2048（单位：Byte）九种object-level 的内存分配/释放功能。(**kmem_cache_alloc （分配内存） / kmem_cache_free （释放内存）**
- slub.c 中的 (**kmalloc / kfree**) 提供内存动态分配/释放的功能。根据请求分配的空间大小，来判断是通过kmem_cache_alloc 来分配 object-level 的空间，还是通过 alloc_pages 来分配 page-level 的空间。kfree同理。

实现了SLUB之后，我们就可以使用 `kmalloc` 和 `kfree` 来进行内存的管理了。

3.3.2 SLUB 接口说明

`slub.c` 中有全局变量 `struct kmem_cache *slub_allocator[NR_PARTIAL] = {};` ，存储了所有的 SLUB 分配器的指针，他们管理的 Object-level 大小分别为 `8, 16, 32, 64, 128, 256, 512, 1024, 2048 Byte` 。

你只需要用到以下两个接口。

```
void *kmem_cache_alloc(struct kmem_cache *cachep)
```

传入 SLUB 分配器的指针，返回该分配器管理的 Object-level 对应大小一个内存的首地址。

例子： `kmem_cache_alloc(slub_allocator[0])` 代表分配一个大小为 8 字节的空间，返回值就是这段空间的首地址。

```
void kmem_cache_free(void *obj)
```

传入一个内存地址，管理该内存地址的 SLUB 分配器将回收该内存地址。

例子： `kmem_cache_free(obj)` 代表回收 obj 地址开头的这段空间，这个地址对应的空间大小在 slub 分配的时候有对应的自动存储管理，调用该函数不需要考虑。

3.3.2 实现内存分配接口 kmalloc 和 kfree

按照代码提示完成 `kmalloc` 和 `kfree` 函数，相关代码在 `slub.c` 文件中。

将你编写好的 kmalloc 函数复制到下面代码框内。

```
void *kmalloc(size_t size) {
  int objindex;
  void *p = NULL;

  if (size == 0) return NULL;

  // TODO:
  // 1. 判断 size 是否处于 slub_allocator[objindex] 管理的范围内
  // 2. 如果是就使用 kmem_cache_alloc 接口分配，否则使用 alloc_pages 接口分配

  // size 若在 kmem_cache_objsize 所提供的范围之内，则使用 slub allocator
  // 来分配内存
  for (objindex = 0; objindex < NR_PARTIAL; objindex++) {
    if(size <= kmem_cache_objsize[objindex]) {
      // TODO:
      p=kmem_cache_alloc(slub_allocator[objindex]);
      return p;
    }
  }

  // size 若不在 kmem_cache_objsize 范围之内，则使用 buddy system 来分配内存
  if (objindex >= NR_PARTIAL) {
    // TODO:
    p=alloc_pages((size - 1) / PAGE_SIZE + 1);

    set_page_attr(p, (size - 1) / PAGE_SIZE + 1, PAGE_BUDDY);
  }

  return p;
}
```

该函数的功能是申请大小为 size 字节的连续物理内存，成功返回起始地址，失败返回 NULL。

- 每个 slub allocator 分配的对象大小见**kmem_cache_objsize**。
- 当请求的物理内存**小于等于**2^11字节时，从 `slub allocator` 中分配内存。
- 当请求的物理内存大小**大于**2^11字节时，从 `buddy system` 中分配内存。

将你编写好的 kfree 函数复制到下面代码框内。

```
void kfree(const void *addr) {
  struct page *page;

  if (addr == NULL) return;

  // 获得地址所在页的属性
  page = ADDR_TO_PAGE(addr);

  // TODO: 判断当前页面属性，使用 free_pages 接口回收或使用 kmem_cache_free 接口回收
```

```
  if (page->flags == PAGE_BUDDY) {
    // TODO:
    free_pages((uint64_t)addr);

    clear_page_attr(ADDR_TO_PAGE(addr)->header);

  } else if (page->flags == PAGE_SLUB) {
    // TODO:
    kmem_cache_free(addr);
  }

  return;
}
```

该函数的功能是回收 `addr` 内存，每一块分配出去的内存都有用 flags 变量保存是 buddy system 分配的还是用 slub 分配器分配的，你只需要对应接口回收即可。

3.5 实现系统调用 （30%）

现在我们有了 kmalloc 和 kfree 的动态内存分配函数，接下来我们需要实现内存相关的系统调用，使得用户应用程序可以动态申请与释放内存。相关的系统调用编号定义在`syscall.h`中。

3.5.1 vm_area_struct

尽管我们已经有了页表来管理我们的虚拟地址空间，但页表主要用于以页为粒度的细粒度内存管理，这在操作系统层面处理虚拟地址空间过于繁琐。为了更高效地管理虚拟地址空间，Linux引入了`vm_area_struct`数据结构。

`vm_area_struct`是一个重要的内核数据结构，它提供了一种更宏观的方式来描述虚拟地址空间的属性。不同于页表，`vm_area_struct`能够以更大的**内存块**为单位来管理虚拟地址空间的不同部分，使操作系统能够更有效地分配、释放和保护内存区域。这种更高级的管理使得虚拟内存管理更为灵活且易于理解，有助于操作系统的性能优化和应用程序的稳定性。

`vm_area_struct` 是虚拟内存管理的基本单元，`vm_area_struct` 保存了有关连续虚拟内存区域(简称vma)的信息。linux 具体某一进程的虚拟内存区域映射关系可以通过 procfs 读取 `/proc/pid/maps` 的内容来获取：

比如，如下一个常规的 `bash` 进程，假设它的进程号为 `7884`，则通过输入如下命令，就可以查看该进程具体的虚拟地址内存映射情况(部分信息已省略)。

```
#cat /proc/7884/maps
556f22759000-556f22786000 r--p 00000000 08:05 16515165
/usr/bin/bash
556f22786000-556f22837000 r-xp 0002d000 08:05 16515165
/usr/bin/bash
556f22837000-556f2286e000 r--p 000de000 08:05 16515165
/usr/bin/bash
556f2286e000-556f22872000 r--p 00114000 08:05 16515165
/usr/bin/bash
556f22872000-556f2287b000 rw-p 00118000 08:05 16515165
/usr/bin/bash
556f22fa5000-556f2312c000 rw-p 00000000 00:00 0                          [heap]
7fb9edb0f000-7fb9edb12000 r--p 00000000 08:05 16517264
/usr/lib/x86_64-linux-gnu/libnss_files-2.31.so
7fb9edb12000-7fb9edb19000 r-xp 00003000 08:05 16517264
/usr/lib/x86_64-linux-gnu/libnss_files-2.31.so
...
7ffee5cdc000-7ffee5cfd000 rw-p 00000000 00:00 0                          [stack]
7ffee5dce000-7ffee5dd1000 r--p 00000000 00:00 0                          [vvar]
7ffee5dd1000-7ffee5dd2000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0
[vsyscall]
```

从中我们可以读取如下一些有关该进程内虚拟内存映射的关键信息：

- `vm_start` ：(第1列) 指的是该段虚拟内存区域的开始地址；
- `vm_end` ：(第2列) 指的是该段虚拟内存区域的结束地址；
- `vm_flags` ：(第3列) 该 `vm_area` 的一组权限(rwx)标志，`vm_flags` 的具体取值定义可参考linux源代码的这里(linux/mm.h)
- `vm_pgoff` ：(第4列) 虚拟内存映射区域在文件内的偏移量；
- `vm_file` ：(第5/6/7列)分别表示：映射文件所属设备号/以及指向关联文件结构的指针(如果有的话，一般为文件系统的inode)/以及文件名；

其它保存在 `vm_area_struct` 中的信息还有：

- `vm_ops` ：该 `vm_area` 中的一组工作函数;
- `vm_next/vm_prev`: 同一进程的所有虚拟内存区域由**链表结构**链接起来。

在我们的实现中，我们使用linux list来实现链表。

`vm_area_struct` 结构已为你定义在 `task_manager.h` 文件中，如下。

```
/* 虚拟内存区域 */
typedef struct {
  unsigned long pgprot;
```

```
} pgprot_t;
struct vm_area_struct {
  /* Our start address within vm_area. */
  unsigned long vm_start;
  /* The first byte after our end address within vm_area. */
  unsigned long vm_end;
  /* linked list of VM areas per task, sorted by address. */
  struct list_head vm_list;
  // vm_page_prot和vm_flags的具体含义本实验不做要求，可以直接把vm_flags用于保存
page_table的权限位。
  /* Access permissions of this VMA. */
  pgprot_t vm_page_prot;
  /* Flags*/
  unsigned long vm_flags;
  /* mapped */
  bool mapped;  // mmap是延迟分配内存的，mapped表示该内存块是否已经实际映射到了物理空间
};
```

有了`vm_area_struct`之后，我们需要在`task_struct`中加入一个`vm_area_struct`的链表来管理每个进程动态申请的虚拟地址空间。

```
/* 内存管理 */
struct mm_struct {
  struct vm_area_struct *vm;
};


/* 进程数据结构 */
struct task_struct {
    ...
  struct mm_struct mm;

};
```

在task init时，你需要初始化`mm`，**将你编写好的 task_init 系统调用复制到下面代码框内：**

```
// task_manager.c

// initialize tasks, set member variables
void task_init(void) {
  // DONE
    // initialize task[i]
    // get the task_struct using alloc_page()
    // set state = TASK_RUNNING, counter = 0, priority = 5,
    // blocked = 0, pid = i, thread.sp, thread.ra
    struct task_struct* new_task =(struct task_struct*)
(VIRTUAL_ADDR(alloc_page()));
    new_task->state = TASK_RUNNING;
    new_task->counter = 0;
```

```
    new_task->priority = 5;
    new_task->blocked = 0;
    new_task->pid = i;
    task[i] = new_task;
    task[i]->thread.sp = (uint64_t)task[i] + PAGE_SIZE; // 内核栈的栈底
    task[i]->thread.ra = (uint64_t)__init_sepc;

    // TODO: initialize task[i]->mm.vm using kmalloc and set the vm_list using
INIT_LIST_HEAD
    task[i]->mm.vm = kmalloc(sizeof(struct vm_area_struct));
    INIT_LIST_HEAD(&(task[i]->mm.vm->vm_list));

    // DONE: task[i]的物理地址
    uint64_t task_addr = PHYSICAL_ADDR((uint64_t)&user_program_start) + i *
PAGE_SIZE;

    // DONE: 完成用户栈的分配，并创建页表项，将用户栈映射到实际的物理地址
    // 1. 为用户栈分配物理页面，使用alloc_page函数
    // 2. 为用户进程分配根页表，使用alloc_page函数
    // 3. 将task[i]->sscratch指定为虚拟空间下的栈地址，即0x1001000 + PAGE_SIZE（注意栈
是从高地址到低地址使用的）
    // 4. 正确设置task[i]->satp，注意设置ASID
    // 5. 将用户栈映射到实际的物理地址，使用create_mapping函数
    // 6. 将用户程序映射到虚拟地址空间，使用create_mapping函数
    // 7. 将将虚拟地址 0xffffffc000000000 开始的 16 MB 空间映射到起始物理地址为
0x80000000 的 16MB 地址空间，注意此时 &rodata_start、... 得到的是虚拟地址还是物理地址？我
们需要的是什么地址？
    // 8. 对内核起始地址 0x80000000 的16MB空间做等值映射（将虚拟地址 0x80000000 开始的
16 MB 空间映射到起始物理地址为 0x80000000 的 16MB 空间），PTE_V | PTE_R | PTE_W |
PTE_X 为映射的读写权限。
    // 9. 修改对内核空间不同 section 所在页属性的设置，完成对不同section的保护，其中text段
的权限为 r-x，rodata 段为 r--，其他段为 rw-，注意上述两个映射都需要做保护。
    // 10. 将必要的硬件地址（如 0x10000000 为起始地址的 UART ）进行等值映射 ( 可以映射连续
1MB 大小 )，无偏移，PTE_V | PTE_R 为映射的读写权限


    uint64_t physical_stack = alloc_page();
    uint64_t root_page_table = alloc_page();
    task[i]->sscratch = (uint64_t)0x1001000 + PAGE_SIZE;
    task[i]->satp = root_page_table >> 12 | 0x8000000000000000 | (((uint64_t)
(new_task->pid))  << 44);
    create_mapping((uint64_t*)root_page_table, 0x1001000, physical_stack,
PAGE_SIZE, PTE_V | PTE_R | PTE_W | PTE_U);
    create_mapping((uint64_t*)root_page_table, 0x1000000, task_addr, PAGE_SIZE,
PTE_V | PTE_R | PTE_X | PTE_U);

    // 调用 create_mapping 函数将虚拟地址 0xffffffc000000000 开始的 16 MB 空间映射到起
始物理地址为 0x80000000 的 16MB 空间
    create_mapping((uint64_t*)root_page_table, 0xffffffc000000000, 0x80000000,
16 * 1024 * 1024, PTE_V | PTE_R | PTE_W | PTE_X);
    // 修改对内核空间不同 section 所在页属性的设置，完成对不同section的保护，其中text段的权
限为 r-x，rodata 段为 r--，其他段为 rw-。
    create_mapping((uint64_t*)root_page_table, 0xffffffc000000000, 0x80000000,
PHYSICAL_ADDR((uint64_t)&rodata_start) - 0x80000000, PTE_V | PTE_R | PTE_X);
    create_mapping((uint64_t*)root_page_table, (uint64_t)&rodata_start,
PHYSICAL_ADDR((uint64_t)&rodata_start), (uint64_t)&data_start -
(uint64_t)&rodata_start, PTE_V | PTE_R);
```

```
    create_mapping((uint64_t*)root_page_table, (uint64_t)&data_start,
PHYSICAL_ADDR((uint64_t)&data_start), (uint64_t)&_end - (uint64_t)&data_start,
PTE_V | PTE_R | PTE_W);

    // 对内核起始地址 0x80000000 的16MB空间做等值映射（将虚拟地址 0x80000000 开始的 16
MB 空间映射到起始物理地址为 0x80000000 的 16MB 空间）
    create_mapping((uint64_t*)root_page_table, 0x80000000, 0x80000000, 16 * 1024
* 1024, PTE_V | PTE_R | PTE_W | PTE_X);
    // 修改对内核空间不同 section 所在页属性的设置，完成对不同section的保护，其中text段的权
限为 r-x, rodata 段为 r--, 其他段为 rw-。
    create_mapping((uint64_t*)root_page_table, 0x80000000, 0x80000000,
PHYSICAL_ADDR((uint64_t)&rodata_start) - 0x80000000, PTE_V | PTE_R | PTE_X);
    create_mapping((uint64_t*)root_page_table,
PHYSICAL_ADDR((uint64_t)&rodata_start), PHYSICAL_ADDR((uint64_t)&rodata_start),
(uint64_t)&data_start - (uint64_t)&rodata_start, PTE_V | PTE_R);
    create_mapping((uint64_t*)root_page_table,
PHYSICAL_ADDR((uint64_t)&data_start), PHYSICAL_ADDR((uint64_t)&data_start),
(uint64_t)&_end - (uint64_t)&data_start, PTE_V | PTE_R | PTE_W);

    // 将必要的硬件地址（如 0x10000000 为起始地址的 UART ）进行等值映射 ( 可以映射连续 1MB
大小 )，无偏移，3 为映射的读写权限
    create_mapping((uint64_t*)root_page_table, 0x10000000, 0x10000000, 1 * 1024
* 1024, PTE_V | PTE_R | PTE_W | PTE_X);




    printf("[PID = %d] Process Create Successfully!\n", task[i]->pid);
  }
  task_init_done = 1;
}
```

为了减少大家的代码量，大家不需要把task init时的用户地址空间添加进`mm`中，只需要分配一个空`vm_area_struct`作为链表头即可。

3.5.2 MMAP

```
void *mmap (void *__addr, size_t __len, int __prot,
                  int __flags, int __fd, __off_t __offset)
```

Linux 系统一般会提供一个 `mmap` 的系统调用，其作用是为当前进程的虚拟内存地址空间中分配新地址空间（从 `void *__addr` 开始，长度为 `size_t __len` 字节），创建和文件的映射关系（`__flags` 和 `__fd` 和 `__offset` 是文件相关的参数）。它可以分配一段匿名的虚拟内存区域，参数 `int __prot` 表示了对所映射虚拟内存区域的权限保护要求。

`mmap`实际上只是记录了用户进程申请了哪些虚拟地址空间，实际的物理空间分配是在使用的时候，再通过缺页异常来分配的。mmap 这样的设计可以保证 mmap 快速分配，并且按需分配物理内存给进程使用。

而创建和文件的映射关系也有诸多好处，例如通过 mmap 机制以后，磁盘上的文件仿佛直接就在内存中，把访问磁盘文件简化为按地址访问内存。这样一来，应用程序自然不需要使用文件系统的 write（写入）、read（读取）、fsync（同步）等系统调用，因为现在只要面向内存的虚拟空间进行开发。

因为 Linux 下一切皆为文件，除了管理磁盘文件，还可以用 mmap 简化 socket，设备等其他 IO 管理。

在我们的实验中，我们只需要实现`mmap`分配内存空间的功能，并且对参数做出简化：

1. 不考虑`__flags`、`__fd`和`__offset`参数
2. `__prot`参数与`PTE`的权限参数含义一致。

具体的函数定义及参数含义如下：

- `size_t`与`__off_t`为`unsigned long`的类型别名
- `__addr`:**建议**映射的虚拟首地址，需要按页对齐
- `__len`: 映射的长度，需要按页对齐
- `__prot`: 映射区的权限
- `__flags`: 由于本次实验不涉及`mmap`在Linux中的其他功能，该参数无意义
- `__fd`: 由于本次实验不涉及文件映射，该参数无意义
- `__offset`: 由于本次实验不涉及文件映射，该参数无意义
- 返回值：**实际**映射的虚拟首地址，若虚拟地址区域`[__addr，__addr+__len)`未与其它地址映射冲突，则返回值就是建议首地址`__addr`，若发生冲突，则需要更换到无冲突的虚拟地址区域，返回值为该区域的首地址。
  注意：为简化实验，你不需要考虑冲突的情况，只需要严格按照 **addr,** len 进行分配即可。

`mmap`映射的虚拟地址区域采用需求分页（Demand Paging）机制，即：`mmap`中不为该虚拟地址区域分配实际的物理内存，仅当进程试图访问该区域时，通过触发 Page Fault，由 Page Fault 处理函数进行相应的物理内存分配以及页表的映射。

该函数的操作步骤如下：

- 利用 kmalloc 创建一个`vm_area_struct`记录该虚拟地址区域的信息，并添加到`mm`的链表中
- - 调用`kmalloc`分配一个新的`vm_area_struct`
  - 设置`vm_area_struct`的起始和结束地址为`start`, `start + length`
  - 设置`vm_area_struct`的`vm_flags`为`prot`

    - 将该 `vm_area_struct` 插入到 `vm` 的链表中
- 返回地址区域的首地址

这样，使用 mmap 函数就可以为当前进程注册一段虚拟内存了，这段申请的相关信息存在本进程的 `task_struct` 的 `vm_area_struct` 中。

将你编写好的 mmap 系统调用复制到下面代码框内：

```
// syscall.c

case SYS_MMAP: {
      // TODO: implement mmap
      // 1. create a new vma struct (kmalloc), if kmalloc failed, return -1
      // 2. initialize the vma struct
      // 2.1 set the vm_start and vm_end according to arg0 and arg1
      // 2.2 set the vm_flags to arg2
      // 2.3 set the mapped flag to 0
      // 3. add the vma struct to the mm_struct's vma list
      // return the vm_start

      struct vm_area_struct *vm = (struct vm_area_struct *)kmalloc(sizeof(struct
vm_area_struct));
        if (vm == NULL) {
            ret.a0 = -1;
            break;
        } else{
            vm->vm_start = arg0;
            vm->vm_end = arg0 + arg1;
            vm->vm_flags = arg2;
            vm->mapped = 0;
            list_add(&(vm->vm_list), &(current->mm.vm->vm_list));
            ret.a0 = vm->vm_start;
        }

        break;
```

> 提示：使用 `list_add(struct list_head* node1, struct list_head* node2)` 可以将node1插入到node2后

3.5.3 MUNMAP

与 `mmap` 相对的是 `munmap` 系统调用，格式如下：

```
int munmap(void *addr, size_t len);
```

`mnumap`将会释放任何与`[addr,addr+len]`有交集的页面映射，如果成功释放返回0，否则返回-1。在我们的实现中，我们要求`[addr, addr+len]`必须是一个完整的内存块，否则释放失败。

将你编写好的 munmap 系统调用复制到下面代码框内：

```
  case SYS_MUNMAP: {
      // TODO: implement munmap
      // 1. find the vma according to arg0 and arg1
      // note: you can iterate the vm_list by list_for_each_entry(vma, &current->mm.vm->vm_list, vm_list), then you can use `vma` in your loop
      // 2. if the vma mapped, free the physical pages (free_pages). Using `get_pte` to get the corresponding pte.
      // 3. ummap the physical pages from the virtual address space (create_mapping)
      // 4. delete the vma from the mm_struct's vma list (list_del).
      // 5. free the vma struct (kfree).
      // return 0 if success, otherwise return -1


      struct vm_area_struct *vm;
        list_for_each_entry(vm, &current->mm.vm->vm_list, vm_list) {
            if(vm->vm_start == arg0 && vm->vm_end == arg0 + arg1) {
                if(vm->mapped == 1) {
                    uint64_t pte = get_pte((current->satp & ((1ULL << 44) - 1)) << 12, vm->vm_start);
                    free_pages((pte >> 10) << 12);
                }
                create_mapping((current->satp & ((1ULL << 44) - 1)) << 12, vm->vm_start, 0, (vm->vm_end - vm->vm_start), 0);
                list_del(&(vm->vm_list));
                kfree(vm);
                ret.a0 = 0;
                break;
            }else{
                ret.a0 = -1;
            }
        }

        // flash the TLB
        asm volatile ("sfence.vma");
        break;


  }
```

> 提示：
>
> 1. 你可以通过`list_for_each_entry(vma, &current->mm.vm->vm_list, vm_list)`来遍历当前进程的vm_area链表，然后在循环体里使用`vma`访问每个vm_area_struct。注意提前定义vma。

> 2. 我们在`vm.c`中实现了`uint64_t get_pte(uint64_t *pgtbl, uint64_t va)`，来获取以pgtbl为根页表地址的虚拟地址va对应的页表项。

3.6 实现用户函数库

我们已经在`user/lib/src/mm.c`中实现了`mmap`和`munmap`两个库函数。

3.7 缺页异常处理 （30%）

现在，每个进程分配过的虚拟内存地址空间可以使用`current->vm`来访问，在前面我们讲过，我们的进程是需求分页机制，直到进程使用该虚拟内存地址的时候，我们才会分配物理内存给他。

也就是说，进程使用该虚拟内存地址的时候会发生缺页异常，内核在缺页异常的处理函数中，通过检查`current->vm`数据结构，检查缺页异常的地址落在什么范围内。最后根据对应范围的相关信息做物理内存映射即可。

3.7.1 缺页异常

缺页异常是一种正在运行的程序访问当前未由内存管理单元（MMU）映射到虚拟内存的页面时，由计算机硬件引发的异常类型。访问未被映射的页或访问权限不足，都会导致该类异常的发生。处理缺页异常通常是操作系统内核的一部分。当处理缺页异常时，操作系统将尝试使所需页面在物理内存中的位置变得可访问（建立新的映射关系到虚拟内存）。如果非法访问内存，即发现触发 Page Fault 的虚拟内存地址(Bad Address)不在当前进程`vm_area_struct`链表中所定义的允许访问的虚拟内存地址范围内，或访问位置的权限条件不满足时，缺页异常处理将终止该程序的继续运行。

当系统运行发生异常时，可即时地通过解析 scause 寄存器的值，识别如下三种不同的 Page Fault。

Interrupt/Exception scause[XLEN-1]	**Exception Code scause[XLEN-2:0]**	**Description**
0	12	Instruction Page Fault
0	13	Load Page Fault
0	15	Store/AMO Page Fault

在lab3中，我们已经把缺页异常委托给了 S 态，当时如果触发缺页异常，我们会直接进入死循环。现在我们按照如下逻辑实现缺页异常处理程序：

1. 首先读取`stval`的值，获取触发缺页异常的内存地址
2. 遍历vm_area，如果该地址落在某个area内部
3. 1. 如果权限符合要求，则为该area分配物理空间，并建立映射。注意不同的缺页异常要求的页面权限不同，如下图所示。具体请参看Sv32标准（Sv39与Sv32的权限位含义相同）
   2. 如果不符合权限要求，则打印错误信息，并把`sepc + 4`跳过错误指令。建议打印出需要什么权限，而vm_area实际是什么权限。
4. 如果没有找到符合条件的vm_area，则报错并跳过错误指令（Done）

将你实现的Page fault处理程序复制在下面代码框内：

```
// instruction page fault
if (cause == 0xc || cause == 0xd || cause == 0xf) {
  uint64_t stval;
  uint64_t* sp_ptr = (uint64_t*)(sp);

  // TODO:
  // 1. get the faulting address from stval register
  asm volatile("csrr %0, stval" : "=r"(stval));


  printf("Page fault! epc = 0x%016lx, stval = 0x%016lx\n", epc, stval);


  struct vm_area_struct* vma;
  list_for_each_entry(vma, &current->mm.vm->vm_list, vm_list) {
    // TODO:
    // 2. check whether the faulting address is in the range of a vm area
    // 3. check the permission of the vm area. The vma must be PTE_X/R/W
according to the faulting cause, and also be PTE_V, PTE_U
    // 4. if the faulting address is valid, allocate physical pages, map it to
the vm area, mark the vma mapped(vma->mapped = 1), and return
    // 5. otherwise, print error message and add 4 to the sepc
    // 6. if the faulting address is not in the range of any vm area, add 4 to
the sepc (DONE)
    if(stval >= vma->vm_start && stval <= vma->vm_end) {
          if((vma->vm_flags & PTE_V) && (vma->vm_flags & PTE_U) && (((vma-
>vm_flags & PTE_X) && cause == 0xc) || ((vma->vm_flags & PTE_R) && cause == 0xd)
|| ((vma->vm_flags & PTE_R) && (vma->vm_flags & PTE_W) && cause == 0xf))) {
            uint64_t phisical_address = alloc_pages((vma->vm_end - vma-
>vm_start) / PAGE_SIZE);
            if (phisical_address == 0) {
              printf("alloc_pages failed!\n");
              sp_ptr[16] += 4;
              return;
            }
            create_mapping((current->satp & ((1ULL << 44) - 1)) << 12, vma-
>vm_start, phisical_address, (vma->vm_end - vma->vm_start), vma->vm_flags);
            vma->mapped = 1;
            return;
          }else {
```

```
            printf("Page Faulting! scause: %llx flags: %llx \n", cause, vma->vm_flags);
            sp_ptr[16] += 4;
            return;
          }
        }


  }
  sp_ptr[16] += 4;
  return;
}
```

3.8 编译和测试

我们对test程序做了修改，其中test1为mmap正常测例，test2为munmap正常测例，test3与test4测试权限控制测例，test5为munmap异常测例。请确保你的每个测例都通过。

对 `main.c` 做修改，确保输出你的学号与姓名。在项目最外层输入 `make run` 命令调用 Makefile 文件完成整个工程的编译及执行。

如果编译失败，及时使用 `make clean` 命令清理文件后再重新编译。

请在此附上你的实验结果截图。

```
task[3]: counter = 1, priority = 4
task[4]: counter = 1, priority = 5
checking in test1
Page fault! epc = 0x000000000100004c, stval = 0x0000000000000000
[CHECK] page store OK

Page fault! epc = 0x00000000010000b0, stval = 0x0000000000001000
[CHECK] page load OK

[*PID = 0] Context Calculation: counter = 1,priority = 1
task[0]: counter = 0, priority = 1
task[1]: counter = 1, priority = 2 <-- next
task[2]: counter = 1, priority = 3
task[3]: counter = 1, priority = 4
task[4]: counter = 1, priority = 5
checking in test2
Page fault! epc = 0x0000000001000044, stval = 0x0000000000000000
Page Faulting! scause: 000000000000000f flags: 000000000000001b
Page fault! epc = 0x0000000001000074, stval = 0x0000000000008000
Page fault! epc = 0x00000000010000e8, stval = 0x0000000000000000
Unhandled page fault! addr = 0x0000000000000000
Page fault! epc = 0x0000000001000100, stval = 0x0000000000008000
Unhandled page fault! addr = 0x0000000000008000
[CHECK] unmap OK

[*PID = 1] Context Calculation: counter = 1,priority = 2
task[0]: counter = 0, priority = 1
task[1]: counter = 0, priority = 2
task[2]: counter = 1, priority = 3 <-- next
task[3]: counter = 1, priority = 4
task[4]: counter = 1, priority = 5
checking in test3
Page fault! epc = 0x0000000001000044, stval = 0x0000000000000000
Page fault! epc = 0x0000000001000078, stval = 0x0000000000000000
Page Faulting! scause: 000000000000000d flags: 000000000000001d
[CHECK] page access control OK

[*PID = 2] Context Calculation: counter = 1,priority = 3
task[0]: counter = 0, priority = 1
task[1]: counter = 0, priority = 2
task[2]: counter = 0, priority = 3
task[3]: counter = 1, priority = 4 <-- next
task[4]: counter = 1, priority = 5
checking in test4
Page fault! epc = 0x0000000001000048, stval = 0x0000000000001000
Page Faulting! scause: 000000000000000f flags: 000000000000001d
Page fault! epc = 0x0000000001000078, stval = 0x0000000000001000
[CHECK] page access control OK

[*PID = 3] Context Calculation: counter = 1,priority = 4
task[0]: counter = 0, priority = 1
task[1]: counter = 0, priority = 2
task[2]: counter = 0, priority = 3
task[3]: counter = 0, priority = 4
task[4]: counter = 1, priority = 5 <-- next
checking in test5
[CHECK] unmap OK
```