Lab 3

操作系统课程实验报告

实验名称：Lab3 缺页异常和页面置换

小组成员： 2113388 高涵 2112849唐静蕾 2113999 陈翊炀

一、实验目的

本次实验是在实验二的基础上，借助于页表机制和实验一中涉及的中断异常处理机制，完成Page Fault异常处理和部分页面替换算法的实现，结合磁盘提供的缓存空间，从而能够支持虚存管理，提供一个比实际物理内存空间“更大”的虚拟内存空间给系统使用。这个实验与实际操作系统中的实现比较起来要简单，不过需要了解实验一和实验二的具体实现。实际操作系统系统中的虚拟内存管理设计与实现是相当复杂的，涉及到与进程管理系统、文件系统等的交叉访问。主要实现以下内容：

- 了解虚拟内存的Page Fault异常处理实现
- 了解页替换算法在操作系统中的实现
- 学会如何使用多级页表，处理缺页异常（Page Fault），实现页面置换算法。

二、实验内容

练习1：理解基于FIFO的页面替换算法

> 描述FIFO页面置换算法下，一个页面从被换入到被换出的过程中，会经过代码里哪些函数/宏的处理（或者说，需要调用哪些函数/宏），并用简单的一两句话描述每个函数在过程中做了什么？至少正确指出10个不同的函数分别做了什么？如果少于10个将酌情给分。我们认为只要函数原型不同，就算两个不同的函数。要求指出对执行过程有实际影响,删去后会导致输出结果不同的函数.

1.触发缺页异常 `do_pgfault`：

```
1 int do_pgfault(struct mm_struct *mm, uint_t error_code, uintptr_t addr)
```

这个函数传了三个参数，首先是mm_struct结构体，包含了 list_entry_t mmap_list双向链表;struct vma_struct *mmap_cache最近访问过的vma;pde_t *pgdir页目录入口地址;int map_count双向链表

中vma数目; void *sm_priv之后替换策略要用到的参数; error_code是错误类型，addr是产生缺页异常时对应异常的地址；主要实现了通过调用之后提到的函数对缺页异常发生时的页面操作处理。

2.查找触发缺页异常的地址信息在哪块vma中 `find_vma`：

```
1 find_vma(struct mm_struct *mm, uintptr_t addr)
```

这个函数通过mm结构体和addr地址，进行其与vma != NULL && vma->vm_start <= addr && vma->vm_end > addr的比较等操作，找到触发缺页异常的地址信息对应的vma。

3.查找或创建一个pte项 `get_pte`：

```
1 pte_t *get_pte(pde_t *pgdir, uintptr_t la, bool create)
```

这个函数通过两个参数页表基址pgdir和虚拟地址la来寻找并分配页表项，由于传入的bool参数create是1，所以会在查询失败时主动创建页表项，并将查询结果放入ptep中，最终实现效果是查找到的页表项有效则返回，无效则通过create创建一个新的页表项。

4.空页表项结果分配页 `pgdir_alloc_page`：

```
1 struct Page *pgdir_alloc_page(pde_t *pgdir, uintptr_t la, uint32_t perm)
```

这个函数同样通过两个参数页表基址pgdir和虚拟地址la来分配页，此处的第三个参数perm指的是此时的用户态（U）。在这个函数中会调用之前lab2提到的 `alloc_page()` 函数，不过多介绍。

5.创建或维护页表项到页的映射 `page_insert`：

```
1 int page_insert(pde_t *pgdir, struct Page *page, uintptr_t la, uint32_t perm)
```

这个函数参数为页表基址pgdir，新建的页指针page，虚拟地址la以及状态perm，具体实现效果为先找到对应页表项的位置（如果原先不存在，前面提到的函数get_pte()会分配页表项的内存），之后根据具体情况维护page_ref以及页表项到页的具体映射关系。此处的一个小函数 `pte2page` 与lab2中的对应函数类似，实现了指针由pte转向page的转变，不再赘述，维护时产生的 `page_ref_inc(page)` 、 `page_ref_dec(page)` 函数也都是简单的对页面引用的加减。

之后经过缺页异常中断触发，到找到中断对应位置vma，对pte和page进行创建、查找或者维护等一系列操作之后就涉及到内存和“磁盘”交互的换页实现了。

6.磁盘的页读入到内存page中的 `swap_in`：

```
int swap_in(struct mm_struct *mm, uintptr_t addr, struct Page **ptr_result)
```

mm结构体、addr虚拟地址、传入指向指针的指针ptr_result，为了使数据能够写回到page*数据类型中，这个函数主要操作是首先找到一个空闲页，此处调用了 `alloc_page()` 函数（由于可能没有空闲页，所以递归调用了 `swap_out()` 函数可能将一个页面换出的操作，之后会具体解释），通过页基址和虚拟地址pgdir, addr找到要写入的对应页表项，调用 `swapfs_read()` 函数，传入页表项的值和需要写入的页，实现磁盘数据读取到内存。

7.从磁盘中读取数据到内存页面 `swapfs_read`：

```
int swapfs_read(swap_entry_t entry, struct Page *page) {
    return ide_read_secs(SWAP_DEV_NO, swap_offset(entry) * PAGE_NSECT, page2kva(
                              PAGE_NSECT);
}
```

entry是入口地址，page指向页面结构的指针，表示要将数据加载到其中的页面，这个函数的目的是从交换设备读取特定页的数据，并将其加载到系统的页面中，在这个函数中递归调用了 `ide_read_secs()` 函数：

8.从指定IDE设备读取扇区数据 `ide_read_secs()`：

```
int ide_read_secs(unsigned short ideno, uint32_t secno, void *dst,
                  size_t nsecs) {

    int iobase = secno * SECTSIZE;
    memcpy(dst, &ide[iobase], nsecs * SECTSIZE);
    return 0;
```

参数分别为接受设备号ideno、起始扇区号secno、目标缓冲区指针dst，以及要读取的扇区数nsecs，这个函数会将扇区数据复制到目标缓冲区，并返回一个整数值，用于表示操作是否成功，其中0表示成功。

9.将页面换出到磁盘 `swap_out()`：

```
int swap_out(struct mm_struct *mm, int n, int in_tick)
```

这个函数接受三个参数：mm表示进程的内存映射结构体，n表示要换出的页面数量，int in_tick是一个传入时就固定的参数，固定为1，表示一次换出一页。

在循环中，首先定义了变量v和page，然后调用了 `sm->swap_out_victim` 函数，该函数是页面置换算法机制与策略的体现，用于选择需要换出的页面（victim）。如果返回值不为0，则表示选择页面失败，打印错误信息并中断循环。如果成功选择了要换出的页面，则会打印出被选择的页面的虚拟地址，然后根据该虚拟地址获取对应的页表项ptep。代码接着通过 `swapfs_write` 函数将页面写入硬盘的交换区。如果写入失败，会打印错误信息，并将这个页面用 `swap_map_swappable()` 函数重新标记为可交换的。如果写入成功，则会打印成功换出的信息，并将失效的页表项更新为指向交换区的地址，然后释放页面。最后，由于页表项发生了改变，使用 `tlb_invalidate` 函数来刷新TLB。

10.页面写入硬盘的交换区 `swapfs_write`：

```
int
swapfs_write(swap_entry_t entry, struct Page *page) {
    return ide_write_secs(SWAP_DEV_NO, swap_offset(entry) * PAGE_NSECT, page2kva
}
```

函数接受两个参数：swap_entry_t entry表示交换区的入口，struct Page *page表示要写入交换区的页面。

递归调用 `ide_write_secs()` 函数，与 `ide_read_secs()` 函数实现效果类似，SWAP_DEV_NO是交换区所在设备的编号，swap_offset(entry) * PAGE_NSECT用于计算要写入的扇区偏移量，page2kva(page)用于获取页面对应的内核虚拟地址，PAGE_NSECT表示每页的扇区数量。返回一个整数值，返回0表示写入成功。

至此，从触发缺页异常之后经过的页面换入换出函数大致介绍完成。

11.还存在其他函数：

比如从页表中移除特定的页表项的函数 `page_remove_pte` ，该函数接受三个参数：pgdir表示页表的基地址，la表示线性地址，ptep表示要移除的页表项；刷新TLB的 `tlb_invalidate()` 函数；调用了函数 `page_remove_pte` 的 `page_remove()` 函数；对应的检查函数 `check_alloc_page(void)` ， `check_pgdir(void)` ， `check_boot_pgdir(void)` 等函数，都在页面换入换出操作中发挥了特定作用。

练习2：深入理解不同分页模式的工作原理

> get_pte()函数（位于 `kern/mm/pmm.c` ）用于在页表中查找或创建页表项，从而实现对指定线性地址对应的物理页的访问和映射操作。这在操作系统中的分页机制下，是实现虚拟内存与物理内存之间映射关系非常重要的内容。get_pte()函数中有两段形式类似的代码， 结合sv32，sv39，sv48的异

> 同，解释这两段代码为什么如此相像。目前get_pte()函数将页表项的查找和页表项的分配合并在一个函数里，你认为这种写法好吗？有没有必要把两个功能拆开？

sv32、sv39和sv48是RISC-V处理器的不同的虚拟内存模式，指的是页表的级数和每级页表项的位数。

sv32模式：页表级数为两级，包括页目录表（PDT）和页表（PT）。PDT和PT的页表项都是32位

sv39模式：页表级数为三级，在实验指导书中我们把页表项里从高到低三级页表的页码分别称作PDX1, PDX0和PTX(Page Table Index)，sv39 里面的一个页表项大小为 64 位，在sv39中页的大小为4KiB=4096字节，大页的大小为2MiB= 2^21 字节，大大页的大小为1 GiB 。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。

sv48模式：页表级数为四级，包括页全局目录表（PGDT）、页目录表（PDT）和两级页表（PT1和PT2），PGDT的页表项为64位，PDT和PT1的页表项为48位，PT2的页表项为40位。（**sv48页表和sv39兼容**）

在其他模式（sv32和sv48模式）下需修改分级查找的级数即可，三个模式只是在页表级数和偏移上有区别，逻辑上的查找过程是类似的，都是逐级查找的过程。

```
pde_t *pdep1 = &pgdir[PDX1(la)];//找到对应的Giga Page
    if (!(*pdep1 & PTE_V)) {//如果下一级页表不存在，那就给它分配一页，创造新页表
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL) {
            return NULL;
        }
        set_page_ref(page, 1);
        uintptr_t pa = page2pa(page);
        memset(KADDR(pa), 0, PGSIZE);
        //我们现在在虚拟地址空间中，所以要转化为KADDR再memset初始化.
        //不管页表怎么构造，我们确保物理地址和虚拟地址的偏移量始终相同，那么就可以用这种方
        *pdep1 = pte_create(page2ppn(page), PTE_U | PTE_V);//注意这里R,W,X全零
    }
    pde_t *pdep0 = &((pde_t *)KADDR(PDE_ADDR(*pdep1)))[PDX0(la)];//再下一级页表
    //这里的逻辑和前面完全一致，页表不存在就现在分配一个
    if (!(*pdep0 & PTE_V)) {
        struct Page *page;
        if (!create || (page = alloc_page()) == NULL) {
                return NULL;
        }
        set_page_ref(page, 1);
        uintptr_t pa = page2pa(page);
        memset(KADDR(pa), 0, PGSIZE);
```

```
24          *pdep0 = pte_create(page2ppn(page), PTE_U | PTE_V);
25      }
26      //找到输入的虚拟地址la对应的页表项的地址(可能是刚刚分配的)
27      return &((pte_t *)KADDR(PDE_ADDR(*pdep0)))[PTX(la)];
```

这两段代码相似之处在于它们都是用来查找特定虚拟地址所对应的页表项的地址。它们的不同之处在于对于不同的页表级别，需要使用不同的位移和掩码操作来计算页表项的地址，两段代码的逻辑是一样的：如果下一级页表不存在，那就给它分配一页，创造新页表。下一级页表存在，就去计算获得下一级页表的地址。

- 对于sv39，页表级别为三级，所以要查找页表项需要经过两次查找，即为此函数中两段形式类似的代码，进行分级查找。第一段代码中的 `pdep1` 是先找到对应的 `Giga Page` ，若下级页表存在那就找到对应的二级页表，第二段代码中的 `pdep0` 是再下级页表项的地址，即我们需要查找的页表项。

对于get_pte()函数将页表项的查找和页表项的分配合并在一个函数的写法，这样做的好处是可以提高代码的复用性和可读性。将页表项的查找和页表项的分配合并在一个函数中，可以使代码更加简洁和易于理解。在使用这个函数时，不需要单独调用两个不同的函数来查找和分配页表项，而是可以在一个函数中完成这两个操作。这样的设计可以减少代码的重复，并且使代码逻辑更加紧凑和一致，将它们合并在一起可以更方便地进行相关的错误处理和状态管理。例如，在分配页表项时，如果分配失败，可以立即返回错误，而不需要在两个单独的函数之间传递错误状态。

具体是否要将两个功能分开要取决于具体的代码需求和设计。如果在某些情况下需要单独进行页表项的查找或页表项的分配，并且这两个操作的逻辑差异较大，那么拆分为单独的函数可能更合适。

练习3：给未被映射的地址映射上物理页

> 补充完成do_pgfault（mm/vmm.c）函数，给未被映射的地址映射上物理页。设置访问权限的时候需要参考页面所在 VMA 的权限，同时需要注意映射物理页时需要操作内存控制结构所指定的页表，而不是内核的页表。请描述页目录项（Page Directory Entry）和页表项（Page Table Entry）中组成部分对ucore实现页替换算法的潜在用处。如果ucore的缺页服务例程在执行过程中访问内存，出现了页访问异常，请问硬件要做哪些事情?

do_pgfault 函数：

```
1  int
2  do_pgfault(struct mm_struct *mm, uint_t error_code, uintptr_t addr) {
3      int ret = -E_INVAL;
4      //try to find a vma which include addr
5      struct vma_struct *vma = find_vma(mm, addr);
6
7      pgfault_num++;
8      //If the addr is in the range of a mm's vma?
```

```
    if (vma == NULL || vma->vm_start > addr) {
        cprintf("not valid addr %x, and  can not find it in vma\n", addr);
        goto failed;
    }

    uint32_t perm = PTE_U;
    if (vma->vm_flags & VM_WRITE) {
        perm |= (PTE_R | PTE_W);
    }
    addr = ROUNDDOWN(addr, PGSIZE);

    ret = -E_NO_MEM;

    pte_t *ptep=NULL;



    ptep = get_pte(mm->pgdir, addr, 1);  //(1) try to find a pte, if pte's
                                         //PT(Page Table) isn't existed, then
                                         //create a PT.
    if (*ptep == 0) {
        if (pgdir_alloc_page(mm->pgdir, addr, perm) == NULL) {
            cprintf("pgdir_alloc_page in do_pgfault failed\n");
            goto failed;
        }
    } else {

        if (swap_init_ok) {
            struct Page *page = NULL;
            // 你要编写的内容在这里，请基于上文说明以及下文的英文注释完成代码编写
            //(1) According to the mm AND addr, try
            //to load the content of right disk page
            //into the memory which page managed.
            //(2) According to the mm,
            //addr AND page, setup the
            //map of phy addr <--->
            //logical addr
            //(3) make the page swappable.

            // (1) 根据mm和addr，尝试将正确的磁盘页内容加载到由page管理的内存页中
            if (swap_in(mm, addr, &page) != 0) {
                cprintf("swap_in 在do_pgfault中失败\n");
                goto failed;
            }

            // (2) 根据mm、addr和page，建立物理地址和逻辑地址之间的映射
            if (page_insert(mm->pgdir, page, addr, perm) != 0) {
```

```
56                 cprintf("page_insert 在do_pgfault中失败\n");
57                 goto failed;
58             }
59 
60             // (3) 使页面可交换
61             swap_map_swappable(mm, addr, page, 1);
62 
63 
64             page->pra_vaddr = addr;
65         } else {
66             cprintf("no swap_init_ok but ptep is %x, failed\n", *ptep);
67             goto failed;
68         }
69    }
70 
71    ret = 0;
72 failed:
73     return ret;
74 }
75 
```

潜在用处：ucore操作系统中的页目录项（PDE）和页表项（PTE）的各个字段具有潜在用处，特别是在页替换算法的实现和缺页服务例程的执行过程中。这些字段可用于标记页面是否存在于物理内存中，页面是否可读写，以及页面的访问和修改情况。在页替换算法中，它们可以协助选择要被换出的页面，优化内存管理。在缺页服务例程中，这些字段用于处理页访问异常，如检查页面有效性、权限和执行必要的页面加载和页表更新操作。

出现页访问异常:如果ucore的缺页服务例程在执行过程中访问内存，出现了页访问异常，硬件将触发异常处理，通常被称为页访问异常或页错误。此时，操作系统内核会捕获该异常，并执行相应的中断处理例程，即缺页服务例程。这个服务例程将尝试从磁盘或其他存储设备中加载缺失的页面，然后更新页表，使其指向新加载的页面。一旦缺页服务例程成功加载页面并更新页表，控制将返回到原始引发异常的指令，允许程序继续执行。如果页面加载和页表更新失败，可能会再次引发异常，直到所需的页面完全加载，从而保证程序的正常执行。硬件在此过程中的作用是引发异常、提供必要的信息，并与操作系统协同工作，以确保缺页服务例程能够有效管理内存和处理页面访问异常。

练习4：补充完成Clock页替换算法

> 在我们给出的框架上，填写代码，实现 Clock页替换算法（mm/swap_clock.c）并且请比较Clock页替换算法和FIFO算法的不同。

_clock_init_mm函数：

```
1 static int
```

```
_clock_init_mm(struct mm_struct *mm)
{
     /*LAB3 EXERCISE 4: YOUR CODE*/
     // 初始化pra_list_head为空链表
     list_init(&pra_list_head);
     // 初始化当前指针curr_ptr指向pra_list_head，表示当前页面替换位置为链表头
     curr_ptr=&pra_list_head;
     // 将mm的私有成员指针指向pra_list_head，用于后续的页面替换算法操作
     mm->sm_priv = &pra_list_head;
     //cprintf(" mm->sm_priv %x in fifo_init_mm\n",mm->sm_priv);
     return 0;
}
```

_clock_map_swappable函数：

```
static int
_clock_map_swappable(struct mm_struct *mm, uintptr_t addr, struct Page *page, in
{
    list_entry_t *entry=&(page->pra_page_link);
    list_entry_t *head=(list_entry_t*) mm->sm_priv;

    assert(entry != NULL && curr_ptr != NULL);
    //record the page access situlation
    /*LAB3 EXERCISE 4: YOUR CODE*/
    // link the most recent arrival page at the back of the pra_list_head qeueue
    // 将页面page插入到页面链表pra_list_head的末尾
    list_add(head, entry);
    // 将页面的visited标志置为1，表示该页面已被访问
    page->visited=1;
    return 0;
}
```

_clock_swap_out_victim函数：

```
static int
_clock_swap_out_victim(struct mm_struct *mm, struct Page ** ptr_page, int in_tic
{
     list_entry_t *head=(list_entry_t*) mm->sm_priv;
         assert(head != NULL);
     assert(in_tick==0);
```

```
     /* Select the victim */
     //(1)  unlink the  earliest arrival page in front of pra_list_head qeueue
     //(2)  set the addr of addr of this page to ptr_page
    struct Page* p = NULL;
    //while (1) {
        /*LAB3 EXERCISE 4: YOUR CODE*/
        // 编写代码
        // 遍历页面链表pra_list_head，查找最早未被访问的页面
        // 获取当前页面对应的Page结构指针
        // 如果当前页面未被访问，则将该页面从页面链表中删除，并将该页面指针赋值给ptr_page
        // 如果当前页面已被访问，则将visited标志置为0，表示该页面已被重新访问
    //}
        for(; ;curr_ptr = list_prev(curr_ptr)){

            if(curr_ptr == head){
                continue;
            }
            cprintf("curr_ptr %p\n", curr_ptr);
            p = le2page(curr_ptr, pra_page_link);
            if(p->visited == 1){// 如果当前页面已被访问
              p->visited = 0;
            }else if(p->visited == 0){// 如果当前页面未被访问
                list_del(curr_ptr);
                curr_ptr = list_prev(curr_ptr);
                *ptr_page = p;
                return 0;
            }
        }

    *ptr_page = NULL;
    return 0;
}
```

Clock页替换算法和FIFO算法的不同：

- FIFO算法：该算法总是淘汰最先进入内存的页，即选择在内存中驻留时间最久的页予以淘汰。
- Clock算法：时钟页替换算法把各个页面组织成环形链表的形式，把当前指针指向最先进来的那个页面，并且在页表项（PTE）中设置了一位访问位（visited）来表示此页表项对应的页当前是否被访问过，被访问时，该标志位置位为1，否则为0。当操作系统需要淘汰页时，对当前指针指向的页所对应的页表项进行查询，如果访问位为“0”，则淘汰该页，如果该页被写过，则还要把它换出到硬盘上；如果访问位为“1”，则将该页表项的此位置“0”，继续访问下一个页。即时钟页替换算法中跳过了访问位为 1 的页。

时钟页替换算法在本质上与 FIFO 算法是类似的，不同之处是在时钟页替换算法中跳过了访问位为 1 的页。

练习5：阅读代码和实现手册，理解页表映射方式相关知识

> 如果我们采用”一个大页“的页表映射方式，相比分级页表，有什么好处、优势，有什么坏处、风险?

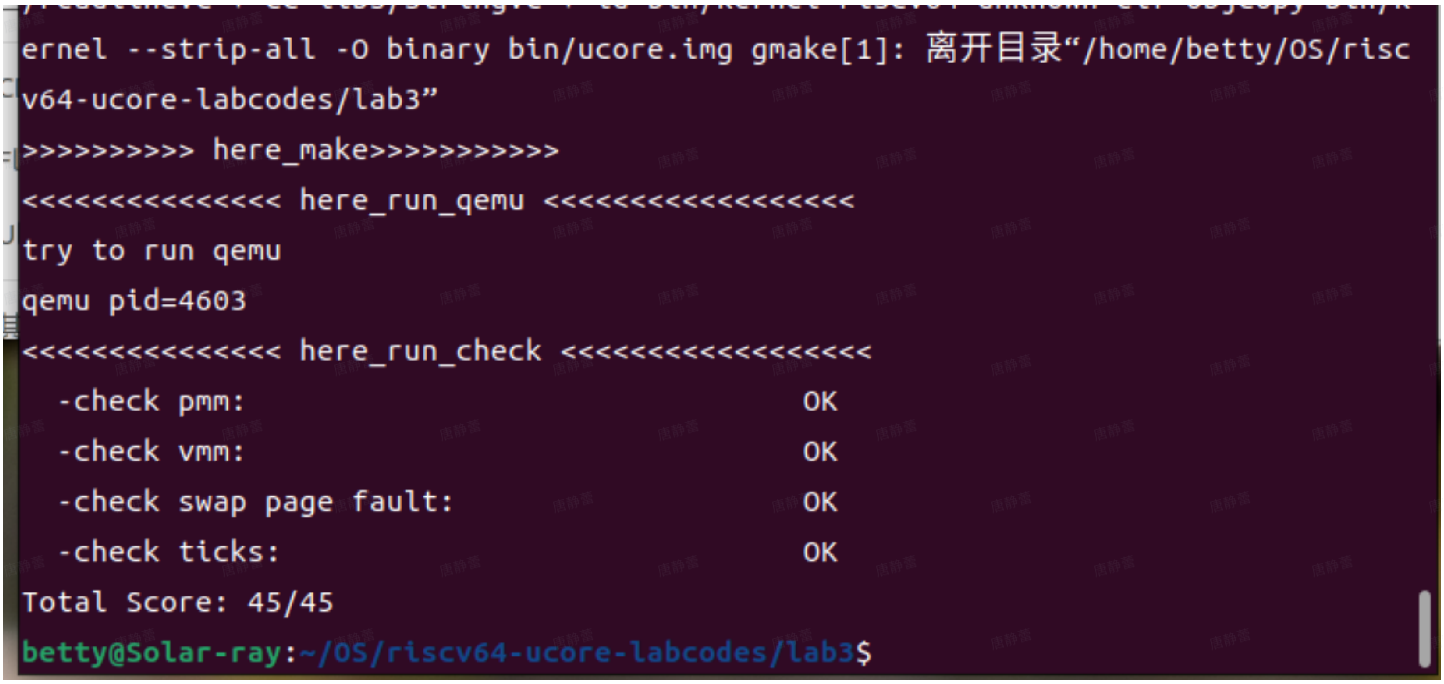
优势：

1. **减少内存开销：** 使用大页表可以减少内存中页表所占用的空间。因为大页表可以存储更多的虚拟地址范围，所以相对于分级页表来说，大页表可以减少存储开销。
2. **减少TLB失效：** 使用大页表可以减少TLB失效的概率，因为更多的虚拟地址可以映射到一个大页中，这意味着TLB可以缓存更多的页表项，从而减少TLB缺失带来的性能损失。
3. **提高访问效率：** 由于大页表可以提供更多连续的物理地址空间，因此可以减少访问页表的次数和寻址次数，提高内存访问的效率。
4. **实现简单：**相比于多级页表来说，一级页表的实现更简单
5. **易于维护：**相比于多级页表来说，大页表的维护更加简易

劣势和风险：

1. **内存碎片问题：** 大页表可能会导致内存碎片问题。当进程只需要一小部分大页中的内容时，可能会造成内存浪费和碎片化，因为系统必须保留整个大页以满足进程的需求。
2. **内存利用率低：** 大页表可能会导致内存分配不够灵活，特别是当内存需求不断变化时。如果进程需要的内存空间不足一个大页的大小，那么可能会浪费一些内存空间，从而影响系统的内存利用率。
3. **启动时间延长：** 由于大页表需要更多的连续内存空间来存储，因此在系统启动时，可能需要更长的时间来为大页表分配足够的内存空间。
4. **TLB缓存频繁缺失：**当一级页表较大时，由于TLB缓存大小有限，虚拟内存太大时无法缓存最近使用的所有页表项，导致命中概率降低，TLB查询转换速率降低。

通过 `Make grade` 测试：


ernel --strip-all -O binary bin/ucore.img gmake[1]: 离开目录“/home/betty/OS/risc
v64-ucore-labcodes/lab3”
>>>>>>>>>> here_make>>>>>>>>>>
<<<<<<<<<<<<<< here_run_qemu <<<<<<<<<<<<<<<<<<
try to run qemu
qemu pid=4603
<<<<<<<<<<<<<< here_run_check <<<<<<<<<<<<<<<<<<
-check pmm: OK
-check vmm: OK
-check swap page fault: OK
-check ticks: OK
Total Score: 45/45
betty@Solar-ray:~/OS/riscv64-ucore-labcodes/lab3$