杜佳男 21210110

操作系统lab2实验报告

1、实验思考题

Thinking 2.1

- **在我们编写的程序中，指针变量中存储的地址是虚拟地址还是物理地址?**

  答：虚拟地址。

- **MIPS 汇编程序中lw, sw使用的是虚拟地址还是物理地址?**

  答：虚拟地址。

Thinking 2.2

- **请从可重用性的角度，阐述用宏来实现链表的好处。**

  答：使用宏来实现链表将部分功能封装，具有很高的可重用性，以宏`LIST_NEXT`为例，在其他宏的实现中会被反复调用，在实际操作过程中也经常会用到这一个宏，无论是在链表中插入结构还是删除结构，都可以通过调用这一个宏来简化代码；而对于其他宏如`LIST_INSERT_HEAD`等，在对内存进行管理时，会频繁的对Page_list进行链表的插入或删除操作，且还有其他操作也可能会有类似的操作，因此在具体实现时直接调用宏，一可以让主代码更简洁清晰，二可以让这段代码反复被调用，有效减少代码量。

- **请你查看实验环境中的 /usr/include/sys/queue.h，了解其中单向链表与循环链表的实现，比较它们与本实验中使用的双向链表，分析三者在插入与删除操作上的性能差异。**

  答：简要对比可以发现，在**双向链表的实现**上，实验环境和本实验中的queue.h几乎相似，只是对其中一些内容进行了封装，如将`(elm)->field.le_next`封装为`LIST_NEXT(elm, field)`，前向指针和后向指针所指位置一致，在插入和删除的实现上为了更直观，此处采用实验环境中的双向链表与单向链表和循环链表作对比。此外，对于插入操作，通过插入到头部时指针的变化，分析指针的修改，进而分析各链表插入的性能（因为之前图已经画好了就只能基于插入头来引入了，但都会说~）。

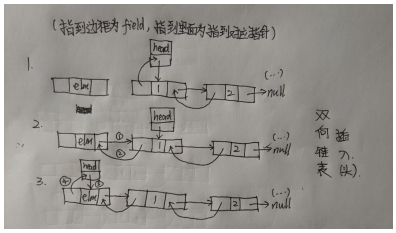
  实验环境的双向链表，基本与本实验一致：

```
#define LIST_INSERT_HEAD(head, elm, field) do {             \
    if (((elm)->field.le_next = (head)->lh_first) != NULL)      \
        (head)->lh_first->field.le_prev = &(elm)->field.le_next;\
    (head)->lh_first = (elm);                       \
    (elm)->field.le_prev = &(head)->lh_first;           \
} while (/*CONSTCOND*/0)

#define LIST_INSERT_AFTER(listelm, elm, field) do {         \
    if (((elm)->field.le_next = (listelm)->field.le_next) != NULL)  \
        (listelm)->field.le_next->field.le_prev =       \
            &(elm)->field.le_next;              \
    (listelm)->field.le_next = (elm);               \
    (elm)->field.le_prev = &(listelm)->field.le_next;       \
```

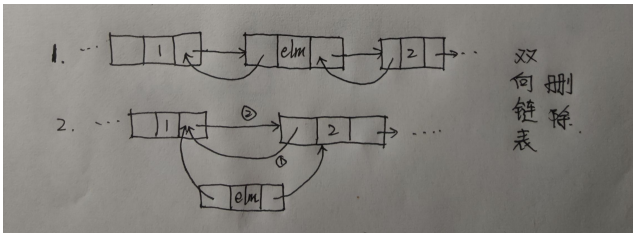
```
} while (/*CONSTCOND*/0)

#define LIST_INSERT_BEFORE(listelm, elm, field) do {            \
    (elm)->field.le_prev = (listelm)->field.le_prev;        \
    (elm)->field.le_next = (listelm);               \
    *(listelm)->field.le_prev = (elm);              \
    (listelm)->field.le_prev = &(elm)->field.le_next;       \
} while (/*CONSTCOND*/0)

#define LIST_REMOVE(elm, field) do {                    \
    if ((elm)->field.le_next != NULL)               \
        (elm)->field.le_next->field.le_prev =           \
            (elm)->field.le_prev;               \
    *(elm)->field.le_prev = (elm)->field.le_next;           \
} while (/*CONSTCOND*/0)
```

双向链表插入，需要对前向指针和后向指针进行修改，修改的指针内容相比单向链表较多，但在给定要插入的位置时，可以通过给定的链表项快速定位到该链表项前一项的后向指针和后一项的整个链表项，因此可以快速地进行插入操作。通过代码可以看出，在向前插入和向后插入中均不存在循环。在代码里可以找到 LIST_INSERT_HEAD，但找不到LIST_INSERT_TAIL，不过向尾部插入的实现在本次实验中有写，因为无法直接访问到尾部的链表项，因此只能通过遍历的方式寻找尾部，性能较差，代码中存在循环。

双向链表删除，需要对链表项前一项的后向指针和后一项的前向指针修改，同样可以通过对这两个指针的快速定位实现快速的删除操作，也不存在循环。



双向链表插入头



双向链表删除

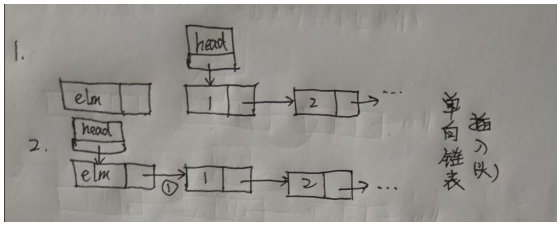
实验环境的单向链表：

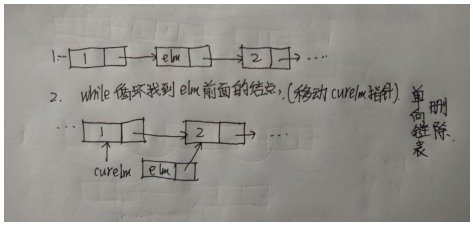
```
#define SLIST_INSERT_HEAD(head, elm, field) do {            \
    (elm)->field.sle_next = (head)->slh_first;          \
    (head)->slh_first = (elm);                  \
} while (/*CONSTCOND*/0)

#define SLIST_INSERT_AFTER(slistelm, elm, field) do {           \
    (elm)->field.sle_next = (slistelm)->field.sle_next;     \
    (slistelm)->field.sle_next = (elm);             \
} while (/*CONSTCOND*/0)

#define SLIST_REMOVE(head, elm, type, field) do {           \
    if ((head)->slh_first == (elm)) {               \
        SLIST_REMOVE_HEAD((head), field);           \
    }                                   \
    else {                                  \
        struct type *curelm = (head)->slh_first;        \
        while(curelm->field.sle_next != (elm))          \
            curelm = curelm->field.sle_next;        \
```

```
        curelm->field.sle_next =                \
            curelm->field.sle_next->field.sle_next;     \
    }                                   \
} while (/*CONSTCOND*/0)
```

单向链表插入，观察实验环境代码，发现并没有SLIST_INSERT_BEFORE的实现。在单向链表中，给定一个链表项，只能快速定位到其下一个链表项，而不可以快速定位到其前一个链表项。在向后插入时，因为只需要更新后向指针，因此相比双向链表会略快一点，且结果简单，如果想要实现向前插入，则需要定义一个指针，从头部向后遍历直到发现其后向指针所指的链表项为待插入链表项的位置参照。也就是说，若要向前插入链表项，就需要使用循环从头部遍历，插入的位置越靠后，所需时间越长，性能越差。类似于双向链表，也没用发现SLIST_INSERT_TAIL的实现（大概是需要尾部插入或向前插入时一般不会采用此种链表结构），如果需要进行尾部插入，同样需要进行遍历，性能较差。

单向链表删除，同理，因为需要修改待删除链表项的前一个链表项的后向指针，因此需要从头遍历，位置越靠后，删除性能越差。





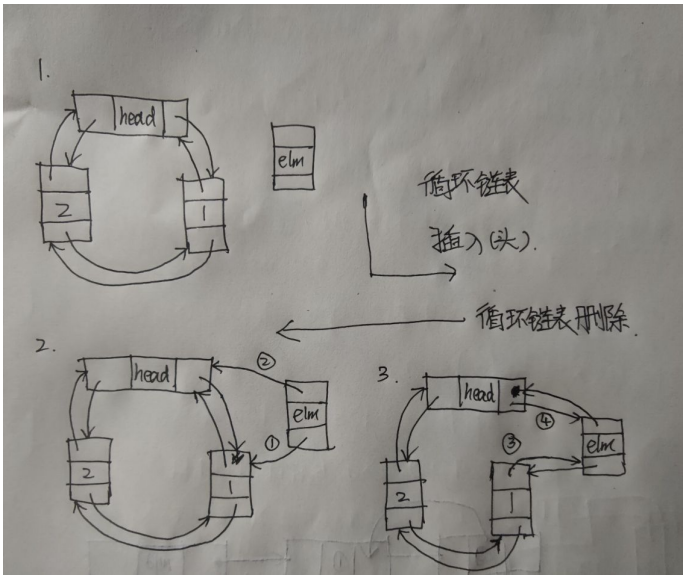
单向链表插入头 | 单向链表删除

实验环境的循环链表：

```
#define CIRCLEQ_INSERT_HEAD(head, elm, field) do {          \
    (elm)->field.cqe_next = (head)->cqh_first;          \
    (elm)->field.cqe_prev = (void *)(head);             \
    if ((head)->cqh_last == (void *)(head))             \
        (head)->cqh_last = (elm);               \
    else                                    \
        (head)->cqh_first->field.cqe_prev = (elm);      \
    (head)->cqh_first = (elm);                  \
} while (/*CONSTCOND*/0)

#define CIRCLEQ_INSERT_TAIL(head, elm, field) do {          \
    (elm)->field.cqe_next = (void *)(head);             \
    (elm)->field.cqe_prev = (head)->cqh_last;           \
    if ((head)->cqh_first == (void *)(head))            \
        (head)->cqh_first = (elm);              \
    else                                    \
        (head)->cqh_last->field.cqe_next = (elm);       \
    (head)->cqh_last = (elm);                   \
} while (/*CONSTCOND*/0)

#define TAILQ_INSERT_AFTER(head, listelm, elm, field) do {      \
    if (((elm)->field.tqe_next = (listelm)->field.tqe_next) != NULL)\
        (elm)->field.tqe_next->field.tqe_prev =         \
            &(elm)->field.tqe_next;             \
    else                                    \
        (head)->tqh_last = &(elm)->field.tqe_next;      \
```

```
    (listelm)->field.tqe_next = (elm);              \
    (elm)->field.tqe_prev = &(listelm)->field.tqe_next;     \
} while (/*CONSTCOND*/0)

#define TAILQ_INSERT_BEFORE(listelm, elm, field) do {           \
    (elm)->field.tqe_prev = (listelm)->field.tqe_prev;      \
    (elm)->field.tqe_next = (listelm);              \
    *(listelm)->field.tqe_prev = (elm);             \
    (listelm)->field.tqe_prev = &(elm)->field.tqe_next;     \
} while (/*CONSTCOND*/0)

#define CIRCLEQ_REMOVE(head, elm, field) do {               \
    if ((elm)->field.cqe_next == (void *)(head))            \
        (head)->cqh_last = (elm)->field.cqe_prev;       \
    else                                \
        (elm)->field.cqe_next->field.cqe_prev =         \
            (elm)->field.cqe_prev;              \
    if ((elm)->field.cqe_prev == (void *)(head))            \
        (head)->cqh_first = (elm)->field.cqe_next;      \
    else                                \
        (elm)->field.cqe_prev->field.cqe_next =         \
            (elm)->field.cqe_next;              \
} while (/*CONSTCOND*/0)
```

循环链表是一种特殊的双向链表，其前向指针指的就是上一个链表项整体而不是上一个链表项的后向指针，且循环链表收尾相接，头部的前向指针指向尾部的链表项，尾部的后向指针指向头部。因为它是一种双向链表，因此具有双向链表的特性，通过前向指针可以访问到上一个链表项，通过后向指针可以访问到后一个链表项，所以也可以快速定位到前后两个链表项，从而实现快速地向前和向后插入。同样的，代码中无循环。可以看到，循环链表中给出了尾部插入的实现，对于循环链表给出头部，可以通过头部的前向指针快速地定位到尾部元素，因此在进行尾部插入时相比其他两种链表结果具有更高的性能。循环链表删除，类似于双向链表。



循环链表插入头和删除

总结：

插入头部：三种链表都可以直接找到头部，性能相似，单向链表可能会因指针修改数量少快一点点；

插入尾部：循环链表可以通过头部的前向指针快速定位尾部，性能占优，其余两种只能遍历；

向前插入：双向链表和循环链表可以通过位置参照的前向指针快速定位到前一个链表项或前一个链表项的后向指针，实现快速插入，单向链表需要遍历；

向后插入：三种链表都可以通过后向指针快速定位到后一个链表项，单向链表可能会因指针修改数量少快一点点；

删除：类似于向前插入，循环链表和双向链表占优。

额外的，单向链表的优点之一，结构简单。

Thinking 2.3

请阅读 include/queue.h 以及 include/pmap.h, 将 Page_list 的结构梳理清楚，选择正确的展开结构。

答：`pp_link` 即为对应的链表项，`pp_ref` 对应这一页物理内存被引用的次数，`Page_list` 包含多个页，每一页都包含一个 `pp_ref` 和一个对应的链表项 `pp_link`。

Page_list正确的展开结构：

```
struct Page_list{
    struct {
        struct {
            struct Page *le_next;
            struct Page **le_prev;
        } pp_link;
        u_short pp_ref;
    }* lh_first;
}
```

Thinking 2.4

请你寻找两个 boot_* 函数boot_pgdir_walk和boot_map_segment 在何处被调用。

- `boot_pgdir_walk` 函数的作用是返回一级页表基地址 pgdir 对应的两级页表结构中，va 这个虚拟地址所在的二级页表项，如果 create 不为 0 且对应的二级页表不存在则会使用 alloc 函数分配一页物理内存用于存放。该函数在 `boot_map_segment` 中被调用，`boot_map_segment` 函数对于size大小虚拟地址区间内的每一个虚拟地址均调用一次 `boot_pgdir_walk` 函数，从而实现区间地址映射。
- `boot_map_segment` 函数的作用是将一级页表基地址 pgdir 对应的两级页表结构做区间地址映射，将虚拟地址区间 [va, va + size − 1] 映射到物理地址区间 [pa, pa + size − 1]，该函数在 `mips_vm_init` 函数中被调用。`mips_vm_init` 函数为全局数组 `pages` 分配适当大小的物理内存用于物理内存管理，需要将虚拟地址与之前分配的物理地址页面做映射，此时则需要靠 `boot_map_segment` 函数来实现；此外，还需要为全局数组 `envs` 分配适当大小的物理内存用于流程管理，同样需要靠 `boot_map_segment` 函数来实现物理地址与虚拟地址间的映射。

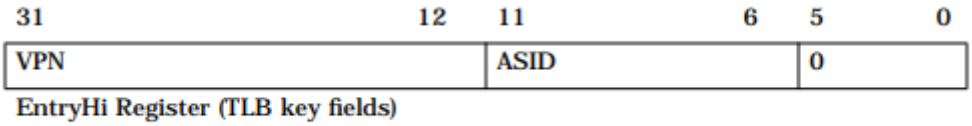
Thinking 2.5

- **请阅读上面有关 R3000-TLB 的叙述，从虚拟内存的实现角度，阐述 ASID 的必要性。**

  在内存管理中使用虚拟内存，在同一时间，多个进程可能会使用到同一个虚拟地址，此时就需要将这一个虚拟地址按进程分别映射到不同的物理地址，因此我们需要使用多个地址空间，而ASID可以唯一标识每个进程，用来匹配当前正在运行的进程和虚拟页面，也即为每个进程提供地址空间保护，从而实现不同进程下虚拟内存到物理内存一对多的映射。

- **请阅读《IDT R30xx Family Software Reference Manual》的 Chapter 6，结合 ASID 段的位数，说明 R3000 中可容纳不同的地址空间的最大数量。**

  以下图片均来自《IDT R30xx Family Software Reference Manual》。



EntryHi Register (TLB key fields)

Figure 6.1. EntryHi and EntryLo register fields

观察 EntryHi 寄存器的结构可以发现ASID段的位数为6，结合以下说明，可以计算出R3000 中可容纳不同的地址空间的最大数量为 $2^6 = 64$，也即允许同时映射最多64个不同的地址空间而不用在进程切换时清除TLB。如果用光了ASID，则必须挑选一个可以彻底将其扔出TLB的进程，在丢弃其所有映射后可以回收其ASID值用于其他进程。

Using ASIDs

By setting up TLB entries with a particular ASID setting and with the *EntryLo* G bit zero, those entries will only ever match a program address when the CPU's *ASID* register is set the same. This allows software to map up to 64 different address spaces simultaneously, without requiring that the OS clear out the TLB on a context change.

In typical usage, new tasks are assigned an "un-initialized" ASID. The first time the task is invoked, it will presumably miss in the TLB, allowing the assignment of an ASID. If the system does run out of new ASIDs, it will flush the TLB and mark all tasks as "new". Thus, as each task is re-entered, it will be assigned a new ASID. This sequence is expected to happen infrequently if ever.

Thinking 2.6

- **tlb_invalidate 和 tlb_out 的调用关系是怎样的?**

  答：阅读 mm/pmap.c 可以发现 `tlb_invalidate` 的定义：

```
// Overview:
//  Update TLB.
void tlb_invalidate(Pde *pgdir, u_long va)
{
    if (curenv) {
        tlb_out(PTE_ADDR(va) | GET_ENV_ASID(curenv->env_id));
    } else {
        tlb_out(PTE_ADDR(va));
    }
}
```

从中可以看出`tlb_out`在`tlb_invalidate`中被调用。

- **请用一句话概括 tlb_invalidate 的作用。**

  当页表项中的物理地址合法，也即虚拟地址va原本就有映射，需要先解除原来的映射时，调用tlb_invalidate可以使快表失效，可以分析出tlb_invalidate的作用是 **取消va与相应物理页之间的关联。**

- **逐行解释 tlb_out 中的汇编代码。**

  汇编指令作用：

  `MFC0`指令：读`CP0`寄存器；

  `MTC0`指令：写`CP0`寄存器；

  汇编代码解释：

```
LEAF(tlb_out)
/*1: j 1b*/
nop
  mfc0    k1,CP0_ENTRYHI  /*将原本ENTRYHI寄存器的值读到k1寄存器进行保存*/
  mtc0    a0,CP0_ENTRYHI    /*将输入参数写入ENTRYHI寄存器*/
  /*联系上一问，在当前存在进程的情况下，输入参数为虚拟地址对应的虚拟页号与当前进程地址空间逻辑或的结果*/
  /*若不存在进程，则为虚拟地址对应的虚拟页号（地址空间为0），之后该参数将被作为查找TLB的KEY*/
  nop
  tlbp
  /*根据ENTRYHI寄存器中的KEY（包含VPN与ASID），查找TLB中与之对应的表项*/
  /*并将表项的索引存入 INDEX 寄存器（若未找到匹配项，则 INDEX 最高位被置 1）*/
  nop
  nop
  nop
  nop
  mfc0    k0,CP0_INDEX /*将INDEX寄存器的值，也即表项索引读到k0寄存器*/
  bltz    k0,NOFOUND /*若最高位（即符号位）若为1，则表明未找到匹配项，跳转到标签NOTFOUND*/
  nop   /*若找到匹配项则继续执行以下操作，使对应的TLB失效*/
  mtc0    zero,CP0_ENTRYHI  /*将ENTRYHI寄存器置0*/
  mtc0    zero,CP0_ENTRYLO0 /*将ENTRYLO0寄存器置0*/
  nop
  tlbwi
  /* INDEX寄存器中之前存入了索引,将此时寄存器ENTRYHI和ENTRYLO0的零值写到索引指定的TLB表项中*/
NOFOUND:

  mtc0    k1,CP0_ENTRYHI    /*将原本ENTRYHI寄存器的值恢复*/

  j   ra /*函数结束，跳转*/
  nop
END(tlb_out)
```

Thinking 2.7

在现代的 64 位系统中，提供了 64 位的字长，但实际上不是 64 位页 式存储系统。假设在 64 位系统中采用三级页表机制，页面大小 4KB。由于 64 位系统中字长为 8B，且页目录也占用一页，因此页目录中有 512 个页目录项，因此每级页表都需要 9 位。因此在 64 位系统下，总共需要 3 × 9 + 12 = 39 位就可以实现三级页表机制，并不需要 64 位。 现考虑上述 39 位的三级页式存储系统，虚拟地址空间为 512 GB，若记三级页表的基地址为 PTbase，请你计算：

- **三级页表页目录的基地址**

  答：在64位系统中，页面大小也为 4KB，因此三级页表的基地址为第`PTbase << 12`个页表项，因为64位系统中字长为 8B，所以有二级页表的基地址相对三级页表的基地址（PTbase）的偏移为`PTbase << 12 >> 3 = PTbase << 9`，二级页表的基地址为`PTbase + PTbase << 9`，页目录基地址相对二级页表基地址的偏移为`PTbase << 9 << 12 >> 3 = PTbase << 18`，最终得到三级页表结构中页目录的基地址为`PTbase + PTbase << 9 + PTbase << 18`。

- **映射到页目录自身的页目录项（自映射）**

  答：与上一题同理，映射到页目录自身的页目录项地址相对页目录基地址的偏移为`PTbase << 18 << 12 >> 3 = PTbase << 27`，因此映射到页目录自身的页目录项地址为`PTbase + PTbase << 9 + PTbase << 18 + PTbase << 27`。

Thinking 2.8

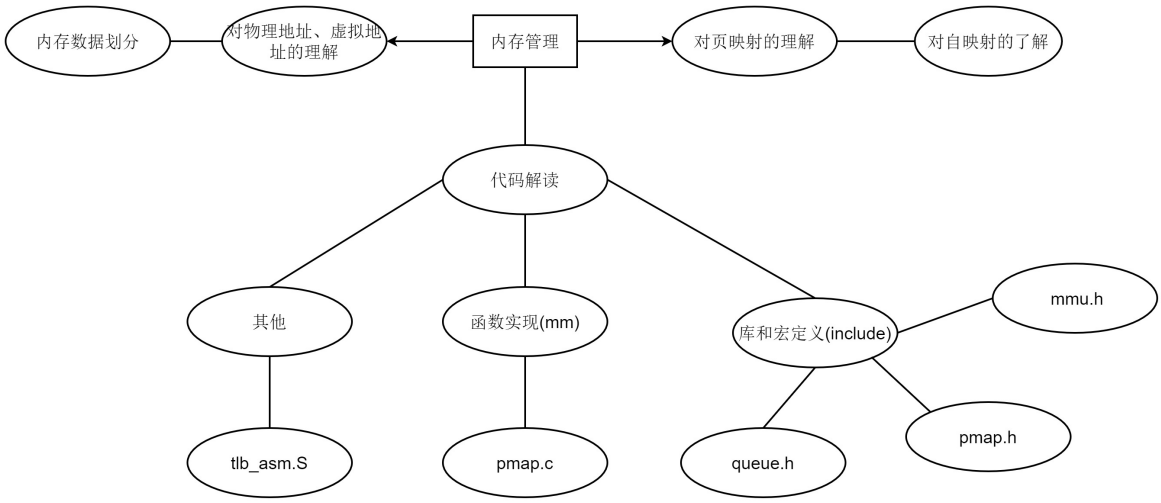
任选下述二者之一回答：

- **简单了解并叙述 X86 体系结构中的内存管理机制，比较 X86 和 MIPS 在内存管理上的区别。**
- **简单了解并叙述 RISC-V 中的内存管理机制，比较 RISC-V 与 MIPS 在内存管理上的区别。**

  选x86：

  答：在MIPS内存管理中使用MMU进行地址映射，其主要硬件是TLB，当给出虚拟地址要对物理地址进行访问时，就需要将虚拟地址通过MMU使用分页机制映射到物理地址，在多进程的情况下还为每个进程分配地址空间。而x86架构的内存管理机制分为分段机制和分页机制两部分，分段机制为程序提供彼此隔离的代码区域、数据区域、栈区域，从而避免了同一个处理器上运行的多个程序互相影响，而分页机制实现了传统的按需分页、虚拟内存机制，可以将程序的执行环境按需映射到物理内存。此外，分页机制还可以用于提供多任务的隔离。对于x86体系结构，处理器无论在何种运行模式下都不可以禁止分段机制，但是分页机制却是可选选项。关于分段机制的简要说明：当程序使用逻辑地址访问内存的某个部分时，CPU通过逻辑地址中的段选择符索引段描述符表，进而得到该内存对应的段描述符，再根据段描述符中的段属性信息检测程序的访问是否合法，如果合法，再根据段描述符中的基地址将逻辑地址转换为线性地址。

2、实验难点图示



3、体会与感受

难度评价适中，实验花费时间约4.5h，实验报告约3h，时间没那么长主要是因为指导书中给了框架，只需对mmu.h和pmap.h中的宏定义和函数定义掌握即可完成实验，很感谢助教。写函数最大的问题就是，看别人写代码知道为什么这样写，但自己上手就会有些无从上手，特别是在课上实验中这种体验尤为明显，而在给框架的情况下，只要理解每个函数的作用，即可根据框架尝试去对函数进行实现。以第一次课上的Extra伙伴系统为例，如果给出一个伙伴结构的结构体，并适当提示一些实现思路（课上不需要像指导书这么详细），结果应该会比现在这样好很多，exam也有类似的情况，上手写第一步比完成后边的内容难度要高很多。再之后就是评测，第二次课上的Extra需要修改很多函数，还需要再去写一个函数，但一旦某一个地方出现bug不是很好查，而且在大多数内容完成的情况下如果有少部分功能没有时间去加或者不会实现导致结果错误（比如映射权限位的复制），则在花费大量时间的同时直接爆零，特别痛苦。总体来说，不管成绩如何，通过这次实验学到了很多，对分页机制和自映射、TLB等内容掌握程度有所加深。

4、指导书反馈

- 在`pgdir_walk`函数的框架中没有对页面引用次数(pp_ref)加一，如果自己写的话因为跟着框架走的惯性思维很容易漏掉，在debug的时候很容易漏掉，建议在框架里加上`p->pp_ref++`；
- 之前群里提到过的`KADDR`定义，将倒数第二行改为`((u_long)(pa)) + ULIM`。

5、残留难点

- TLB部分的内容，如地址空间、TLB重填等。