操作系统lab2实验报告

姓名：高远 学号：19374242 班级：202113

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。

MIPS 汇编程序中lw, sw使用的是虚拟地址。

Thinking 2.2:

宏实现链表的好处：宏的好处是让C语言这个本没有“泛型”的语言实现了类似“泛型”的功能，因为指向前方与后方的指针被结构体内置起来，其与链表每个节点中的数据类型相对独立，可以在任何数据类型中得到很好的使用。其次，前指针不指向上一节点而是指向其后指针，从而对链表指针的操作变得更加容易，灵活，从而是许多对链表的基本操作更加容易，具有较高的可重用性。

性能差异：本实验中的双向链表节点信息中存放了前一项中存放下一项的指针的地址，从而更加方便了插入和移出节点的操作。这样可以在只知道一个链表节点的情况下对链表进行插入或删除节点操作。

Thinking 2.3:

由选项易知，本题选C

Thinking 2.4:

第一个boot在boot_map_segment被调用，第二个boot*在mips_vm_init()被调用。

Thinking 2.5:

有了ASID后，TLB在进程切换的时候，操作系统不需要去刷新TLB了，因为MMU在做地址转换时会将TLB表项里的ASID和当前进程的ASID值做比较，只有ASID值相等，MMU才认为这条表项是我需要的。所以和没有ASID技术相比，在进程切换上提升了较大的性能。
ASID是6位，所以可容纳不同的地址空间的最大数量为2^6

Thinking 2.6:

tlb_invalidate()调用tlb_out()把va对应的tlb项清空

tlb_invalidate()作用：删除特定虚拟地址的映射

tlb_out()汇编代码：前两个指令读取并写入cp0的$10，其中$a0是进入函数时的参数。然后的tlbp指令，这个指令会把检查TLB，如果TLB中有项和*EntryHi*寄存器匹配，就把*Index*寄存器设置为对应的项，如果没有匹配的项，就把*Index*最高位设置为1（也就是变成负数）。接下来我们读取Index的值，如果小于零，显然没找到，那就跳转到NOFOUND，恢复遍历之前的状况，跳转回之前函数。如果找到了，那就写入TLB，tlbwi把ENTRYHI和ENTRYLO0写入Index所指的项，在这里，就是把这一项清零了。 最后我们还是会进入NOFOUND，把一开始ENTRYHI数值重新输入回去，恢复现场，跳转回去。

Thinking 2.7:

三级页表页目录基地址：PTbase+(PTbase>>12)✕8+
((PTbase>>12)✕8>>12)✕8=PTbase+PTbase>>9+PTbase>>18

映射到页目录自身的页目录项(自映射)：PTbase+PTbase>>9+PTbase>>18+PTbase>>27

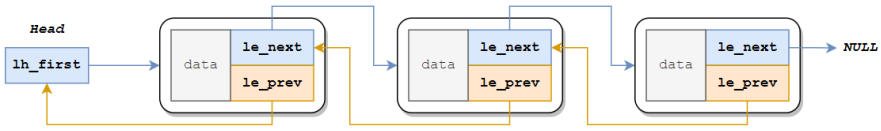
Thinking 2.8:

选择讲述RISC-V 中的内存管理机制：RISC-V 架构是基于精简指令计算原理建立的开放指令集架构，它对指令集功能做了良好分割，以实现渐进式兼容和灵活的扩展性，做到了指令功能的平衡与规整。RISC-V有一系列的CSR用来管理各种内存设置，其中有一个叫做satp的寄存器，用来管理页表，mode部分填写使用页表类型（RISC-V可以使用多种长度和层数的页表，也可以不使用页表直接访问物理内存），ASID填写进程编号，PPN填写最初一级页表物理地址。所以在RISC-V架构下，MMU直接去物理地址找根页表，然后一路往下找到物理地址。

区别：MIPS没有物理的MMU，是用TLB模拟的。MIPS指令都是固定长度的，RISC-V指令是变长的。

实验难点图示

难点1：链表宏



链表宏的理解花费了我大量的时间，这种链表的结构其实非常特殊，不同于普通的链表，该链表的pp_link是由LIST_ENTRY定义，可以看出LIST_ENTRY是作为Page结构体的子结构体使用的，其prev指针也并不是指向了上一个Page结构体，而是指向了指向自身所在的Page结构体的指针。

难点2：do{}while(0)的使用

观察代码可以注意到，许多宏都定义为了`do{...} while(0)`的形式，定义函数宏有多种方式，例如直接将宏定义为多个语句，或者将宏定义为一个语句块（即由大括号括起的一组语句），也可以像此处一样定义为do`-while`的形式。这三种各有优缺点。

以`SWAP`为例，第一种方式为：

```
#define SWAP(a, b, type) type tmp = *a;\
*a = *b;\
*b = tmp;
```

第二种方式为：

```
#define SWAP(a, b, type) {type tmp = *a;\
*a = *b;\
*b = tmp;\
}
```

第三种方式为：

```
#define SWAP(a, b, type) do{\
type tmp = *a;\
*a = *b;\
*b = tmp;\
} while(0)
```

第一种的优点是方便，但如果跟在一个没有加大括号的`if`后边，只有第一个语句属于这个`if`，无形之间引入问题。第二种语句块整体的值是最后一个语句的值，相当于一个有返回值的函数宏，然而如果跟在一个没有加大括号的`if`后边，如果加了分号（例如`if(a != b) SWAP(a, b, int); else ...`），则`if`的作用域会被分号提前终结，导致在`else`处报错。第三种没有前两者的问题，但整体无返回值，相当于一个返回值为`void`的函数宏。

难点3：页面与地址的转换

本次实验涉及很多页面与地址的转化，总结如下：

page2kva(struct Page pp) *得到页pp的虚地址*
page2ppn(struct Page pp) 得到页pp的物理页号
page2pa(struct Page *pp) 得到页pp的实地址
pa2page(u_long *pa*) 得到实地址pa的页
PPN(*va*) 得到实地址va的物理页号

源码如下：

```
#define PPN(va)     (((u_long)(va))>>12)
#define VPN(va)     PPN(va)
```

```
page2ppn(struct Page *pp)
{
    return pp - pages;
}
```

```
page2pa(struct Page *pp)
{
    return page2ppn(pp) << PGSHIFT;
}
```

```
page2kva(struct Page *pp)
{
    return KADDR(page2pa(pp));
}
```

```
pa2page(u_long pa)
{
    if (PPN(pa) >= npage) {
        panic("pa2page called with invalid pa: %x", pa);
    }

    return &pages[PPN(pa)];
}
```

难点4：系统启动相关函数和进程运行相关函数

首先便是标志位的问题，因为页表地址只有前20位有效，后12位为标志位，因此可以将后12位置为0，然后通过或上一些只有十二位的常数，进行标志位的确定，常数如下图：

```
20 #define PTE_G         0x0100    // 全局位
21 #define PTE_V         0x0200    // 有效位
22 #define PTE_R         0x0400    // 修改位，如果是0表示只对该页面进行过读操作，否则进行过写操作，要引发中断将内容写回内存
23 #define PTE_D         0x0002    // 文件缓存的修改位dirt
```

而在访问物理地址时，需将标志位置零，因此可以使用函数PTE_ADDR(pde) 将后12位清零，得到物理地址，源码如下：

```
#define PTE_ADDR(pte)   ((u_long)(pte)&~0xFFF)
```

其次，CPU 发出的地址均为虚拟地址，因此获取相关物理地址后，需要转换为虚拟地址再访问。对页表进行操作时处于内核态，因此使用宏 `KADDR` 即可。

难点5：tlb_invalidate函数

使用 `tlb_invalidate` 函数可以实现删除特定虚拟地址的映射，每当页表被修改，就需要调用该函数以保证下次访问该虚拟地址时诱发 TLB 重填以保证访存的正确性。值得注意的是，该函数作用为清空TLB中va对应的项，而不是直接改变映射。源码如下：

```
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));
    }
}
```

难点6：自映射

自映射的概念较为简单，但是计算页目录地址还是需要一段时间的理解。例如0x7fc00000-0x80000000这 4MB 空间的起始位置（也就是第一个二级页表的基地址）对应着页目录的第一个页目录项。同时由于 1M 个页表项和 4GB 地址空间是线性映射的，不难算出 0x7fc00000 这一个地址对应的应该是第 0x7fc00000 >> 12 个页表项（这一个页表项也就是第一个页目录项）。由于一个页表项占 4B 空间，因此第 0x7fc00000>> 12 个页表项相对于页表基地址的偏移为 (0x7fc00000 >> 12) * 4，即 0x1ff000。最终即可得到页目录基地址为 0x7fdff000。

体会与感想

就我而言，本次实验给我的挑战远远大于lab1，尤其是页表的使用，给我带来了极大的困惑，但是，做出lab2也给我带来了极大的收获。因为看不懂题目中的函数，我大量阅读源码，对页表初始化、分配以及宏和结构体的使用有了更为深刻的了解。对页转化为物理/虚拟地址和虚拟/物理地址转化为页/页号的函数的使用可以说是滚瓜烂熟。与此同时，我也复习巩固了指针的使用方法，可以说，lab2实验使我受益匪浅。

指导书反馈

指导书中对tlb的讲解较为抽象，也没有很好的实例帮助理解，希望能够在后面的实验中对tlb有更好地讲解。

残留难点

本次实验残留的难点主要在于对tlb相关函数的作用还不是特别清楚，对其使用环境也模糊不清。同时，对很多有关地址的函数代表的是实地址还是虚地址难以很快的进行判断。