BUAA_OS_Lab2

一、实验思考题

Thinking 2.1

> 请根据上述说明，回答问题：在编写的 C 程序中，指针变量中存储的地址被视为虚拟地址，还是物理地址？MIPS汇编程序中lw和sw指令使用的地址被视为虚拟 地址，还是物理地址？

指针变量中存储的地址是虚拟地址，MIPS汇编程序中lw和sw指令使用的地址也是虚拟地址。

Thinking 2.2

> 请思考下述两个问题：
>
> - 从可重用性的角度，阐述用宏来实现链表的好处。
> - 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。

宏实现链表可以减少代码的复用，可重用性强，链表的插入和删除等操作使用宏可以保证这些操作都是一致的，从而减少了出错的可能性，可读性强，易于维护。

单向链表简单的插入与删除操作时间复杂度为O(1), 但是对任意的第n个元素的插入和删除操作需要遍历链表，时间复杂度为O(N); 双向链表对于任意第n个元素的插入与删除操作时间复杂度都为O(1); 循环链表中单向循环链表与单向链表一样，对任意的第n个元素的插入和删除操作时间复杂度为O(N)，双向循环链表对于任意第n个元素的插入与删除操作时间复杂度都为O(1)。

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。

C

Thinking 2.4

> 请思考下面两个问题：
>
> - 请阅读上面有关TLB的描述，从虚拟内存和多进程操作系统的实现角度，阐述ASID 的必要性。
> - 请阅读 MIPS 4Kc 文档《MIPS32® 4K™ Processor Core Family Software User's Manual》的 Section 3.3.1 与 Section 3.4，结合 ASID 段的位数，说明 4Kc 中可容纳不同的地址空间的最大数量。

ASID用于区分不同的地址空间，同一虚拟地址在不同的地址空间中通常映射到不同的物理地址。ASID机制可以确保一个进程无法访问其他进程的内存区域，从而实现地址空间的隔离，起到内存保护的作用。同时，多进程操作系统通过ASID来管理和跟踪每个进程的虚拟地址空间。当进程切换时，操作系统可以根据ASID更新TLB的内容。

ASID段位数为8位，所以4Kc 中可容纳不同的地址空间的最大数量为256。

Thinking 2.5

> 请回答下述三个问题：
>
> - tlb_invalidate和tlb_out的调用关系？
> - 请用一句话概括tlb_invalidate的作用。
> - 逐行解释tlb_out中的汇编代码。

tlb_invalidate调用tlb_out; tlb_invalidate 函数实现删除特定虚拟地址在 TLB中的旧表项。

```
#include <asm/asm.h>

LEAF(tlb_out)
.set noreorder
    mfc0    t0, CP0_ENTRYHI //将当前的VPN和ASID存储到t0寄存器中，用于后续恢复
CP0_ENTRYHI
    mtc0    a0, CP0_ENTRYHI //将调用函数传入的旧表项的Key(由VPN和ASID组成)写进
CP0_ENTRYHI
    nop
    /* Step 1: Use 'tlbp' to probe TLB entry */
    /* Exercise 2.8: Your code here. (1/2) */
    tlbp //根据CP0_EntryHi中的Key查找TLB中对应的旧表项，将表项的索引存入CP0_Index。
    nop
    /* Step 2: Fetch the probe result from CP0.Index */
    mfc0    t1, CP0_INDEX //将CP0_Index存储t1寄存器。
.set reorder
    bltz    t1, NO_SUCH_ENTRY //如果t1寄存器中的值小于0（即TLB中不存在
Key对应的表项），跳转到NO_SUCH_ENTRY标签处。
.set noreorder
    //如果t1寄存器中的值大于等于0（即TLB中存在Key对应的表项），我们向EntryHi和EntryLo0、
EntryLo1中写入0
    mtc0    zero, CP0_ENTRYHI
    mtc0    zero, CP0_ENTRYLO0
    mtc0    zero, CP0_ENTRYLO1
    nop
    /* Step 3: Use 'tlbwi' to write CP0.EntryHi/Lo into TLB at CP0.Index  */
    /* Exercise 2.8: Your code here. (2/2) */
    tlbwi   //将EntryHi和EntryLo0、EntryLo1中的值写入索引指定的表项。此时旧表项的Key和
Data被清零，实现将其无效化
.set reorder

NO_SUCH_ENTRY:
    mtc0    t0, CP0_ENTRYHI //将原始的ENTRYHI寄存器的值（保存在t0中）恢复回ENTRYHI寄存
器。
    j       ra //函数结束并返回
END(tlb_out)
```

Thinking A.1

> 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512GB，若三级页表的基地 址为PTbase，请计算：
>
> •三级页表页目录的基地址。
>
> •映射到页目录自身的页目录项（自映射）。

三级页表页目录的基地址是(PTbase >> 30) << 30; 映射到页目录自身的页目录项（自映射）为PTbase + ((PTbase >> 12) << 3) + ((((PTbase >> 12) << 3) >> 12) << 3) + ((((((PTbase >> 12) << 3) >> 12) << 3) >> 12) << 3)

Thinking 2.6

> 从下述三个问题中任选其一回答：
>
> • 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。
>
> • 简单了解并叙述LoongArch 中的内存管理机制，比较 LoongArch 与 MIPS 在内存 管理上的区别。

X86 体系结构中的内存管理机制:

- 通过分段将逻辑地址转换为线性地址，通过分页将线性地址转换为物理地址。
- X86采用分段机制来管理内存。内存被划分为多个段，每个段都有起始地址和长度描述。CPU通过逻辑地址访问内存，这个逻辑地址包含段选择符和偏移量。
- 当CPU尝试访问某个内存位置时，它会使用段选择符在段描述符表中查找对应的段描述符，然后根据段描述符中的基地址和偏移量相加就是线性地址。
- X86使用分页机制来进一步管理内存。内存被划分为固定大小的页面，每个页面都有一个唯一的物理地址。
- 操作系统创建全局描述符表和提供逻辑地址，之后的分段操作x86的CPU会自动完成，并找到对应的线性地址。
- 从线性地址到物理地址的转换是CPU自动完成的，转化时使用的Page Directory和Page Table等需要操作系统提供。

X86和MIPS在内存管理上的区别：

- 在TLB处理方面，当TLB不命中时（即TLB中没有缓存所需的虚拟地址到物理地址的映射关系），MIPS会触发一个异常（如TLB Refill异常），然后由内核的特定处理程序来处理这个异常并更新TLB。而X86则是由硬件的MMU（内存管理单元）直接处理不命中情况，它会使用页表来找到正确的物理地址，并更新TLB以便下次快速访问。
- MIPS属于RISC（精简指令集计算机）架构，其指令集固定长度且数量有限，通常通过Load/Store指令来访问内存。而X86则属于CISC（复杂指令集计算机）架构，其指令集更加丰富和复杂，包括多种直接操作内存的指令。
- 分页方式不同：一种MIPS系统内部只有一种分页方式，x86的CPU支持三种分页模式。
- 逻辑地址不同：MIPS地址空间32位，x86支持64位逻辑地址，同时提供转换为32位定址选项。

二、实验难点

mips_detect_memory 函数补全

总页数npage等于memsize除以单个页面的大小。

```
npage = memsize / PAGE_SIZE;
```

LIST_INSERT_AFTER函数

LIST_INSERT_AFTER的作用是将新加elm节点插入到链表现有listelm结点之后，我们需要利用已经定义的宏**LIST_NEXT(elm, field)**，按照**LIST_INSERT_BEFORE**的写法来完成函数。**注意**：宏定义函数每行需要用“\”进行链接。

```
#define LIST_NEXT(elm, field) ((elm)->field.le_next)
```

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

根据注释的提示，我们第一步需要将**listelm.next**分配给**elm.next**。

```
LIST_NEXT((elm), field) = LIST_NEXT((listelm), field); // assign 'elm.next' from
'listelm.next'.
```

第二步：如果**listelm.next**不为空，则将**elm**分配为**listelm.next**的前一个节点**listelm.next.pre**。

```
if(LIST_NEXT((listelm), field)) {                          \
            *((LIST_NEXT((listelm), field))->field.le_prev) = (elm);
      \
            (LIST_NEXT((listelm), field))->field.le_prev = &LIST_NEXT((elm),
field);   \
        }
```

第三步：将**elm**分配为**listelm.next**。

```
LIST_NEXT((listelm), field) = (elm);
```

第四步：将**elm**的前一个结点**elm.pre**置为**listelm.next**。

```
(elm)->field.le_prev = &LIST_NEXT((listelm), field);
```

page_init函数

需要利用**LIST_INIT**函数先初始化链表。

```
#define LIST_INIT(head)
                \
    do {
            \
        LIST_FIRST((head)) = NULL;
          \
    } while (0)
```

```
LIST_INIT(&page_free_list);
```

再利用**ROUND**函数将**freemem**与**PAGE_SIZE**大小对齐。

```
freemem = ROUND(freemem, PAGE_SIZE);
```

接着将**freemem**以下的内存的标记为**used**表明已使用过，即**pp_ref**置为1。需要用的页表头指针**pages**，计算出使用过的空间的页面数量，用循环遍历的方法赋值。

```
struct Page *pp = pages;
    u_long used = freemem & (0x80000000 - 1);
    int p1 = used / PAGE_SIZE;
    int i = 0;
    for (i = 0; i < p1; i++) {
        pp->pp_ref = 1;
        pp++;
    }
```

最后将其他内存标记为**free**，表示没使用过并使用LIST_INSERT_HEAD 将 其插入空闲链表。

```
while (i < npage) {
        pp->pp_ref = 0;
        LIST_INSERT_HEAD(&page_free_list, pp, pp_link);
        pp++;
        i++;
    }
```

page_alloc 函数

从空闲空间中取一个页面，如果不成功，则返回错误。

```
if (LIST_EMPTY(&page_free_list)) {
        return -E_NO_MEM;
    }
```

使用**memset**函数初始化这个页面，如果空闲链表有可用的页，取出链表头部的一页；初始化后，将该页对应的页 控制块的地址放到调用者指定的地方。

```
struct Page *pp;
pp = LIST_FIRST(&page_free_list);
LIST_REMOVE(pp, pp_link);
memset((void *)page2kva(pp), 0, PAGE_SIZE);
*new = pp;
```

page_free 函数

使用链表宏LIST_INSERT_HEAD，将页结构体插入空闲页结构体链表。

```
void page_free(struct Page *pp) {
    assert(pp->pp_ref == 0);
    /* Just insert it into 'page_free_list'. */
    /* Exercise 2.5: Your code here. */
    LIST_INSERT_HEAD(&page_free_list, pp, pp_link);
}
```

pgdir_walk函数

该函数的作用是：给定一个虚拟地址，在给定的页目录中查找这个虚拟地址对应的页表项，将其地址写入*ppte。如果这一虚拟地址对应的二级页表存在，则设置*ppte为这一页表项的地址；如果这一虚拟地址对应的二级页表不存在（即这一虚拟地址对应的页目录项无效）， 则当create不为0时先创建二级页表再查找页表项，为0时则将*ppte设置为空指针。

```
pgdir_entryp = &pgdir[PDX(va)];
if (!(*pgdir_entryp & PTE_V)) {
        if (create) {
            if (page_alloc(&pp) == 0) {
                *pgdir_entryp = page2pa(pp) | PTE_C_CACHEABLE | PTE_V;
                pp->pp_ref++;
            } else {
                return -E_NO_MEM;
            }
        } else {
            *ppte = NULL;
            return 0;
        }
    }
Pte *pgt = (Pte*)KADDR(PTE_ADDR(*pgdir_entryp));
*ppte = &pgt[PTX(va)];
return 0;
```

page_insert函数

但先要将TLB中缓存的页表项删掉，然后更新内存中的页表项。

```
tlb_invalidate(asid, va);
```

获得页表目录。

```
if(pgdir_walk(pgdir, va, 1, &pte) != 0) {
        return -E_NO_MEM;
    }
```

插入新的页。

```
*pte = page2pa(pp) | PTE_V | perm | PTE_C_CACHEABLE;
pp->pp_ref++;
```

kern/tlb_asm.S中的tlb_out函数

需要在两个位置插入两条指令，其中一个位置为tlbp，另一个位置为tlbwi。

```
LEAF(tlb_out)
.set noreorder
    mfc0    t0, CP0_ENTRYHI
    mtc0    a0, CP0_ENTRYHI
    nop
    /* Step 1: Use 'tlbp' to probe TLB entry */
    /* Exercise 2.8: Your code here. (1/2) */
    tlbp
    nop
```

```
    /* Step 2: Fetch the probe result from CP0.Index */
    mfc0    t1, CP0_INDEX
.set reorder
    bltz    t1, NO_SUCH_ENTRY
.set noreorder
    mtc0    zero, CP0_ENTRYHI
    mtc0    zero, CP0_ENTRYLO0
    mtc0    zero, CP0_ENTRYLO1
    nop
    /* Step 3: Use 'tlbwi' to write CP0.EntryHi/Lo into TLB at CP0.Index  */
    /* Exercise 2.8: Your code here. (2/2) */
    tlbwi
.set reorder
```

kern/tlbex.c中的_do_tlb_refill函数

根据Hints来写就行。

> /* Hints:
>
> * Invoke 'page_lookup' repeatedly in a loop to find the page table entry '*ppte'
>
> * associated with the virtual address 'va' in the current address space 'cur_pgdir'.
>
> *
>
> * **While** 'page_lookup' returns 'NULL', indicating that the '*ppte' could not be found,
>
> * allocate a new page using 'passive_alloc' until 'page_lookup' succeeds.
>
> */

```
while(1) {
        if (page_lookup(cur_pgdir, va, &ppte) == NULL) {
            passive_alloc(va, cur_pgdir, asid);
        } else {
            break;
        }
    }
```

kern/tlb_asm.S中的do_tlb_refill函数

```
/* Hint: use 'tlbwr' to write CP0.EntryHi/Lo into a random tlb entry. */
    /* Exercise 2.10: Your code here. */
tlbwr
```

三、实验体会

本次实验总体而言难度很大，主要是因为教程晦涩难懂而且要补全的代码大多需要使用许多前面定义的宏或者函数，这导致在写的时候需要耗费大量时间去了解这个宏或函数的用法，同时也需要注意一些细节，容易在细节上出错。