Lab 2 Report

21371295 张昊翔

思考题

Thinking 2.1

虚拟地址。虚拟地址是由操作系统分配给进程的，它将物理内存抽象为一个连续的地址空间，使得每个进程都可以独立地使用它们的地址空间，而不需要担心其他进程的地址空间冲突问题。

Thinking 2.2

通过宏实现链表，可以将链表的结构和操作封装在一起。这意味着如果我们需要在程序中多次使用链表，我们只需要在每个链表的头文件中包含链表的宏定义即可，而不需要在每个链表中分别定义链表节点和链表操作函数。

- 单向链表

在单向链表中，每个节点只有一个指针指向下一个节点，因此在插入和删除节点时，需要修改两个节点的指针，即当前节点和其前驱节点。**删除节点**的操作性能差，因为需要**遍历整个链表**找到待删除节点的前驱节点，而插入节点只需要修改指针即可。

- 循环链表

循环链表与单向链表类似，只是在链表的尾部将指向 NULL 的指针改为指向链表的头部。在循环链表中，插入和删除节点的操作与单向链表性能相同，但是对于**尾节点**和**头节点**的操作比较复杂，因为它们不再有 NULL 指针。

- 双向链表

双向链表中每个节点有两个指针，一个指向前驱节点，一个指向后继节点。因此，在双向链表中插入和删除节点时，只需要修改当前节点、前驱节点和后继节点的指针即可。相比单向链表，双向链表在删除节点时更加高效，因为可以通过当前节点找到前驱节点，而不需要遍历整个链表查找前驱节点。

Thinking 2.3

C。**页控制块**Page结构体可展开为：

```
struct Page {
    LIST_ENTRY(Page) pp_link; /* free list link */
    u_short pp_ref;
};
```

由LIST_HEAD和LIST_ENTRY宏定义可得。

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

Thinking 2.4

ASID(Address Space Identifier)是一个用于区分不同进程或线程**虚拟地址空间**的标识符。

- 避免地址冲突: ASID可以确保每个进程或线程拥有独立的虚拟地址空间，避免了不同进程或线程之间的地址冲突。
- 提高TLB的效率：如果不使用ASID，TLB需要存储所有的虚拟地址和物理地址的映射关系，这样会浪费TLB的空间和降低效率。使用ASID，TLB只需要存储每个进程或线程的虚拟地址和物理地址的映射关系，从而提高了TLB的效率。

ASID占据EntryHi寄存器的11-6位，因此可容纳2^6=64个不同的地址空间。

Thinking 2.5

`tlb_invalidate`调用`tlb_out`，其作用是更新TLB，删除特定**虚拟地址**在TLB中的旧表项。

```
LEAF(tlb_out)
.set noreorder
    mfc0    t0, CP0_ENTRYHI         # 将Hi存入t0
    mtc0    a0, CP0_ENTRYHI         # 将旧表项的Key存入Hi
    nop
    tlbp                            # 根据Hi中Key查找TLB中表项，将其索引存入Index
    nop

    mfc0    t1, CP0_INDEX           # 将Index存入t1
.set reorder
    bltz    t1, NO_SUCH_ENTRY       # t1<0, TLB中没有Key对应的表项
.set noreorder
    mtc0    zero, CP0_ENTRYHI       # Hi = 0
    mtc0    zero, CP0_ENTRYLO0      # Lo = 0
    nop
    tlbwi                           # 将Hi和Lo写入索引指定TLB表项，即Key和Data均清零

.set reorder

NO_SUCH_ENTRY:
    mtc0    t0, CP0_ENTRYHI         # 恢复Hi
    j       ra
END(tlb_out)
```

Thinking 2.6

在内存管理机制上，MIPS主要采用的是分页存储管理，X86主要采用的是**段页式**管理，地址空间首先被分为若干**逻辑段**，然后将每段分为若干大小固定的页。
X86中内存被分为三种形式，分别是逻辑地址(Logical Address)，线性地址(Linear Address)和物理地址(Physical Address)。通过分段可以将逻辑地址转换为线性地址，而通过分页可以将线性地址转换为物理地址。

难点分析

2.1 mips_detect_memory()

npage = memsize / 页大小，可在mmu.h中发现页大小为4096(BY2PG)B，即4KB。

2.2 LIST_INSERT_AFTER()

LIST_ENTRY(type)结构十分重要，它包含指向下一个元素的指针le_next，以及指向前一个元素链表项的**le_next**的指针le_prev。

```
struct {
    struct type *le_next;  /* next element */
    struct type **le_prev; /* address of previous next element */
}
```

2.3 page_init()

- page_free_list的类型是结构体Page_list

```
#define LIST_INIT(head)
    do {
        ((head)->lh_first) = NULL;
    } while (0)
```

- `PADDR(freemem)` 宏返回kseg0中**虚拟地址**freemem对应的物理地址
- `alloc` 函数返回初始虚拟地址给pages
- `LIST_INSERT_HEAD(head, elm, field)` 宏将elm插到链表head头部，链表项pp_link为field

2.4 page_alloc()

`LIST_EMPTY(&page_free_list)` 返回true代表链表头为NULL，**空闲链表**没有可用页。

- `page2pa(struct Page *pp)` 页控制块 -> 物理地址
- `page2kva(struct Page *pp)` 页控制块 -> kseg0中虚拟地址

注意memset使用方式: `void *memset(void *dst, int c, size_t n)`

2.5 page_free()

page_decref函数中调用page_free，将**空闲页面**回收。

2.6 pgdir_walk()

使ppte指向**二级页表项**的指针

- typedef u_long Pde: 一级页表/页目录(Page Directory)
- typedef u_long Pte: 二级页表/页表(Page Table)
- `PTE_ADDR(pte)` 获取页表项中的物理地址

虚拟地址共32位:

- 31-22: PDX(va)
- 21-12: PTX(va)
- 11-0: 页内偏移量

(*pgdir_entryp & PTE_V) == 0: 二级页表不存在
*pgdir_entryp = page2pa(pp) | PTE_D | PTE_V : 设置页表权限

2.7 page_insert()

将**虚拟地址**va映射到页控制块pp对应的物理页面，并将页表项权限设置为perm

- `tlb_invalidate` 删除虚拟地址在TLB中的映射
- `page_lookup` 返回**页控制块**，并使ppte指向**二级页表项**的指针

2.8 tlb_out

TLB表项为64位，高32位是Key，低32位是Data，分别对应CP0中**寄存器**EntryHi和EntryLo。

2.9 _do_tlb_refill()

根据虚拟地址和ASID查找页表，返回包含**物理地址**的页表项 cur_pgdir存储了**一级页表**基地址对应的kseg0中**虚拟地址**，加上偏移可**直接**访问二级页表。

2.10 do_tlb_refill

v0中存放**物理地址**，因此放进EntryLo。 `tlbwr` 指令将EntryHi与EntryLo随机写入一个TLB表项。

实验体会

本次实验主要涉及操作系统中的**内存管理**，由于我对虚拟地址与分页存储管理的机制很不熟悉，在实验时遇到了一些困难。我对此的解决方法是回归理论，先理解内存管理的大致思路与宏观架构，再着眼于实验中具体的代码实现。实验与理论的学习进度互相促进有助于我对操作系统整体知识的掌握。