操作系统 Lab2实验报告

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` 使用的也是**虚拟地址**。

Thinking 2.2

- 用宏来实现链表省去了变量类型的限制，而可以直接通过变量名进行访问和操作，从而实现了类似于其它语言中"泛型"的效果，提升了代码的可重用性。
- 实验环境中的 `/usr/include/sys/queue.h` 文件内对单向链表的实现中，有关元素插入的宏只有 `SLIST_INSERT_AFTER(slistelm, elm, field)` （将 `elm` 插到已有元素 `slistelm` 之后）和 `LIST_INSERT_HEAD(head, elm, field)` （将 `elm` 插到头部结构体 `head` 对应链表的头部），因此可直接实现的插入操作比较少。而对于双向链表的实现，其表头结构体存储了第一个元素和最后一个元素的地址，因此在实现 `CIRCLEQ_INSERT_TAIL(head, elm, field)` 时省去了查找尾结点的循环，效率会比较高。

Thinking 2.3

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

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

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

- 正确的 `Page_list` 展开结构为C。

Thinking 2.4

- `boot_pgdir_walk(Pde *pgdir, u_long va, int create)` 函数在 `boot_map_segment(Pde *pgdir, u_long va, u_long size, u_long pa, int perm)` 函数中被调用，用来获取虚拟地址所对应二级页表项的地址。

```
void boot_map_segment(Pde *pgdir, u_long va, u_long size, u_long pa, int perm)
{
    int i, va_temp;
    Pte *pgtable_entry;

    /* Step 1: Check if `size` is a multiple of BY2PG. */
    size = ROUND(size, BY2PG);

    /* Step 2: Map virtual address space to physical address. */
    /* Hint: Use `boot_pgdir_walk` to get the page table entry of virtual address `va`. */
    for (i = 0; i < size; i += BY2PG) {
        pgtable_entry = boot_pgdir_walk(pgdir, va + i, 1);
        *pgtable_entry = (pa + i) | perm | PTE_V;
    }
    *(pgdir + PDX(va)) = *(pgdir + PDX(va)) | perm | PTE_V;
}
```

```
git@20373785:~/20373785$ grep -rn boot_pgdir_walk
mm/pmap.c:90:static Pte *boot_pgdir_walk(Pde *pgdir, u_long va, int create)
mm/pmap.c:137:  /* Hint: Use `boot_pgdir_walk` to get the page table entry of virtual address `va`. */
mm/pmap.c:139:          pgtable_entry = boot_pgdir_walk(pgdir, va + i, 1);
mm/pmap.c:287:This function has something in common with function `boot_pgdir_walk`.*/
```

- `boot_map_segment(Pde *pgdir, u_long va, u_long size, u_long pa, int perm)` 函数在 `mips_vm_init()` 函数中被调用，用来将物理地址和虚拟地址进行映射。

```
void mips_vm_init()
{
    extern char end[];
    extern int mCONTEXT;
    extern struct Env *envs;

    Pde *pgdir;
    u_int n;

```

```
    /* Step 1: Allocate a page for page directory(first level page table). */
    pgdir = alloc(BY2PG, BY2PG, 1);
    printf("to memory %x for struct page directory.\n", freemem);
    mCONTEXT = (int)pgdir;

    boot_pgdir = pgdir;

    /* Step 2: Allocate proper size of physical memory for global array `pages`,
     * for physical memory management. Then, map virtual address `UPAGES` to
     * physical address `pages` allocated before. In consideration of alignment,
     * you should round up the memory size before map. */
    pages = (struct Page *)alloc(npage * sizeof(struct Page), BY2PG, 1);
    printf("to memory %x for struct Pages.\n", freemem);
    n = ROUND(npage * sizeof(struct Page), BY2PG);
    boot_map_segment(pgdir, UPAGES, n, PADDR(pages), PTE_R);

    /* Step 3, Allocate proper size of physical memory for global array `envs`,
     * for process management. Then map the physical address to `UENVS`. */
    envs = (struct Env *)alloc(NENV * sizeof(struct Env), BY2PG, 1);
    n = ROUND(NENV * sizeof(struct Env), BY2PG);
    boot_map_segment(pgdir, UENVS, n, PADDR(envs), PTE_R);

    printf("pmap.c:\t mips vm init success\n");
}
```

```
git@20373785:~/20373785$ grep -rn boot_map_segment
mm/pmap.c:128:void boot_map_segment(Pde *pgdir, u_long va, u_long size, u_long pa, int perm)
mm/pmap.c:173:  boot_map_segment(pgdir, UPAGES, n, PADDR(pages), PTE_R);
mm/pmap.c:179:  boot_map_segment(pgdir, UENVS, n, PADDR(envs), PTE_R);
include/pmap.h:101:void boot_map_segment(Pde *pgdir, u_long va, u_long size, u_long pa, int perm);
```

Thinking 2.5

- 由于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，因此需要 `ASID` 来标识唯一进程，为当前进程提供空间保护，避免在切换进程时因清空TLB而浪费过多资源。《IDT R30xx Family Software Reference Manual》中相关描述如下：

> If the TLB mechanism wasnot supported with an ASID, when the OS switches from one task to another, it would have to find and invalidate all TLB translations relating to the old task's address space, to prevent them from being erroneously used for the new one. This would be desperately inefficient.

- 根据《IDT R30xx Family Software Reference Manual》中的描述：

> Instead, the OS assigns a 6-bit unique code to each task's distinct address space.
>
> ……
>
> Since the ASID is only 6 bits long, OS software does have to lend a hand if there are ever more than 64 address spaces in concurrent use; but it probably won't happen too often. In such a system, new tasks are assigned new ASIDs until all 64 are assigned; at that time, all tasks are flushed of their ASIDs "de-assigned" and the TLB flushed; as each task is re-entered, a new ASID is given. Thus, ASID flushing is relatively infrequent.

`ASID` 段有6位，因此最多可以同时标识64个进程，若有更多进程需要分配 `ASID` 码时，当前所有 `ASID` 码都会被重新分配，TLB也会被刷新，每次重新进入任务时，都会给出一个新的 `ASID` 。

Thinking 2.6

- 在 `mm/pmap.c` 中 `tlb_invalidate` 调用了 `tlb_out` 。

```
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));
    }
}
```

- `tlb_invalidate` 用来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。
-

```
#include <asm/regdef.h>
#include <asm/cp0regdef.h>
#include <asm/asm.h>

LEAF(tlb_out)
//1: j 1b
nop
    mfc0    k1,CP0_ENTRYHI      // 把 CP0_ENTRYHI 原有值存储到 $k1 中
    mtc0    a0,CP0_ENTRYHI      // 把 $a0 中值存放到 CP0_ENTRYHI 中
                                // 此时 CP0_ENTRYHI 存放了虚拟地址空间及其标志位
    nop
    // insert tlbp or tlbwi
    tlbp                        // 根据 EntryHi 中的 Key（包含 VPN 与 ASID）查找 TLB 中与之对应的表项
                                // 如果有则把匹配项的 index 保存到 Index 寄存器中
                                // 没有匹配则置 Index 的最高位为 1
    nop                         // 流水线阻塞等待 tlbp 指令执行完成
    nop
    nop
    nop
    mfc0    k0,CP0_INDEX        // 把 CP0_ENTRYHI 中的新值存储到 $k0 中
    bltz    k0,NOFOUND          // 如果 $k0 中值小于 0，即 CP0_INDEX 最高位置 1
                                // 表示 TLB 缺失，跳转到 NOFOUND
    nop
    mtc0    zero,CP0_ENTRYHI    // 清空 CP0_ENTRYHI 和 CP0_ENTRYLO0
    mtc0    zero,CP0_ENTRYLO0
    nop
    // insert tlbp or tlbwi
    tlbwi                       // 更新 TLB，以 Index 寄存器中的值为索引
                                // 将此时 EntryHi 与 EntryLo 的值写到索引指定的 TLB 表项中
NOFOUND:
    mtc0    k1,CP0_ENTRYHI      // 把 $k1 中值存放到 CP0_ENTRYHI 中，还原 TLB
    j   ra
```

```
33      nop
34  END(tlb_out)
```

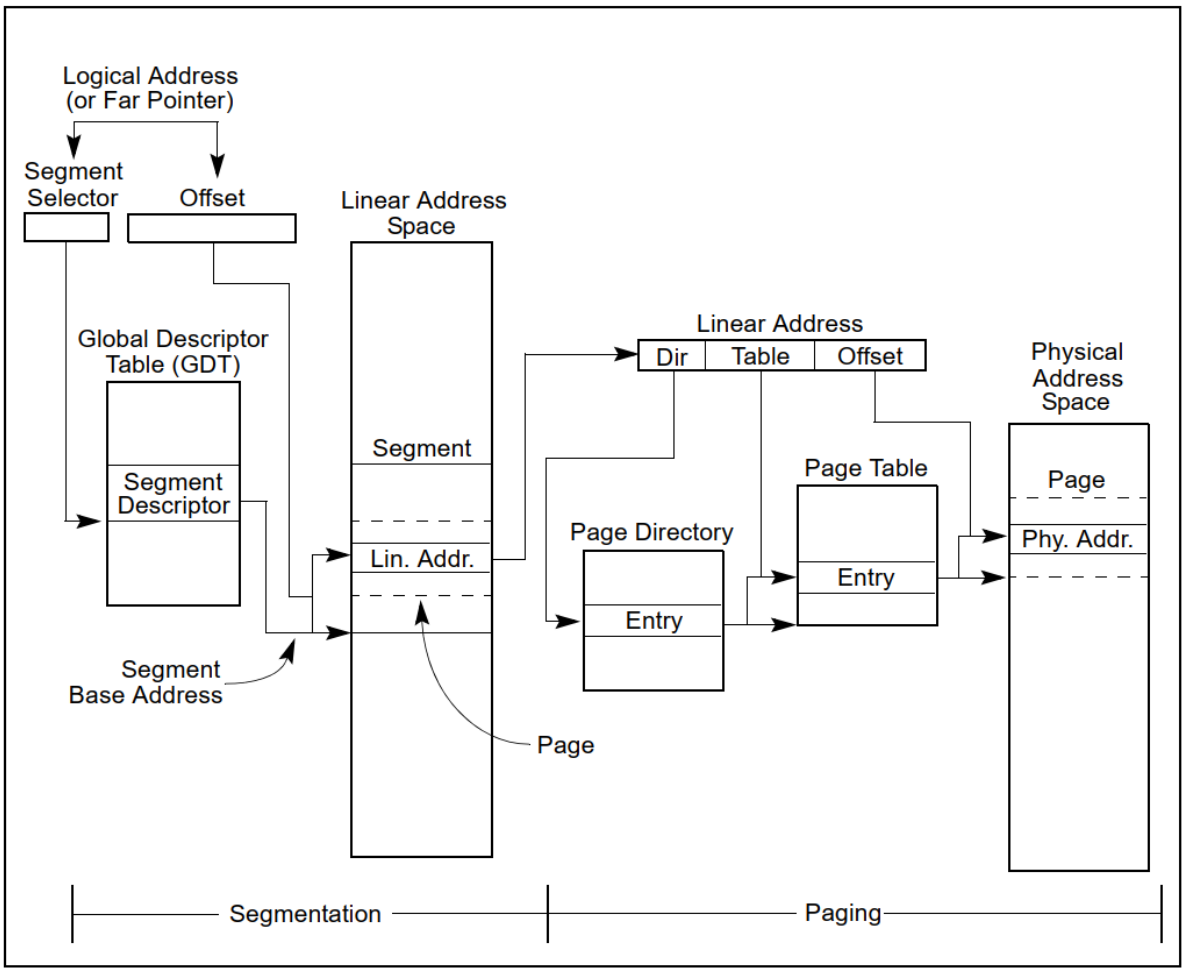
Thinking 2.7

- $PD_{base} = PT_{base} + (PT_{base} >> 9) + (PT_{base} >> 18) + (PT_{base} >> 27)$
- $PD_{base} = PT_{base} + (PT_{base} >> 9) + (PT_{base} >> 18) + (PT_{base} >> 27) + (PT_{base} >> 27)$

Thinking 2.8

x86体系结构内存管理机制

在 x86 体系结构中，内存管理机制与 MIPS 体系结构大体相同，虚拟地址到物理地址的转换流程如下图：



通过逻辑地址访问全局描述符表（GDT），每个段描述包含段基址、段长度、属性，由此获得线性地址空间的段地址，进而获得线性地址。再根据线性地址访问页目录和页表，从而可以索引到物理地址，访问物理内存空间。

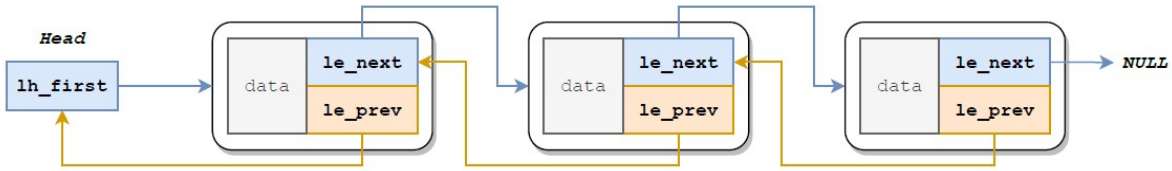
x86与MIPS内存管理比较

两者的差别在于对 TLB 不命中时的处理上：

- MIPS 在 TLB 不命中时会触发 TLB Refill 异常，内核的 tlb_refill_handler 会以 pgd_current 为当前进程的 PGD 基址，索引获得转换失败的虚址对应的 PTE，并将其填入 TLB，之后 CPU 再把刚刚转换失败的虚地址再填入 TLB 中。
- 而 X86 在 TLB 不命中时，由硬件 MMU 以 CR3 为当前进程的 PGD 基址，索引获得 PFN 后，直接输出 PA，同时 MMU 会填充 TLB 以加快下次转换的速度。
- 另外转换失败的虚址，MIPS 使用 BadVAddr 寄存器存放，X86 使用 CR2 存放。

实验难点

宏定义链表



初次接触实验中的链表结构时花费了很多时间来理解其实现过程，尤其是 `le_prev` 的二级指针设计，`include/queue.h` 中的注释对此进行了解释：

```
The le_prev points at the pointer to the structure containing
this very LIST_ENTRY, so that if we want to remove this list entry,
we can do *le_prev = le_next to update the structure pointing at us.
```

在移出链表项时可以直接使用 `*le_prev = le_next` 减少一次访存，提高程序性能。

虚拟内存管理

- 在完成虚拟内存管理部分的函数时，需要熟练运用已定义的宏和函数，避免重复造轮子。

mmu.h

宏定义	作用
BY2PG	页面大小（4KB = 4096Byte）
PDMAP	每个页目录项映射的字节大小（4MB）
PGSHIFT	虚拟地址中二级页表号的右移位数（12位）
PDSHIFT	虚拟地址中页目录号的右移位数（22位）
PDX(va)	虚拟地址va的页目录索引（31~22位）
PTX(va)	虚拟地址va的页表索引（21~12位）
PTE_ADDR(pte)	获取页表项上的物理地址（清空低12位的标识位，后面会说）
PPN	物理页号（因为每个页面大小4KB, 所以右移12位）
VPN	虚拟页号（与PPN相同）
VA2PFN	虚拟地址转换为物理页号（清空低12位）
PTE_G~PTE_LIBRARY	页表的标识位（有效位、dirty bit、uncached等）
PADDR(kva)	内核虚拟地址kva对应的物理地址，就是减去 `0x80000000`
KADDR(pa)	物理地址pa对应的内核虚拟地址，加上 `0x80000000`

queue.h

定义了一系列的宏函数来简化对链表的操作。

宏函数	作用
`LIST_EMPTY(head)`	判断链表是否为空
`LIST_FIRST(head)`	获取链表的表头
`LIST_FOREACH(var, head, field)`	for循环遍历链表
`LIST_INIT(head)`	初始化链表表头为NULL
`LIST_INSERT_AFTER(listelm, elm, field)`	在 `listelm` 后面插入 `elm`
`LIST_INSERT_BEFORE(listelm, elm, field)`	在 `listelm` 前面插入 `elm`
`LIST_INSERT_HEAD(head, elm, field)`	头插法插入 `elm`
`LIST_INSERT_TAIL(head, elm, field)`	尾插法插入 `elm`
`LIST_NEXT(elm, field)`	`elm` 的 `next` 指针
`LIST_REMOVE(elm, field)`	删除链表中的 `elm`

pmap.h

相关定义	作用
Page	结构体名，用于表示相应物理内存页的信息
Page_list	成员为Page的结构体名
Page_LIST_entry_t	结构体名，用于定义链表的指针域
pp_link	结构体变量，同时又是Page的成员，为Page_LIST_entry_t类型，它是链表的指针域
page_free_list	结构体变量，为Page_list类型，用于管理空闲物理页面
pages	相当于是个数组，数组的一项(可称为页指针) 都是Page类型， 表示相应物理内存页的信息(每一项都对应着一个物理页)
page2ppn	将页指针转化为物理页号
page2pa	将页指针转化物理地址(下面有说明)
pa2page	由物理地址pa得到其对应的页指针
page2kva	将页指针转化虚拟地址(先转为物理地址，再转为虚拟地址)
va2pa	将虚拟地址转化为物理地址（通过二级页表）

> 图片摘自mooc评论区

- 与 TLB 有关的几个重要的寄存器：
  - `EntryHi`：CP0寄存器号为10，包含了输入的关键字 VPN2 和记录当前活动的 ASID
  - `EntryLo0-1`：CP0寄存器号为2和3，包含了输出用到的关键字，以及标志位
  - `PageMask`：CP0寄存器号为5，配置映射页的大小
  - `Index`：CP0寄存器号为0，指定要写入或者读取的 TLB 表项
  - `Random`：CP0寄存器号为1，这是一个随机数寄存器，被 `tlbwr` 用来将一个新的 TLB 表项写入到一个随机的位置
  - `BadVAddr`：CP0寄存器号为8，记录最近一次导致tlb或寻址错误例外的虚拟地址

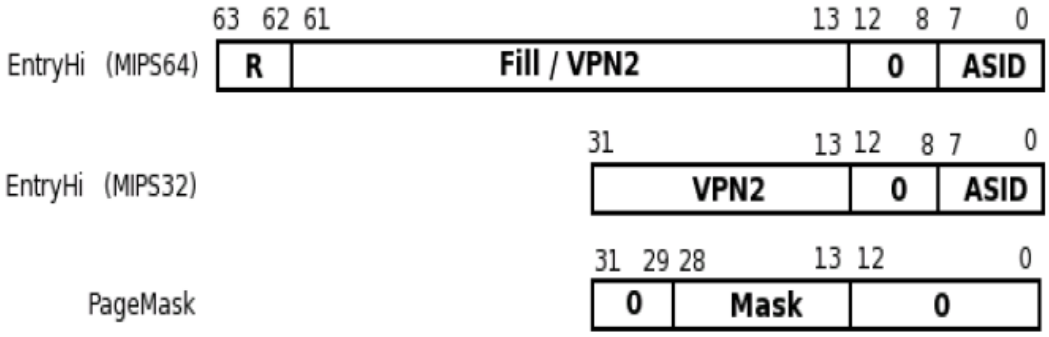


图 6.2: EntryHi 和 PageMask 寄存器域

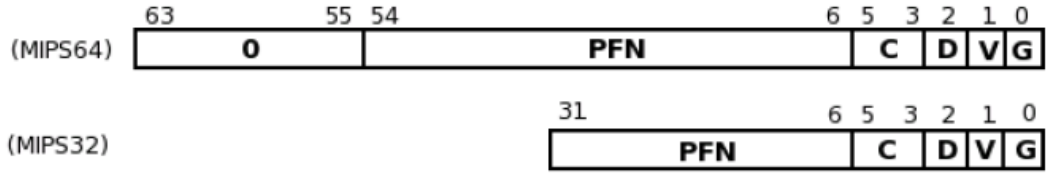


图 6.3: EntryLo0–1 寄存器域

体会感想

lab2课下内容总计编写和调试代码大概用时10个小时左右，但其背后有许多知识需要花费更多时间来进行理解和消化。特别是虚拟内存管理的部分，需要将页表的工作流程烂熟于心，才能够流畅顺利地读懂代码，写对代码，否则就需要来回反复于理论知识和实验代码之间。

本次实验相较于之前的Lab在体量和难度上都有了很大提升，相较于实际编写代码，阅读代码和理解代码所花费的时间会多得多。同时，实验对于理论知识的要求也逐渐提高，包括上学期计组课上所学的TLB相关内容，以及c语言中宏定义和指针的使用。这也对我之后的学习提出了更高的要求和挑战。

指导书 bug

启动流程

> - Step 1，`ROUND(a, n)` 是一个定义在 include/types.h 的宏，它的作用是返回 `[a/n]*n`（这里的中括号表示向上取整），要求 `n` 必须是 2 的非负整数次幂，因此 `align` 也必须是一个非负整数次幂。这行代码的含义即找到最小的符合条件的初始虚拟地址，中间未用到的地址空间全部放弃。

其中一句应为“因此 `align` 也必须是一个**2的**非负整数次幂”。