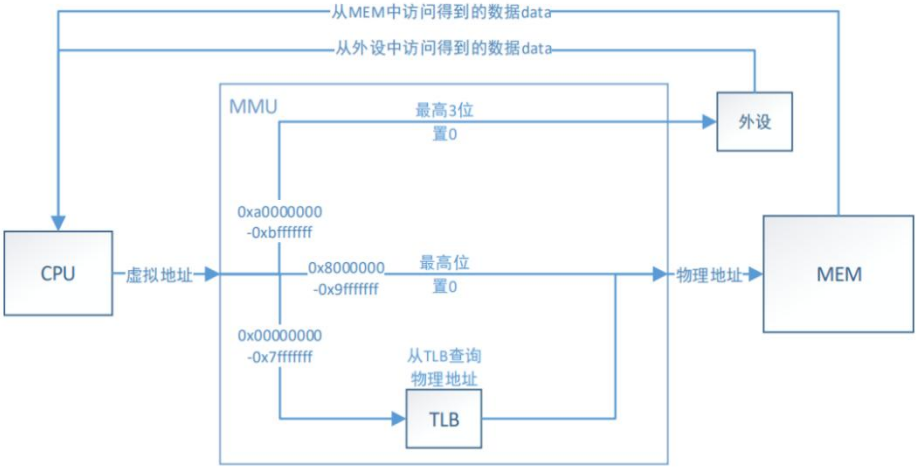
操作系统 LAB2 实验报告

210617 21373037 杨博文

一．思考题

1.在编写的 C 程序中，指针变量中存储的地址是虚拟地址，还是物理地址？MIPS 汇编程序中 lw 和 sw 使用的是虚拟地址，还是物理地址？

代码在 CPU 内执行时得到的都是虚拟地址。



2. 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。

用宏实现链表，主要是可以模拟出 C 语言不具有的泛型功能。因为宏的实现方式是字符串替换，替换时不会考虑到语法等等问题，因此只要给好参数，则任何结构体的链表均能由一组宏统一实现，简单便捷。

插入操作(考虑先寻找，再插入)

双向链表:头部插入 O(1),尾部插入 O(n)，指定节点前 O(1),指定节点后 O(1)

单向链表:头部插入 O(1),尾部插入 O(n)，指定节点前 O(n),指定节点后 O(1)

双向循环链表:头部插入 O(1),尾部插入 O(1)，指定节点前 O(1),指定节点后 O(1)

删除操作:

双向链表：头部节点删除 O(1)，尾部节点删除 O(n)，指定节点删除 O(1)

单向链表：头部节点删除 O(1)，尾部节点删除 O(n)，指定节点删除 O(n)

双向循环链表：头部节点删除 O(1)，尾部节点删除 O(1)，指定节点删除 O(1)

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

选 C,Page_list 是 Page 结构体的头指针,里面包含一个指向 Page 结构体的指针即 lh_first,Page 结构体由引用次数 pp_ref 和链表项

pp_link 组成，故选 C。

4. 请阅读上面有关 R3000-TLB 的描述，从虚拟内存的实现角度，阐述 ASID 的必要性。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。

ASID 称作是地址空间，每个用户进程有一个专属的 ASID 码，用来区分不同进程。由于每个进程都有自己的虚拟地址空间且相同，在 MIPS 上都是 0-4G,因此进程间会使用到相同的虚拟地址，但实际上他们对应的是不同的物理地址，若此刻不增加标志位来区分不同进程，TLB 便分不出来不同的进程，出现严重错误，当然也可以每切换一次进程刷新所有 TLB 重新装填，但是这样效率会大大降低。

ASID 共六位，一共最多容纳 64 个地址空间。

5. 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_out 代码如下：

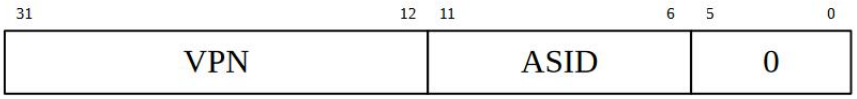
```
#include <asm/asm.h>

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
        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

NO_SUCH_ENTRY:
        mtc0    t0, CP0_ENTRYHI
        j       ra
END(tlb_out)

```

```
/* Overview:
 *   Invalidate the TLB entry with specified 'asid' and virtual address 'va'.
 *
 * Hint:
 *   Construct a new Entry HI and call 'tlb_out' to flush TLB.
 *   'tlb_out' is defined in mm/tlb_asm.S
 */
void tlb_invalidate(u_int asid, u_long va) {
        tlb_out(PTE_ADDR(va) | (asid << 6));
}
```

tlb_invalidate 代码如下：

tlb_out 被 tlb_invalidate 调用，tlb_invalidate 的作用是删除特定(即特定 ASID 码)虚拟地址在 TLB 中的旧表项。

PTE_ADDR 作用是清空 va 虚拟地址后十二位偏移位，接着与位移到 6 到 11 位的 ASID 码进行拼接，生成 TLB 的 Key，如下图。



EntryHi Register (TLB Key Fields)

TLB 相关指令

- tlbr: 以 Index 寄存器中的值为索引，读出 TLB 中对应的表项到 EntryHi 与 EntryLo。
- tlbwi: 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。
- tlbwr: 将 EntryHi 与 EntryLo 的数据随机写到一个 TLB 表项中（此处使用 Random 寄存器来“随机”指定表项，Random 寄存器本质上是一个不停运行的循环计数器）。
- tlbp: 根据 EntryHi 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）。

结合相关指令,按行解读 tlb_out 代码：

mfc0 t0, CP0_ENTRYHI：将原 EntryHi 寄存器内容取出

mtc0 a0, CP0_ENTRYHI：将 tlb_invalidate 向其传递的参数 a0(这一步在编译器种执行)写进 EntryHi 寄存器

tlbp：根据 EntryHi 中的 Key 查找对应的 Value，并将查到的表项索引存入 Index 寄存器

mfc0 t1, CP0_INDEX：将 Index 寄存器值存入 t1 寄存器

bltz t1, NO_SUCH_ENTRY：判断 t1 是否为负数，若为负数，意味着无对应表项，也就不需要清空，跳转到 NO_SUCH_ENTRY。若不为负数，意味着查到表项，继续执行。

mtc0 zero, CP0_ENTRYHI：将 EntryHi 清空

mtc0 zero, CP0_ENTRYLO0：将 EntryLow 清空

tlbwi：根据上面 Index 查到的索引，用清空后的 Hi 和 Low 对相应表项清空，完成表项无效化操作。

6. 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）。**

首先正如二级页表结构里页目录的自映射一样，三级页表结构中，512 张二级页表一定有一张正是一级页表，而每张二级页表对应的 512 张三级页表也一定有一张正是二级页表，因此总共有 512*512*512=128M 页表项，每个页表项对应着 4KB 大小的页面，共 512GB,没问题。

三级映射要经历三步。PTbase 对应的页表项是第 PTbase>>12 个，而每个页表有 512 个页目录项，每个页表项的大小是 8B（页面大小 4KB，有 512 个页表项），所以 PTbase 在三级页表区中对应的偏移量是 PTbase>>9。记作二级页表项基地址。

然后，三级页表区（1G）映射到二级页表区（512*512=256K 个 8B 页表项，2M大小）。每个页表项映射了 4K 的空间，则二级页表区基地址对应的页表项是第(PTbase>>9)>>12 个，所以二级页表区基地址

在二级页表区中对应的偏移量是(PTbase>>9)>>9。记作一级页表基地址。

最后，二级页表区（2M）映射到一级页表（512 个 8B页表项，4K 大小）。每个页表项映射了 4K 的空间，则一级页表基地址对应的页表项是第(PTbase>>18)>>12 个，所以一级页表基地址在一级页表自身中对应的偏移量是(PTbase>>18)>>9。

那么，三级页表页目录（即一级页表）的基地址为 PTbase + (PTbase>>9) + (PTbase>>18)

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

7. 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。

x86 架构的内存管理机制分为两部分：分段机制和分页机制。 而 Mips 架构是单纯的分页机制。

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。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。X86 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。

针对分段机制而言，分段相当于完成了虚拟地址到线性地址的转化，x86 架构提供了两种段描述符表：GDT（全局段描述符表 Global

Descriptor Table）和 LDT （本地段描述符表 Local Descriptor Table）。其中 GDT 描述系统段，包括操作系统本身；LDT 描述局部于每个系统的段，包括其代码、数据、堆栈等，可以将 GDT 当作一级段表，LDT 看作二级段表。当需要访问一个段时，需要将该段的选择子装进段寄存器中，对应的描述符从 GDT 或 LDT 取出。

若开启分页机制，则将得到的线性地址解释成页目录偏移+二级页表偏移+页内偏移，像是纯分页机制中的“虚拟地址”继续通过页表得到实际地址。若不开启分页机制，则得到的线性地址就是物理地址。

- 80386之后的处理器共有6个段选择子，
  - CS寄存器：程序指令段起始地址；
  - DS寄存器：程序数据段起始地址；
  - SS寄存器：栈起始地址；
  - ES, FS, GS寄存器：额外段寄存器。

段选择符结构

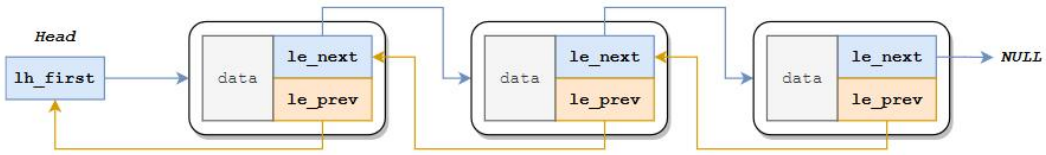
TI（加载指示）：值为0处理器从GDT中加载；1则处从LDT中加载。
RPL（请求优先级）：00最高，11最低。

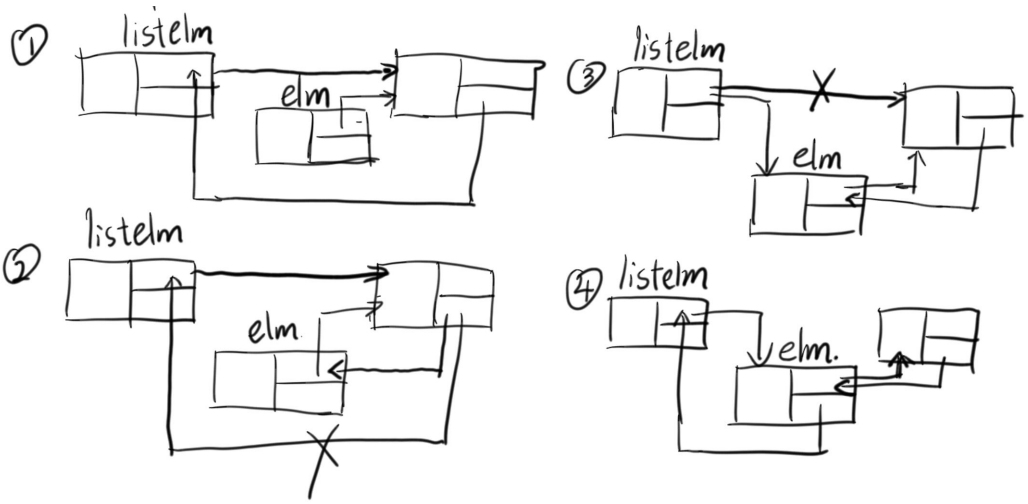
二. 实验难点

本次实验重点在于对内存管理机制的初始化工作，对物理内存分页，并对页面实现申请与删除，并完成了一些与页表相关的函数。我认为 pmap.c 中页表函数是本次实验的难点，接下来我将依次梳理一

下各个函数的用途与实现思路，并总结这些函数搭配使用产生的最终效果。

首先是物理页面的创建与构建空闲页面双向链表，难点在于链表宏的实现，需要用到数据结构的相关知识，前插后插需要分四步，每一步的顺序至关重要，以后插为例，elm 为待插入元素，listelm 为当前元素，先判断 listelm 后是否还有元素，若无则直接尾插即可，若有则如下图四步执行(画的略难看)，特别要注意指针的指针存的是指针的地址，即 le_prev 存的是 le_next 的地址，应使用 le_prev = &le_next。在构建页面时应按 Page 结构体物理地址从低到高与对应的物理页面地址从低到高对应。





接下来是对几个页表函数的理解。

Page_walk：

```c
static int pgdir_walk(Pde *pgdir, u_long va, int create, Pte **ppte) {
	Pde *pgdir_entryp;
	struct Page *pp;

	/* Step 1: Get the corresponding page directory entry. */
	/* Exercise 2.6: Your code here. (1/3) */
	pgdir_entryp = PDX(va) + pgdir;
	/* Step 2: If the corresponding page table is not existent (valid) and parameter `create`
	 * is set, create one. Set the permission bits 'PTE_D | PTE_V' for this new page in the
	 * page directory.
	 * If failed to allocate a new page (out of memory), return the error. */
	/* Exercise 2.6: Your code here. (2/3) */
	if ((*pgdir_entryp & PTE_V)  == 0 && create == 1) {   //valid = 0 不合法
		if(page_alloc(&pp) == -E_NO_MEM) {
			return -E_NO_MEM;
		}
		*pgdir_entryp = page2pa(pp) | PTE_D | PTE_V;
		pp->pp_ref = 1;
		*ppte = PTX(va) + (Pte *)page2kva(pp);
		//*ppte = PTX(va) + (Pte *)KADDR(PTE_ADDR(*pgdir_entryp));
	} else if ((*pgdir_entryp & PTE_V) == 0 && create == 0) {
		*ppte = NULL;
	} else if ((*pgdir_entryp & PTE_V) != 0) { //valid = 1 合法
		*ppte = PTX(va) + (Pte *)page2kva(pages + PPN(*pgdir_entryp)); //第xx个物理页面的地址
		//*ppte = PTX(va) + (Pte *)KADDR(PTE_ADDR(*pgdir_entryp));
	}
	/* Step 3: Assign the kernel virtual address of the page table entry to '*ppte'. */
	/* Exercise 2.6: Your code here. (3/3) */
```

给定虚拟地址，查找页目录，若能查到相应的二级页表，将 va 虚拟地址所在的二级页表项的指针存储在 ppte 指向的空间上，若没查到，则建立二级页表，完成对页表页面的初始化，即引用次数置为 1，后续操作同。

Page_insert：

```
Pte *pte;

/* Step 1: Get corresponding page table entry. */
pgdir_walk(pgdir, va, 0, &pte);

if (pte && (*pte & PTE_V)) {
        if (pa2page(*pte) != pp) {
                page_remove(pgdir, asid, va);
        } else {
                tlb_invalidate(asid, va);
                *pte = page2pa(pp) | perm | PTE_V;
                return 0;
        }
}

/* Step 2: Flush TLB with 'tlb_invalidate'. */
/* Exercise 2.7: Your code here. (1/3) */
tlb_invalidate(asid, va);
/* Step 3: Re-get or create the page table entry. */
/* If failed to create, return the error. */
/* Exercise 2.7: Your code here. (2/3) */
if (pgdir_walk(pgdir, va, 1, &pte) == -E_NO_MEM) {
        return -E_NO_MEM;
}
/* Step 4: Insert the page to the page table entry with 'perm | PTE_V' and increase its
 * 'pp_ref'. */
/* Exercise 2.7: Your code here. (3/3) */
*pte = page2pa(pp) | perm | PTE_V;
pp->pp_ref++;
return 0;
```

将一级页表基地址 pgdir 对应的两级页表结构中虚拟地址 va 映射到页控制块 pp 对应的物理页面，若 va 在页表中无相应二级页表则使用 page_walk 创建，若 va 在二级页表中对应的物理页面跟本次设置的不同，则需要清空 tlb 并重写二级页表，最后将对应的物理页面号填到二级页表当中，并设置权限。

Page_look:

```c
struct Page *page_lookup(Pde *pgdir, u_long va, Pte **ppte) {
        struct Page *pp;
        Pte *pte;

        /* Step 1: Get the page table entry. */
        pgdir_walk(pgdir, va, 0, &pte);

        /* Hint: Check if the page table entry doesn't exist or is not valid. */
        if (pte == NULL || (*pte & PTE_V) == 0) {
                return NULL;
        }

        /* Step 2: Get the corresponding Page struct. */
        /* Hint: Use function `pa2page`, defined in include/pmap.h . */
        pp = pa2page(*pte);
        if (ppte) {
                *ppte = pte;
        }

        return pp;
}
```

作用是返回一级页表基地址 *pgdir* 对应的两级页表结构中虚拟地址 *va* 映射的物理页面的页控制块，同时将 *ppte* 指向的空间设为对应的二级页表项地址，若无则返回 *null*，主要是用来配合 *tlb* 重填函数。

这一切的一切当发生缺页异常时，*OS* 将执行如下步骤。

1. 从 BadVAddr 中取出引发 TLB 缺失的虚拟地址。

2. 从 EntryHi 的 6－11 位取出当前进程的 ASID。

3. 以虚拟地址和 ASID 为参数，调用 _do_tlb_refill 函数。该函数是 TLB 重填过程的核心，其功能是根据虚拟地址和 ASID 使用 look 函数查找页表，若查找不到则会利用页面申请函数来申请一个物理页面，并使用 insert 函数将物理页面号存入页表之中，在 insert 和 look 的过程中又会调用 walk 函数查找二级页表项存不存在，直至 look 函数返回不为 null。

4. 将物理地址存入 EntryLo ，并执行 tlbwr 将此时的 EntryHi 与 EntryLo 写入到 TLB 中。

以上函数间的调用关系便非常清晰明了，且我们发现这些页表函数统一服务于 tlb 重填操作，在重填的操作过程中，完成了页表的建立。

三．实验感受

在 Lab2 中，我们首次接触到了如此大型的程序，头文件 C 文件汇编文件应有尽有无一缺席，各种文件之间的配合也是令人眼花缭乱。在笔者拿到之初，一度无从下手，令人不禁翻看往届学长的代码，不过最后及时收手，想着还是自己先挑战挑战。最让人记忆犹新的是写 page_walk，page_insert 那个夜晚，因为 create 的逻辑问题，debug 一度 de 到破防，甚至在怀疑是不是真出现了以前 Lab 写错的情况发生，de 到在主楼沙发上撒泼打滚，好在我的朋友乐于助人，在一个个文件的来回切换中，最终锁定了问题，当时已然临近 11 点。好在第三部分的内容简单，Lab2 实验部分在高潮之后很快便迎来尾声。不过想要理清内存管理的逻辑有些困难，想必有许多同学在写代码时只是单纯的根据提示去写，没有思考内在的逻辑，笔者也是在之后又继续翻阅指导书与代码才大致弄清逻辑。

总的来说本次实验是一次具有挑战感与收获感的实验，随着一遍遍阅读内核代码，笔者对于 OS 的原理也是愈发清晰，也愈发明白为什么说写 OS 的程序员都是最优秀的程序员了。

Lab3 冲冲冲！