Lab2 实验报告

周美廷 | 76066002

# **思考2.1**

请思考 cache 用虚拟地址来查询的可能性，并且给出这种方式对访存带来的好处和坏处。另外，你能否能根据前一个问题的解答来得出用物理地址来查询的优势?

物理地址是唯一的。虚拟地址进程和进程之间是不一样的。用虚拟地址查要求每个进程有一个cache，显然不现实。共用cache就有查不准的问题。所以所有号称虚拟地址查询的都是虚拟地址的几个位做为hint预查询，最终还是要物理地址确认是否真的hit。

# **思考2.2**

请查阅相关资料，针对我们提出的疑问，给出一个上述流程的优化版本，新的版本需要有更快的访存效率。（提示：考虑并行执行某些步骤）

# **思考2.3**

在我们的实验中，有许多对虚拟地址或者物理地址操作的宏函数 (详 见include/mmu.h),那么我们在调用这些宏的时候需要弄清楚需要操作的地址是物 理地址还是虚拟地址，阅读下面的代码，指出 x 是一个物理地址还是虚拟地址。

int x;

char\* value = return\_a\_pointer();

\*value = 10;

x = (int) value;

x是虚拟地址，因为下面层还有操作系统负责把虚拟地址转换成物理地址，允许修改存储中某个部分。

# **思考2.4**

我们注意到我们把宏函数的函数体写成了 do { /\* ... \*/ } while(0) 的形式，而不是仅仅写成形如 { /\* ... \*/ } 的语句块，这样的写法好处是什么？

由于宏展开是编译器在编译时进行的一定程度上对源码的处理替换，它可以便于程序的控制，但错误的宏展开可能导致错误的运行结果或者是编译失败。为了保证宏展开后的原子性和适配能力，需要对宏进行良好的设计。对于宏函数，习惯上会在函数的末尾加上一个分号，如果使用题述中后者的写法，则会在写到else语句之前的位置时出现编译错误，原因是多余一个分号使编译器无法分析，改用题述中前者的写法恰好可以成功适配末尾的分号。

# **思考2.5**

注意，我们定义的 Page 结构体只是一个信息的载体，它只代表了相应物理内存页的信息，它本身并不是物理内存页。那我们的物理内存页究竟在哪呢？Page 结构体又是通过怎样的方式找到它代表的物理内存页的地址呢？请你阅读include/pmap.h 与 mm/pmap.c 中相关代码，给出你的想法。

物理内存页是在page\_free\_list里获取的，Page结构体是通过pmap.c中的page\_alloc函数找到它代表的物理内存页的地址。

# **思考2.6**

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 (请注意指针)。

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;

}

正确的展开方式是C：

struct Page\_list{

struct {

struct {

struct Page \*le\_next;

struct Page \*\*le\_prev;

} pp\_link;

u\_short pp\_ref;

}\* lh\_first;

}

参考函数/结构体：

struct Page {

Page\_LIST\_entry\_t pp\_link;

u\_short pp\_ref;

};

#define LIST\_HEAD(name, type) \

struct name { \

struct type \*lh\_first; /\* first element \*/ \

}

#define LIST\_ENTRY(type) \

struct { \

struct type \*le\_next; /\* next element \*/ \

struct type \*\*le\_prev; /\* address of previous next element \*/ \

}

# **思考2.7**

在 mmu.h 中定义了bzero(void \*b, size\_t)这样一个函数, 请你思 考，此处的 b 指针是一个物理地址，还是一个虚拟地址呢？

b是虚拟地址（要被清空的虚拟地址）。

# **思考2.8**

了解了二级页表页目录自映射的原理之后，我们知道，Win2k 内核的虚存管理也是采用了二级页表的形式，其页表所占的 4M 空间对应的虚存起始地址为 0xC0000000，那么，它的页目录的起始地址是多少呢？

由于是二级页表映射，实际将 4GB 寻址空间划分为 1M 个页表，而 0xC0000000 对应的是第 (0xC0000000 >> 12)=0x000C0000 个页表，由于需要建立一级页目录到二级页表中的自映射，而每个页表项占 4 个字节，故第 0x000C0000 个页表的第 0x000C0000 项的起始地址是 0xC0000000+(0x000C0000 << 2)=0xC00300000 ，因此页目录的第一项的起始地址是 0xC00300000 。

# **思考2.9**

思考一下 tlb\_out 汇编函数，结合代码阐述一下跳转到 NOFOUND 的流程？从 MIPS 手册中查找 tlbp 和 tlbwi 指令，明确其用途，并解释为何第 10 行处指令后有 4 条 nop 指令。

TLB数据项使用了CP0的EntryHi,PageMask,EntryLo0,EntryLo1这四个寄存器，其中EntryHi包含VPN2（虚拟页号），EntryLo0，EntryLo1分别包含一个PFN（页帧号），tlb的作用也就是根据虚拟页号寻找物理页帧号。

CP0 的寄存器 Index(0号) 中的值决定了相应的指令要读的是哪一个TLB表项，取值从0一直到表项的总数减一。用tlbp软件搜索一个TLB表项时，Index也会自动重置。

Index的低位保存TLB的索引。这不需要很多位，因为还没有MIPS CPU有超过128项的TLB。最高位（位31）有特殊意义，当检测未能找到匹配项的时候，由tlbp置为1。位31选的很好，因为这样看起来像负值，很容易测试。

而我们在tlb\_out中也是这么干的，下面我分析一下关键汇编指令的作用，也就是跳转到

NOFOUND的流程：

mfc0 k1,CP0\_ENTRYHI # CP0的CP0\_ENTRYHI寄存器->GPR[K1]，这个寄存器包含两个域，VPN2(虚拟页号)和ASID(地址空间标识符）；这条命令将存在中CP0的指定寄存器的虚拟页号存入GPR[K1]中，为tlb处理做准备。

mtc0 a0,CP0\_ENTRYHI # 将GPR[a0]中的值存入CP0的ENTRYHI寄存器,而GPR[a0]按照约定存的是调用tlb\_out函数时传入的参数

tlbp # 软件搜索tlb表项，会自动设置CP0寄存器Index的值,并且如果未找到匹配项的话，Index的最高位会设置为1，也就是负数。

mfc0 k0,CP0\_INDEX #将CP0\_INDEX寄存器中的值装入GPR[k0]

bltz k0,NOFOUND #如果GPR[k0] < 0，也就是CP0\_INDEX < 0,也就是tlbp未能成功找到虚拟页号对应的页帧号,跳转到NOFOUND

以上资料均参照了 see MIPS run linux，以MIPS32为准，MIPS64有些许不同.

补充关于以下3条指令的理解：

mtc0 zero,CP0\_ENTRYHI # 将CP0\_ENTRYHI寄存器置为0

mtc0 zero,CP0\_ENTRYLO0 # 将CP0\_ENTRYLO0寄存器置为0

tlbwi # write TLB entry at index

这3条指令会在TLB命中的情况下执行,此时CP0\_INDEX存储着相应的TLB项，前两条指令将CP0\_ENTRYHI和CP0\_ENTRYLO0置为0，最后一条指令将其写入index对应的TLB项，也就是将这一项清0，实现tlb\_out的功能，如果虚拟地址va对应的虚拟页号页存在于TLB，废止va对应的TLB项。

# **思考2.10**

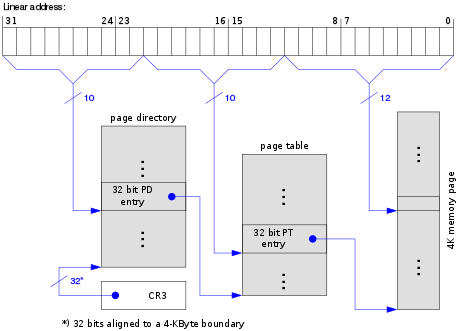
显然，运行后结果与我们预期的不符，va 值为 0x88888，相应的 pa中的值为 0。这说明我们的代码中存在问题，请你仔细思考我们的访存模型，指出问题所在。

参考PADDR的宏函数可看到pa的值等于kva-ULIM，而题目中pa+ULIM相当于获取kva的值而不是pa的

# **思考2.11**

在 X86 体系结构下的操作系统，有一个特殊的寄存器 CR4，在其中 有一个 PSE 位，当该位设为 1 时将开启 4MB 大物理页面模式，请查阅相关资料， 说明当 PSE 开启时的页表组织形式与我们当前的页表组织形式的区别。

“传统“4 KiB分页



使用PSE进行4 MiB分页