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？

- C指针变量存储：虚拟地址
- lw/sw 使用：虚拟地址

Thinking 2.2

> 从可重用性的角度，阐述用宏来实现链表的好处。

- 能将宏函数封装，降低代码耦合度，便于修改，提高代码可移植性。
- 宏在执行时会被展开替换，无需进行调用，减少空间开销，提高代码效率。
- 使用宏无需考虑指针引用以及跳转问题。

> 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。

- 单向链表实现

```
/*
 * Singly-linked List functions.
 */
#define SLIST_HEAD(name, type)                                          \
struct name {                                                           \
        struct type *slh_first; /* first element */                     \
}

#define SLIST_HEAD_INITIALIZER(head)                                    \
        { NULL }

#define SLIST_ENTRY(type)                                               \
struct {                                                                \
        struct type *sle_next;  /* next element */                      \
}

/*
 * Singly-linked List functions.
 */
#define SLIST_INIT(head) do {                                           \
        (head)->slh_first = NULL;                                       \
} while (/*CONSTCOND*/0)

#define SLIST_INSERT_AFTER(slistelm, elm, field) do {                   \
        (elm)->field.sle_next = (slistelm)->field.sle_next;             \
        (slistelm)->field.sle_next = (elm);                             \
} while (/*CONSTCOND*/0)

#define SLIST_INSERT_HEAD(head, elm, field) do {                        \
        (elm)->field.sle_next = (head)->slh_first;                      \
```

```
        (head)->slh_first = (elm);                                      \
} while (/*CONSTCOND*/0)

#define SLIST_REMOVE_HEAD(head, field) do {                             \
        (head)->slh_first = (head)->slh_first->field.sle_next;          \
} while (/*CONSTCOND*/0)

#define SLIST_REMOVE(head, elm, type, field) do {                       \
        if ((head)->slh_first == (elm)) {                               \
                SLIST_REMOVE_HEAD((head), field);                       \
        }                                                               \
        else {                                                          \
                struct type *curelm = (head)->slh_first;                \
                while(curelm->field.sle_next != (elm))                  \
                        curelm = curelm->field.sle_next;                \
                curelm->field.sle_next =                                \
                    curelm->field.sle_next->field.sle_next;             \
        }                                                               \
} while (/*CONSTCOND*/0)

#define SLIST_FOREACH(var, head, field)                                 \
        for((var) = (head)->slh_first; (var); (var) = (var)->field.sle_next)

/*
 * Singly-linked List access methods.
 */
#define SLIST_EMPTY(head)       ((head)->slh_first == NULL)
#define SLIST_FIRST(head)       ((head)->slh_first)
#define SLIST_NEXT(elm, field)  ((elm)->field.sle_next)
```

- 循环链表实现

```
/*
 * Circular queue functions.
 */
#define CIRCLEQ_INIT(head) do {                                         \
        (head)->cqh_first = (void *)(head);                             \
        (head)->cqh_last = (void *)(head);                              \
} while (/*CONSTCOND*/0)

#define CIRCLEQ_INSERT_AFTER(head, listelm, elm, field) do {            \
        (elm)->field.cqe_next = (listelm)->field.cqe_next;              \
        (elm)->field.cqe_prev = (listelm);                              \
        if ((listelm)->field.cqe_next == (void *)(head))                \
                (head)->cqh_last = (elm);                               \
        else                                                            \
                (listelm)->field.cqe_next->field.cqe_prev = (elm);      \
        (listelm)->field.cqe_next = (elm);                              \
} while (/*CONSTCOND*/0)

#define CIRCLEQ_INSERT_BEFORE(head, listelm, elm, field) do {           \
        (elm)->field.cqe_next = (listelm);                              \
        (elm)->field.cqe_prev = (listelm)->field.cqe_prev;              \
        if ((listelm)->field.cqe_prev == (void *)(head))                \
                (head)->cqh_first = (elm);                              \
```

```
        else                                                            \
                (listelm)->field.cqe_prev->field.cqe_next = (elm);      \
        (listelm)->field.cqe_prev = (elm);                              \
} while (/*CONSTCOND*/0)

#define CIRCLEQ_INSERT_HEAD(head, elm, field) do {                      \
        (elm)->field.cqe_next = (head)->cqh_first;                      \
        (elm)->field.cqe_prev = (void *)(head);                         \
        if ((head)->cqh_last == (void *)(head))                         \
                (head)->cqh_last = (elm);                               \
        else                                                            \
                (head)->cqh_first->field.cqe_prev = (elm);              \
        (head)->cqh_first = (elm);                                      \
} while (/*CONSTCOND*/0)

#define CIRCLEQ_INSERT_TAIL(head, elm, field) do {                      \
        (elm)->field.cqe_next = (void *)(head);                         \
        (elm)->field.cqe_prev = (head)->cqh_last;                       \
        if ((head)->cqh_first == (void *)(head))                        \
                (head)->cqh_first = (elm);                              \
        else                                                            \
                (head)->cqh_last->field.cqe_next = (elm);               \
        (head)->cqh_last = (elm);                                       \
} while (/*CONSTCOND*/0)

#define CIRCLEQ_REMOVE(head, elm, field) do {                           \
        if ((elm)->field.cqe_next == (void *)(head))                    \
                (head)->cqh_last = (elm)->field.cqe_prev;               \
        else                                                            \
                (elm)->field.cqe_next->field.cqe_prev =                 \
                    (elm)->field.cqe_prev;                              \
        if ((elm)->field.cqe_prev == (void *)(head))                    \
                (head)->cqh_first = (elm)->field.cqe_next;              \
        else                                                            \
                (elm)->field.cqe_prev->field.cqe_next =                 \
                    (elm)->field.cqe_next;                              \
} while (/*CONSTCOND*/0)

#define CIRCLEQ_FOREACH(var, head, field)                               \
        for ((var) = ((head)->cqh_first);                               \
                (var) != (const void *)(head);                          \
                (var) = ((var)->field.cqe_next))

#define CIRCLEQ_FOREACH_REVERSE(var, head, field)                       \
        for ((var) = ((head)->cqh_last);                                \
                (var) != (const void *)(head);                          \
                (var) = ((var)->field.cqe_prev))

/*
 * Circular queue access methods.
 */
#define CIRCLEQ_EMPTY(head)             ((head)->cqh_first == (void *)
(head))
#define CIRCLEQ_FIRST(head)             ((head)->cqh_first)
#define CIRCLEQ_LAST(head)              ((head)->cqh_last)
```

```
#define CIRCLEQ_NEXT(elm, field)        ((elm)->field.cqe_next)
#define CIRCLEQ_PREV(elm, field)        ((elm)->field.cqe_prev)

#define CIRCLEQ_LOOP_NEXT(head, elm, field)                             \
        (((elm)->field.cqe_next == (void *)(head))                      \
            ? ((head)->cqh_first)                                       \
            : (elm->field.cqe_next))
#define CIRCLEQ_LOOP_PREV(head, elm, field)                             \
        (((elm)->field.cqe_prev == (void *)(head))                      \
            ? ((head)->cqh_last)                                        \
            : (elm->field.cqe_prev))
```

- 插入：
  - 双向链表：
    - INSERT_HEAD: O(1)
    - INSERT_BEFORE: O(1)
    - INSERT_AFTER: O(1)
    - INSERT_TAIL: O(n) 无尾节点指针，需要遍历
  - 单向链表：
    - INSERT_HEAD: O(1)
    - INSERT_BEFORE: O(n) 无前向指针，需要遍历
    - INSERT_AFTER: O(1)
    - INSERT_TAIL: O(n) 无尾节点指针，需要遍历
  - 单向循环链表：
    - INSERT_HEAD: O(1)
    - INSERT_BEFORE: O(n) 无前向指针，需要遍历
    - INSERT_AFTER: O(1)
    - INSERT_TAIL: O(1)
- 删除
  - 双向链表:
    - REMOVE: O(1)
  - 单向链表:
    - REMOVE: O(n)
  - 单向循环链表:
    - REMOVE: O(n)

相比较于单向链表与单向循环链表，双向链表的插入和删除操作效率一般更高。

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。

- 选择C

Thinking 2.4

> 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。

- ASID 可用来唯一标识进程，为进程提供地址空间保护。每一个 TLB 表项会有一个 ASID，标识这个表项是属于哪一个进程，CP0_EntryHi 中的 ASID 是当前进程的 ASID，当进程对 TLB 的查询操作，即使 VPN 命中，但若该表项不是 global 且 ASID 与 CP0_EntryHi 的 ASID 不一致，则也是访问缺少，从而实现保护。
- ASID 允许 TLB 同时包含多个进程的条目。如果 TLB 不支持独立的 ASID，每次选择一个页表时（例如，上下文切换时），TLB 就必须被冲刷（flushed）或删除，以确保下一个进程不会使用错误的地址转换。所以有 ASID 就不用每次切换进程都要 flush 所有 TLB。

> 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。

- TLB 通过设置进程的 ASID 和 EntryLo G 位，匹配时需要同时匹配地址、ASID且为 global 。 这允许软件同时映射 **64 个**不同的地址空间，而无需操作系统在上下文更改时清除 TLB。（参考如下）

  *"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."*

Thinking 2.5

> tlb_invalidate 和 tlb_out 的调用关系?

- tlb_invalidate 函数内部调用 tlb_out

> 请用一句话概括 tlb_invalidate 的作用。

- tlb_invalidate 函数调用 tlb_out() 将虚拟地址 va 对应的 TLB 页表项清空，使其失效

> 逐行解释 tlb_out 中的汇编代码。

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

LEAF(tlb_out) // 定义叶函数
.set noreorder
    mfc0    t0, CP0_ENTRYHI // 先将CP0_ENTRYHI中原有的值写入t0寄存器
    mtc0    a0, CP0_ENTRYHI // 将待清空表项的key写到CP0_ENTRYHI中
    nop
    /* Step 1: Use 'tlbp' to probe TLB entry */
    /* Exercise 2.8: Your code here. (1/2) */
    tlbp // 根据EntryHi中的Key，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，则 Index最高位被置1
    nop
    /* Step 2: Fetch the probe result from CP0.Index */
    mfc0    t1, CP0_INDEX // 取出CP0_INDEX的值
.set reorder
    bltz    t1, NO_SUCH_ENTRY // 如果是Index最高位被置1，即未找到，跳转到NO_SUCH_ENTRY标签，否则为找到
.set noreorder
    mtc0    zero, CP0_ENTRYHI // 清空CP0_ENTRYHI
    mtc0    zero, CP0_ENTRYLO0 // 清空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 // 根据找到的Index将EntryHi与EntryLo的值写到索引对应的TLB表项中
.set reorder

NO_SUCH_ENTRY: // 没找到
    mtc0    t0, CP0_ENTRYHI // 将原来的key写回CP0_ENTRYHI
    j       ra // 返回调用函数处
END(tlb_out) // 函数结束
```

Thinking 2.6

> 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 架构中的内存管理机制基于**分段和分页**两种技术。具体来说，X86 中将物理内存地址划分为多个段（Segment）并给每个段指定权限，然后再将每个段分成大小相等的页（Page），每页大小通常是4KB或者2MB，这样可以实现虚拟内存的映射。

  在 X86 架构中，有两种模式：**实模式和保护模式**。在实模式下，所有的内存地址都是物理地址，没有虚拟地址的概念。而在保护模式下，X86 根据内存地址的不同进行了特权级的划分，即用户态和内核态，用户态只能访问自己被授权的内存空间，而内核态可以访问所有的内存空间。

  MIPS 架构使用基于**分页**的内存管理机制。MIPS 架构中虚拟地址空间被划分为大小相等的页（通常为4KB或8KB），同时也有一个页表来记录虚拟地址和物理地址之间的映射关系。MIPS 架构中没有类似于 X86 架构中的分段机制，因此所有的内存地址都是线性地址（Linear Address），不需要像 X86 一样进行虚拟地址到线性地址再到物理地址的转换。

> 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。

- RISC-V 支持大页（例如 2MB 和 1GB），而 MIPS 只支持 4KB 和 16MB 两种页面大小。

  RISC-V 还引入了 Sv39 和 Sv48 两种分页模式，它们分别支持 39 位和 48 位的虚拟地址空间。这使得 RISC-V 可以支持更大的虚拟地址空间，从而提供更大的可用内存。

  RISC-V 引入了虚拟地址空间的隔离，将虚拟地址空间分为多个地址空间，以实现内核空间和用户空间之间的隔离。

难点分析

链表

- 传入函数的是指向对象的指针
- le_prev指向的是前一个元素的le_next
- 插入时需要考虑是否为NULL

地址转换

- page2pa: page结构体指针 > page物理地址
- pa2page: page物理地址 > page结构体指针
- page2kva: page结构体指针 > 内核虚拟地址
- page2ppn: page结构体指针 > page页号
- PPN: 虚拟地址 > 页号 （即右移12位）
- PADDR: 内核虚拟地址 > 物理地址
- KADDR: 物理地址 > 内核虚拟地址

实验体会

这次实验让我进一步加深了对MIPS内存管理的理解，掌握了Cache、MMU和TLB之间的联系。

相较于前两次实验，这次实验的难度大大增加，主要体现在对各种工具函数和宏的掌握使用，对整体内存管理流程的分析。虽然完成了这次实验，但感觉似乎仍然存在一些疑问，可能还需要再学习下。

这次实验最困难的地方还是各种地址对象的转换，要弄清楚每一步需要哪种类型，还需要考虑到函数调用时传入的是对象还是指向对象的指针，还是指向对象的指针的指针，第一遍做的时候被绕晕了。