实验报告

一、实验目的

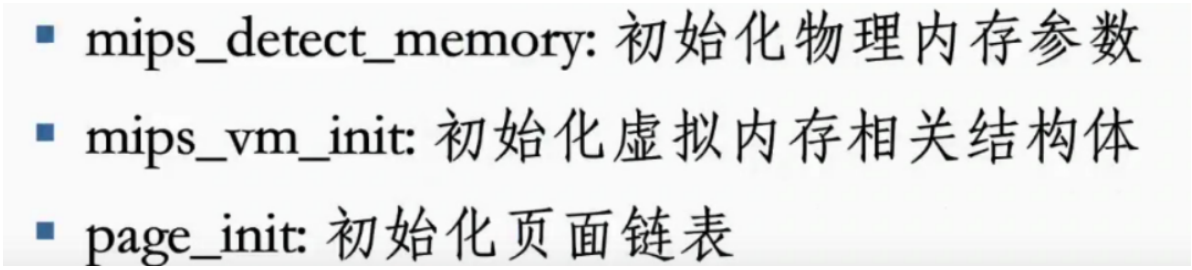
> **1. 了解内存访问原理**
>
> **2. 了解 MIPS 内存映射布局**
>
> **3. 掌握使用空闲链表的管理物理内存的方法**
>
> **4. 建立页表，实现分页式虚存管理**
>
> **5. 实现内存分配和释放的函数**
>
> 本次实验中，我们需要掌握 MIPS 页式内存管理机制，需要使用一些数据结构来记 录内存的使用情况，并实现内存分配和释放的相关函数，完成物理内存管理和虚拟内存 管理。

二、实验步骤

找到lab2的文件

首先`git pull`，然后如下：

```
jovyan@70234fb21137:~/ouc21020007131-1ab$ git branch -a
  1ab0
* 1ab1
  remotes/origin/1ab0
  remotes/origin/1ab0-result
  remotes/origin/1ab1
  remotes/origin/1ab1-result
  remotes/origin/1ab2
jovyan@70234fb21137:~/ouc21020007131-1ab$ ls
boot     include     lib       tools
drivers  include.mk  Makefile
gxemul   init        readelf
jovyan@70234fb21137:~/ouc21020007131-1ab$ git checkout 1ab2
Branch '1ab2' set up to track remote branch '1ab2' from 'origin'.
Switched to a new branch '1ab2'
jovyan@70234fb21137:~/ouc21020007131-1ab$ ls
boot     include     lib       readelf
drivers  include.mk  Makefile  tags
gxemul   init        mm        tools
```



思考

Thinking 2.1

> Thinking 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?

从原理上，将cache设置为用虚拟地址来查询是可行的，这就相当于将cache从页表的后方挪移到页表的前方。

- 好处：不需要使用`mmu`转换成物理地址就可以直接根据虚拟地址查找`cache`中的数据
- 坏处：不同的进程可能会有相同的虚拟地址，安全性和数据正确性很难保证，需要引入物理地址或者其他的标识符加以区分

若采用物理地址查询，优势即在于没有根据虚拟地址查询时需要区分不同的进程的问题，查询的安全性和数据正确性更有保证

Thinking 2.2

> Thinking 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)

访问`TLB`出现缺失时，要更新`TLB`，但在更新`TLB`时访问页表的过程中我们需要的物理地址就已经可以获得了，因此可以考虑并行更新`TLB`和查询`cache`

Thinking 2.3

> Thinking 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。

```
1 ^^Iint x;
2 ^^Ichar* value = return_a_pointer();
3 ^^I*value = 10;
4 ^^Ix = (int) value;
```

由代码得：x是虚拟地址

分析代码：在代码中，首先声明了一个整型变量x。然后，声明了一个指向字符类型的指针变量value，并通过函数return_a_pointer()获得了一个指向某个内存位置的指针。随后，通过解引用指针value，将该内存位置的值设为10。最后，value转换为整，并将转换后的结果赋值给变量x。由于value是指针类型，其转换为整型的结果相当于将指针的值直接赋给x。而指针的值是内存地址。

而C语言中指针所表示的地址均为虚拟地址。

Thinking 2.4

> Thinking 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？
> ■

1. 避免语法错误：使用 do-while(0) 形式可以避免一些语法错误。宏函数在展开时，会直接替换为宏定义中的代码，而不会检查语法。如果宏函数的函数体写成普通的代码块形式，当宏在使用时没有被包含在花括号中，或者使用了错误的语法，例如 if、else 等，就会导致编译器报错。而使用 do-while(0) 形式可以使得宏函数无论在何处使用，总是作为一个完整的语句出现，避免了这种语法错误。
2. 方便使用宏函数：使用 do-while(0) 形式的宏函数可以更方便地使用，尤其是在一些需要在 if-else 语句中使用宏函数时。由于 C/C++ 语法要求 if-else 结构中必须有一个完整的语句作为条件判断或执行块，在使用普通的代码块形式时，无法直接在 if 后面直接跟一个宏函数。而使用 do-(0) 形式的宏函数，可以通过在代码块中使用该宏函数，使宏函数作为一个完整的语句出现在 if-else 结构中。

Thinking 2.5

> Thinking 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。 ■

物理内存页存在RAM中，是在真实地物理内存里

物理内存页的地址的计算主要由两个函数实现：

1. 函数 `page2ppn(struct Page *pp)`：将指向结构体 Page 的指针 pp 转换为物理页号（Physical Page Number，`ppn`）。pp - pages 表示指针 pp 指向的结构体 Page 在 pages 数组中的索引位置。这里的 pages 是一个指向 `struct Page` 的数组，外部定义为 `extern struct Page *pages`。`ppn` 表示物页号，表示在 `pages` 数组中的索引位置，左移 `PGSHIFT` 位（`PGSHIFT` 是一个宏定义，表示页表偏移量的位数）即可得到物理地址。
2. 函数 `page2pa(struct Page *pp)`：通过调用 `page2ppn(struct Page *pp)` 函数获取物理页号 `ppn`，并将其左移 `PGSHIFT` 位，得到物理地址。

```
1  extern struct Page *pages;
2
3  static inline u_long
4  page2ppn(struct Page *pp)
5  {
6      return pp - pages;
7  }
8
```

```
/* Get the physical address of Page 'pp'.
 */
static inline u_long
page2pa(struct Page *pp)
{
    return page2ppn(pp) << PGSHIFT;
}
```

Thinking 2.6

> **Thinking 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;
}
```

可以看到，在`queue.h`中定义了以下结构体：

```
#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 */  \
        }
```

在`pmap.h`中定义了以下结构体：

```
struct Page {
	Page_LIST_entry_t pp_link;	/* free list link */

	// Ref is the count of pointers (usually in page table entries)
	// to this page.  This only holds for pages allocated using
	// page_alloc.  Pages allocated at boot time using pmap.c's
"alloc"
	// do not have valid reference count fields.

	u_short pp_ref;
};
```

则正确的的展开结构是C

```
1  struct Page_list{
2      struct{
3          struct{
4              struct Page * le_next;
5              struct Page ** e_prev;
6          } pp_link;
7          u_short pp_ref;
8      }* lh_first;
9  }
```

Thinking 2.7

> Thinking 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 指针是一个虚拟地址

在`mm/pmap.c`中的`alloc()`函数中，b的值为`alloced_men`

```
    if (clear) {
        bzero((void *)alloced_mem, n);
    }
```

而`alloced_men`的值为`freemen`

```
    alloced_mem = freemem;
```

`freemem`是一个静态变量，其初始化如下

```
16  struct Page *pages;
17  static u_long freemem;
18

 if (freemem == 0) {
     freemem = (u_long)end;
 }
```

而end找到在`scse0_3.lds`中，其值为`0x80400000`，这个值为我们在`Lab1`中加载内核之后设置的结束地址，是虚拟地址。

```
SECTIONS
{
  . = 0x80010000;
  .text : {
    *(.int)
    *(.text)
    *(.fini)
        }
  .data : {
    *(.data)
    }
  .bss  : {
    *(.bss)
    }
  .sdata : {
    *(.sdata)
    }
  . = 0x80400000;
  end = . ;
}
```

Thinking 2.8

> Thinking 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？ ■

页目录的起始地址为：`0xc0000000 + (0xc0000000 >> 12) * 4 =0xc0300000`

Thinking 2.9

> Thinking 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。 ■

流程：判断协处理器 0 `(Coprocessor 0)` 中的 `CP0_INDEX` 寄存器的值是否小于0，若小于0，则跳转到`NOFOUND`

- `tlbp` 指令：在 `TLB` 中搜索匹配的 `TLB` 项，并将匹配的结果保存在协处理器0中的指定寄存器中
- `tlbwi` 指令：将数据写入协处理器0中指定的 `TLB` 项，将数据写入协处理器0的`EntryHi`、`EntryLo0` 和 `EntryLo1` 寄存器中，并通过相关的 `TLB` 索引指定将数据写入的 `TLB` 项。

四条 `nop`：

`TLB`查找匹配的表项后会将索引写入`CP0_INDEX`寄存器，而下一条指令又需要读该寄存器，会产生写后读的冲突，所以需要等待四个时钟周期。

Thinking 2.10

> Thinking 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。 ■

`va2pa()` 函数的功能为返回虚拟地址的二级页表的物理地址，由于这里的虚拟地址不是`4K`的整数倍，因此得出的物理地址pa并非是该虚拟地址对应的实际的物理地址，而是这一物理页的首地址，所以读取这个地址的值并不能得出写入结果

Thinking 2.11

> Thinking 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。 ■

当开启大物理页面模式时，页目录的表项中增加了一个新的标识（第7位，又称PS位Page Size），如果PS = 1, 则这个页目录项指向一个4MB的大物理页面，而PS = 0，则这个页目录项指向一个二级页表。所以在PS = 1 时，虚拟地址的低22位全部用来在大物理页面中寻址。在PS = 0时，寻址方式与二级页表相同

练习

Exercise 2.1

> Exercise 2.1 在阅读 queue.h 文件之后，相信你对宏函数的巧妙之处有了更深的体会。请完成 queue.h 中的LIST_INSERT_AFTER函数和LIST_INSERT_TAIL函数。 ■

完成代码

```
//将一个元素插入到已有元素之后
#define LIST_INSERT_AFTER(listelm, elm, field) do { \
                LIST_NEXT((elm), field) = LIST_NEXT((listelm), field); \
                if (LIST_NEXT((listelm), field)) { \
                        LIST_NEXT((listelm), field)->field.le_prev = 
&LIST_NEXT((elm), field) ; \
                } \
                LIST_NEXT((listelm), field) = (elm); \
                (elm)->field.le_prev = &LIST_NEXT((listelm), field); \
        } while (0)
```

```
//将一个元素插入到链表尾部
#define LIST_INSERT_TAIL(head, elm, field) do { \
            if (LIST_FIRST((head)) != NULL) { \
                LIST_NEXT((elm), field) = LIST_FIRST((head)); \
                while (LIST_NEXT(LIST_NEXT((elm), field), field) != NULL) {
\
                    LIST_NEXT((elm), field) = LIST_NEXT(LIST_NEXT((elm),
field), field); \
                } \
                LIST_NEXT(LIST_NEXT((elm), field), field) = (elm); \
                (elm)->field.le_prev = &(LIST_NEXT(LIST_NEXT((elm), field),
field)); \
                LIST_NEXT((elm), field) = NULL; \
            } else { \
                LIST_INSERT_HEAD((head), (elm), field); \
            } \
    }while(0)
```

Exercise 2.2

`maxpa`：物理地址的最大值+1

`basemem`：物理内存对应的字节数

`npage`：总物理页数

`extmem`：扩展内存的大小

> Exercise 2.2 我们需要在 mips_detect_memory() 函数中初始化这几个全局变量，以确定内核可用的物理内存的大小和范围。根据代码注释中的提示，完成 mips_detect_memory() 函数。 ■

```
jovyan@70234fb21137:~/ouc21020007131-1ab$ gxemul -E testmips -C R3000 -M 64 gxemul/vmlin
GXemul 0.4.6    Copyright (C) 2003-2007  Anders Gavare
Read the source code and/or documentation for other Copyright messages.

Simple setup...
    net: simulating 10.0.0.0/8 (max outgoing: TCP=100, UDP=100)
        simulated gateway: 10.0.0.254 (60:50:40:30:20:10)
            using nameserver 192.168.224.14
    machine "default":
        memory: 64 MB
        cpu0: R3000 (I+D = 4+4 KB)
        machine: MIPS test machine
        loading gxemul/vmlinux
        starting cpu0 at 0x80010000
------------------------------------------------------------------------------
```

可以看到实验的系统内存为`64MB`，即`2^26`，且没有外存（则`extmem = 0`），而页面大小为`4KB`

完成初始化：

```
/* Overview:
    Initialize basemem and npage.
    Set basemem to be 64MB, and calculate corresponding npage value.*/

void mips_detect_memory()
{
    /* Step 1: Initialize basemem.

     * (When use real computer, CMOS tells us how many kilobytes there are).
*/
    maxpa = 1 << 26;
    basemem = 1 << 26;
    // Step 2: Calculate corresponding npage value.
    npage = basemem >> PGSHIFT;
    extmem = 0;

    printf("Physical memory: %dK available, ", (int)(maxpa / 1024));
    printf("base = %dK, extended = %dK\n", (int)(basemem / 1024),
           (int)(extmem / 1024));
}
```

Exercise 2.3

> Exercise 2.3 完成 page_init 函数，使用 include/queue.h 中定义的宏函数将未分配的物理页加入到空闲链表 page_free_list 中去。思考如何区分已分配的内存块和未分配的内存块，并注意内核可用的物理内存上限。 ■

- 初始化空闲物理页面链表
- 页对齐`freemem`变量
- 所有非空闲页面引用次数 +1

- 将其余空闲页面插入链表中

```
void
page_init(void)
{
    /* Step 1: Initialize page_free_list. */
    /* Hint: Use macro `LIST_INIT` defined in include/queue.h. */
    LIST_INIT(&page_free_list);

    /* Step 2: Align `freemem` up to multiple of BY2PG. */
    freemem = ROUND(freemem, BY2PG);
    int k = PADDR(freemem) / BY2PG;

    /* Step 3: Mark all memory blow `freemem` as used(set `pp_ref`
     * filed to 1) */
    int i = 0;
    while (i<k) {
        pages[i].pp_ref = 1;
        i++;
    }

    /* Step 4: Mark the other memory as free. */
    while (i < npage) {
        pages[i].pp_ref = 0;
        LIST_INSERT_HEAD(&page_free_list, &pages[i], pp_link);
        i++;
    }
}
```

Exercise 2.4

> Exercise 2.4 完成 mm/pmap.c 中的 page_alloc 和 page_free 函数，基于空闲内存链表 page_free_list ，以页为单位进行物理内存的管理。并在 init/init.c 的函数 mips_init 中注释掉 page_check();。此时运行结果如下。 ■

```
main.c: main is start ...
init.c: mips_init() is called
Physical memory: 65536K available, base = 65536K, extended = 0K
to memory 80401000 for struct page directory.
to memory 80431000 for struct Pages.
pmap.c:  mips vm init success
*temp is 1000
phycial_memory_manage_check() succeeded
panic at init.c:17: ----------------------------------------
```

`page_alloc`：实现新物理页面的申请

代码思路：将`page_free_list`空闲链表头部内存控制块对应的物理页面分配出去，将其从空闲链表中移除，并清空对应的物理页面，最后将 `pp` 指向的空间赋值为这个内存控制块的地址。

```
int
page_alloc(struct Page **pp)
{
    struct Page *ppage_temp;

    /* Step 1: Get a page from free memory. If fail, return the error
code.*/
    ppage_temp = LIST_FIRST(&page_free_list);
    if (ppage_temp == NULL) return -E_NO_MEM;
    /* Step 2: Initialize this page.
     * Hint: use `bzero`. */

    LIST_REMOVE(ppage_temp, pp_link);

    bzero((void *)page2kva(ppage_temp), BY2PG);

    *pp = ppage_temp;

    return 0;
}
```

`page_free`：实现物理页面的释放

代码思路：判断 `pp`指向内存控制块对应的物理页面引用次数是否为 `0`，是则该应物理页面为空闲页面，将其对应的内存控制块重新插入到`page_free_list`

```
void
page_free(struct Page *pp)
{
    /* Step 1: If there's still virtual address refers to this page, do
nothing. */
    if (pp->pp_ref > 0) {
        return;
    }

    /* Step 2: If the `pp_ref` reaches to 0, mark this page as free and
return. */
    else if (pp->pp_ref == 0) {
        LIST_INSERT_HEAD(&page_free_list, pp, pp_link);
    }

    /* If the value of `pp_ref` less than 0, some error must occurred
before,
     * so PANIC !!! */
    else
        panic("cgh:pp->pp_ref is less than zero\n");
}
```

此时，完成了物理内存部分

在`init/init.c`的函数`mips_init`中注释掉`page_check();`

`make`后运行`gxemul -E testmips -C R3000 -M 64 gxemul/vmlinux`命令进行测试

结果如下：

```
jovyan@74846d05e007:~/ouc21020007131-1ab$ gxemul -E testmips -C R3000 -M 64 gxemul/vmlinux
GXemul 0.4.6    Copyright (C) 2003-2007  Anders Gavare
Read the source code and/or documentation for other Copyright messages.

Simple setup...
    net: simulating 10.0.0.0/8 (max outgoing: TCP=100, UDP=100)
        simulated gateway: 10.0.0.254 (60:50:40:30:20:10)
            using nameserver 192.168.224.14
    machine "default":
        memory: 64 MB
        cpu0: R3000 (I+D = 4+4 KB)
        machine: MIPS test machine
        loading gxemul/vmlinux
        starting cpu0 at 0x80010000
--------------------------------------------------------------------------------

main.c: main is start ...

init.c: mips_init() is called

Physical memory: 65536K available, base = 65536K, extended = 0K

to memory 80401000 for struct page directory.

to memory 80431000 for struct Pages.

pmap.c:  mips vm init success

The number in address temp is 1000

physical_memory_manage_check() succeeded

panic at init.c:19: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

至此，我们的内核已经能够按照分页的方式对物理内存进行管理

Exercise 2.5

> Exercise 2.5 完成 mm/pmap.c 中的 boot_pgdir_walk 和 pgdir_walk 函数，实现虚拟地址到物理地址的转换以及创建页表的功能。 ■

`boot_pgdir_walk`函数：

- 获取相应的页目录条目和页表
- 如果对应的页表不存在，并且设置了create参数，则创建一个，并为这个新页表设置正确的权限位
- 获取 `va` 的页表条目，并返回它

```
1  static Pte *boot_pgdir_walk(Pde *pgdir, u_long va, int create)
2  {
3
4      Pde *pgdir_entryp;
5      Pte *pgtable, *pgtable_entry;
6
```

```
    /* Step 1: Get the corresponding page directory entry and page table. */
    /* Hint: Use KADDR and PTE_ADDR to get the page table from page directory
     * entry value. */
    pgdir_entryp = pgdir + PDX(va);
    pgtable = (Pte *)KADDR(PTE_ADDR(*pgdir_entryp));

    /* Step 2: If the corresponding page table is not exist and parameter `create`
     * is set, create one. And set the correct permission bits for this new page
     * table. */
    if(!(*pgdir_entryp & PTE_V)){
        if(create) {
            *pgdir_entryp = PADDR(alloc(BY2PG, BY2PG, 0)) | PTE_V;
            pgtable = (Pte *)KADDR(PTE_ADDR(*pgdir_entryp));
        }else{
            return 0;
        }
    }

    /* Step 3: Get the page table entry for `va`, and return it. */
    pgtable_entry = pgtable + PTX(va);
    return pgtable_entry;
}
```

`pgdir_walk`函数:

- 获取相应的页目录条目和页表
- 如果对应的页表不存在(有效)，并且设置了参数create，则创建一个。并为这个新页表设置正确的权限位。创建新页表时，可能内存不足
- 设置页表项为 `*ppte` 作为返回值

```
int
pgdir_walk(Pde *pgdir, u_long va, int create, Pte **ppte)
{
    Pde *pgdir_entryp;
    Pte *pgtable;
    struct Page *ppage;

    /* Step 1: Get the corresponding page directory entry and page table. */
    pgdir_entryp = pgdir + PDX(va);
    pgtable = KADDR(PTE_ADDR(*pgdir_entryp));

    /* Step 2: If the corresponding page table is not exist(valid) and parameter `create`
     * is set, create one. And set the correct permission bits for this new page
     * table.
     * When creating new page table, maybe out of memory. */
    if (!(*pgdir_entryp & PTE_V)) {
        if (create) {
            int ret = page_alloc(&ppage);
            if (ret < 0) {
                return ret;
```

```
            }
            ppage->pp_ref++;
            *pgdir_entryp = page2pa(ppage) | PTE_R | PTE_V;
            pgtable = KADDR(PTE_ADDR(*pgdir_entryp));
        }else{
            *ppte = 0;
            return 0;
        }
    }

    /* Step 3: Set the page table entry to `*ppte` as return value. */
    *ppte = pgtable + PTX(va);

    return 0;
}

```

Exercise 2.6

> Exercise 2.6 实现 mm/pmap.c 中的 boot_map_segment 函数，实现将制定的物理内存与虚拟内存建立起映射的功能。

- 检查 `size` 是否为 `BY2PG` 的倍数
- 将虚拟地址空间映射到物理地址
- 使用 `boot_pgdir_walk` 来获取虚拟地址 `va` 的页表项

```
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. */
    if (size % BY2PG != 0) {
        return;
    }

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

Exercise 2.7

> Exercise 2.7 完成 mm/pmap.c 中的 page_insert 函数。

- 获取相应的页表项
- 更新 `TLB` （更新页表必须使用 `tlb_invalidate` ）
- Do check，重新获取页表条目以验证插入
- 检查页面是否可以插入，如果不能返回 `-E_NO_MEM`
- 插入页面并增加 `pp_ref`

```
int
page_insert(Pde *pgdir, struct Page *pp, u_long va, u_int perm)
{
    u_int PERM;
    Pte *pgtable_entry;
    PERM = perm | PTE_V;

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

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

    /* Step 2: Update TLB. */
    tlb_invalidate(pgdir, va);
    /* hint: use tlb_invalidate function */
    /* Step 3: Do check, re-get page table entry to validate the insertion.
*/

    /* Step 3.1 Check if the page can be insert, if can't return -E_NO_MEM
*/
    if (pgdir_walk(pgdir, va, 1, &pgtable_entry) != 0) {
        return -E_NO_MEM;
    }
    /* Step 3.2 Insert page and increment the pp_ref */
    *pgtable_entry = (page2pa(pp) | PERM);
    pp->pp_ref++;

    return 0;
}
```

Exercise 2.8

> Exercise 2.8 完成 mm/tlb_asm.S 中 tlb_out 函数。 ■

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

LEAF(tlb_out)
//1: j 1b
nop
    mfc0    k1,CP0_ENTRYHI
    mtc0    a0,CP0_ENTRYHI
    nop
    // insert tlbp or tlbwi
    tlbp
    nop
    nop
    nop
    nop
    mfc0    k0,CP0_INDEX
    bltz    k0,NOFOUND
    nop
    mtc0    zero,CP0_ENTRYHI
    mtc0    zero,CP0_ENTRYLO0
    nop
    // insert tlbp or tlbwi
    tlbwi
NOFOUND:

    mtc0    k1,CP0_ENTRYHI

    j   ra
    nop
END(tlb_out)
```

至此，完成虚拟内存部分

在 `init/init.c` 的函数 `mips_init` 中，将 `page_cheack` 还原，并注释掉 `physical_memory_manage_check();`

重新`make`，运行命令 `gxemul -E testmips -C R3000 -M 64 gxemul/vmlinux`

结果如下：

```
---------------------------------------------------------------------------

main.c: main is start ...

init.c: mips_init() is called

Physical memory: 65536K available, base = 65536K, extended = 0K

to memory 80401000 for struct page directory.

to memory 80431000 for struct Pages.

pmap.c:  mips vm init success

va2pa(boot_pgdir, 0x0) is 3ffe000

page2pa(pp1) is 3ffe000

start page_insert

pp2->pp_ref 0

end page_insert

page_check() succeeded!

panic at init.c:19: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

提交 lab2

```
jovyan@74846d05e007:~/ouc21020007131-1ab$ git add .
jovyan@74846d05e007:~/ouc21020007131-1ab$ git config --global user.email "zym8004@stu.ouc.edu.cn"
jovyan@74846d05e007:~/ouc21020007131-1ab$ git config --global user.name "munume"
jovyan@74846d05e007:~/ouc21020007131-1ab$ git commit -m "lab2"
[lab2 95bcbd0] lab2
 16 files changed, 1102 insertions(+), 30 deletions(-)
 rewrite drivers/gxconsole/console.o (100%)
 create mode 100644 include/.ipynb_checkpoints/queue-checkpoint.h
 create mode 100644 init/.ipynb_checkpoints/init-checkpoint.c
 rewrite init/init.o (66%)
 create mode 100644 lib/env.o
 create mode 100644 lib/genex.o
 create mode 100644 mm/.ipynb_checkpoints/pmap-checkpoint.c
 create mode 100644 mm/.ipynb_checkpoints/tlb_asm-checkpoint.S
 create mode 100644 mm/pmap.o
 create mode 100644 mm/tlb_asm.o
jovyan@74846d05e007:~/ouc21020007131-1ab$ git push
Counting objects: 25, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (24/24), done.
Writing objects: 100% (25/25), 24.44 KiB | 3.05 MiB/s, done.
Total 25 (delta 11), reused 0 (delta 0)
remote: ************************************************************
remote:
remote:                     BUAA OSLAB AUTOTEST SYSTEM
remote:                   Copyright (c) BUAA 2015-2019
remote:
remote: ************************************************************
remote:
remote: [ You are changing the branch: refs/heads/lab2. ]
remote:
remote: Autotest: Begin at Fri Jan  5 20:29:13 CST 2024
remote:
```

```
remote: End build at Fri Jan  5 20:29:27 CST 2024
remote: [ PASSED:3 ]
remote: [ TOTAL:3 ]
remote: [ You got 100 (of 100) this time. Fri Jan  5 20:29:37 CST 2024 ]
remote:
remote:
remote: >>>>>> Collecting autotest results >>>>>>
```

三、实验总结

遇到的问题和解决办法：

1. 在做`Exercise2.1`的时候，发现一直报错，说我有语法错误，后面改了好久，发现是因为多了一些空行，而且需要使用`\`进行换行

收获与体会

对于虚拟地址和物理地址之间的转换更加熟悉，也更加理解了二级页表的机制

总时长：大概七八个小时吧