2.1

请参考代码注释，补全mips_detect_memory 函数。在实验中，从外设中获取了硬件可用内存大小memsize，请你用内存大小memsize 完成总物理页数npage 的初始化

```
void mips_detect_memory() {
    /* Step 1: Initialize memsize. */
    memsize = *(volatile u_int *)(KSEG1 | DEV_MP_ADDRESS | DEV_MP_MEMORY);

    /* Step 2: Calculate the corresponding 'npage' value. */
    /* Exercise 2.1: Your code here. */
    npage = memsize >> 12;                      // 一个页面大小是4KB
    /*            ↑                    */

    printk("Memory size: %lu KiB, number of pages: %lu\n", memsize / 1024,
npage);
}

```

2.2

完成include/queue.h 中空缺的函数LIST_INSERT_AFTER。 [将elm 插到已有元素listelm 之后。]
其功能是将一个元素插入到已有元素之后，可以仿照LIST_INSERT_BEFORE 函数来实现。

```
/*
 * Insert the element 'elm' *after* 'listelm' which is already in the list. The
'field'
 * name is the link element as above.
 *
 * Hint:
 * Step 1: assign 'elm.next' from 'listelm.next'.
 * Step 2: if 'listelm.next' is not NULL, then assign 'listelm.next.pre' from a
proper value.
 * Step 3: assign 'listelm.next' from a proper value.
 * Step 4: assign 'elm.pre' from a proper value.
 */
#define LIST_INSERT_AFTER(listelm, elm, field)
     \
    /* Exercise 2.2: Your code here. */
      \
    do {
     \
        LIST_NEXT((elm), field) = LIST_NEXT((listelm), field);
     \
        if(LIST_NEXT((listelm), field) != NULL) {
      \
            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)

// 宏中分行写要加上“\”换行。
```

2.3

完成page_init 函数。请按照函数中的注释提示，完成上述三个功能。此外，这里也给出一些提示：

> 1. 使用链表初始化宏LIST_INIT。
> 2. 将freemem 按照BY2PG 进行对齐（使用ROUND 宏为freemem 赋值）。
> 3. 将freemem 以下页面对应的页控制块中的pp_ref 标为1。
> 4. 将其它页面对应的页控制块中的pp_ref 标为0 并使用LIST_INSERT_HEAD 将其插入空闲链表。

```
void page_init(void) {
    /* Step 1: Initialize page_free_list. */
    /* Hint: Use macro `LIST_INIT` defined in include/queue.h. */
    /* Exercise 2.3: Your code here. (1/4) */
    LIST_INIT(&page_free_list);

    /* Step 2: Align `freemem` up to multiple of BY2PG. */
    /* Exercise 2.3: Your code here. (2/4) */
    freemem = ROUND(freemem, BY2PG);

    /* Step 3: Mark all memory below `freemem` as used (set `pp_ref` to 1) */
    /* Exercise 2.3: Your code here. (3/4) */
    int used_page = PADDR(freemem) / BY2PG;   //[PADDR]:physical address [BY2PG]:bytes to per page
    int i;
    for(i=0; i<used_page; i++) {
        pages[i].pp_ref = 1;
    }

    /* Step 4: Mark the other memory as free. */
    /* Exercise 2.3: Your code here. (4/4) */
    int j;
    for(j=used_page; j<npage; j++) {
        pages[j].pp_ref = 0;
        LIST_INSERT_HEAD(&page_free_list, &pages[j], pp_link);
    }

}
```

2.4

`page_alloc(struct Page **pp)`，它的作用是将page_free_list 空闲链表头部页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页控制块的地址。

完成page_alloc 函数。在page_init 函数运行完毕后，在MOS 中如果想申请存储空间，都是通过这个函数来申请分配。该函数的逻辑简单来说，可以表述为：

> 1. 如果空闲链表没有可用页了，返回异常返回值。
> 2. 如果空闲链表有可用的页，取出第一页；初始化后，将该页对应的页控制块的地址放到调用者指定的地方。填空时，你可能需要使用链表宏LIST_EMPTY 或函数page2kva

```
/* Overview:
 *   Allocate a physical page from free memory, and fill this page with zero.
 *
 * Post-Condition:
 *   If failed to allocate a new page (out of memory, there's no free page), return -E_NO_MEM.
 *   Otherwise, set the address of the allocated 'Page' to *pp, and return 0.
 *
 * Note:
 *   This does NOT increase the reference count 'pp_ref' of the page - the caller must do these if
 *   necessary (either explicitly or via page_insert).
 *
 * Hint: Use LIST_FIRST and LIST_REMOVE defined in include/queue.h.
 */
int page_alloc(struct Page **new) {
    /* Step 1: Get a page from free memory. If fails, return the error code.*/
    struct Page *pp;
    /* Exercise 2.4: Your code here. (1/2) */
    pp = LIST_FIRST(&page_free_list);
    if(pp == NULL) {
        return -E_NO_MEM;
    }

    LIST_REMOVE(pp, pp_link);

    /* Step 2: Initialize this page with zero.
     * Hint: use `memset`. */
    /* Exercise 2.4: Your code here. (2/2) */
    u_long temp = page2kva(pp);    //清空对应的物理页面。 [page2kva]:page to kernel virtual address
    memset((void*)temp, 0, BY2PG);   //都是使用虚拟地址

    *new = pp;
    return 0;
}
```

2.5

`page_free(struct Page *pp)`，它的作用是判断pp 指向页控制块对应的物理页面引用次数是否为0，若为0 则该物理页面为空闲页面，将其对应的页控制块重新插入到page_free_list。

完成page_free 函数。提示：使用链表宏LIST_INSERT_HEAD，将页结构体插入空闲页结构体链表。

```
/* Overview:
 *   Release a page 'pp', mark it as free.
 *
 * Pre-Condition:
 *   'pp->pp_ref' is '0'.
 */
void page_free(struct Page *pp) {
    assert(pp->pp_ref == 0);
    /* Just insert it into 'page_free_list'. */
    /* Exercise 2.5: Your code here. */
    LIST_INSERT_HEAD(&page_free_list, pp, pp_link);
    return;
}
```

2.6

`int pgdir_walk(Pde *pgdir, u_long va, int create, Pte **ppte)`，该函数将一级页表基地址pgdir 对应的两级页表结构中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。如果create 不为0 且对应的二级页表不存在，则会使用page_alloc 函数分配一页物理内存用于存放二级页表，如果分配失败则返回错误码。

完成pgdir_walk 函数。

该函数的作用是：给定一个虚拟地址，在给定的页目录中查找这个虚拟地址对应的物理地址，如果存在这一虚拟地址对应的页表项，则返回这一页表项的地址；如果不存在这一虚拟地址对应的页表项（不存在这一虚拟地址对应的二级页表、即这一虚拟地址对应的页目录项为空或无效），则根据传入的参数进行创建二级页表，或返回空指针。

注意，这里可能会在页目录表项无效且create 为真时，使用page_alloc 创建一个页表，此时应维护申请得到的物理页的pp_ref 字段。

```
/* Overview:
 *   Given 'pgdir', a pointer to a page directory, 'pgdir_walk' returns a
pointer to the page table
 *   entry (with permission PTE_D|PTE_V) for virtual address 'va'.
 *
 * Pre-Condition:
 *   'pgdir' is a two-level page table structure.
 *
 * Post-Condition:
 *   If we're out of memory, return -E_NO_MEM.
 *   Otherwise, we get the page table entry, store
 *   the value of page table entry to *ppte, and return 0, indicating success.
 *
 * Hint:
```

```
 *   We use a two-level pointer to store page table entry and return a state code to indicate
 *   whether this function succeeds or not.
 */
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 = (pgdir + PDX(va));    //PDX(va) 可以获取虚拟地址va 的31-22位

    /* 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 )) {     //不存在这一虚拟地址对应的二级页表
        if(create) {
            if(page_alloc(&pp) == -E_NO_MEM) {
                *ppte = NULL;
                return -E_NO_MEM;
            }
            pp->pp_ref++;
            *pgdir_entryp = page2pa(pp) | PTE_D | PTE_V;
        } else {
            *ppte = NULL;
            return 0;
        }
    }

    /* Step 3: Assign the kernel virtual address of the page table entry to '*ppte'. */
    /* Exercise 2.6: Your code here. (3/3) */
    Pte *pgtable;
    pgtable = KADDR(PTE_ADDR(*pgdir_entryp));   //[PTE_ADDR]:page table entry address
    *ppte = pgtable + PTX(va);

    return 0;
}
```

2.7

`int page_insert(Pde *pgdir, u_int asid, struct Page *pp, u_long va, u_int perm)`，作用是将一级页表基地址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，并将页表项权限为设置为perm。

完成page_insert 函数（补全TODO 部分）。

```c
int page_insert(Pde *pgdir, u_int asid, struct Page *pp, u_long va, u_int perm)
{
    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);    //清空对应在TLB中的缓存

    /* 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) */
    pgdir_walk(pgdir, va, 1, &pte);   //获取/创建
    if(pte == 0) {
        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 += 1;

    return 0;
}

```

2.8

完成kern/tlb_asm.S 中的tlb_out 函数。该函数根据传入的参数（TLB 的Key）找到对应的TLB 表项，并将其清空。

具体来说，需要在两个位置插入两条指令，其中一个位置为tlbp，另一个位置为tlbwi。因流水线设计架构原因，tlbp 指令的前后都应各插入一个nop 以解决数据冒险。

> 我们通过tlb_invalidate 实现删除特定虚拟地址在TLB 中的旧表项，函数的主要逻辑位于kern/tlb_asm.S 中的tlb_out 中。当tlb_out 被tlb_invalidate 调用时, a0 寄存器中存放着传入的参数，寄存器值为旧表项的Key(由虚拟页号和ASID 组成)。我们使用mtc0 将Key写入EntryHi，随后使用tlbp 指令，根据EntryHi 中的Key 查找对应的旧表项，将表项的索引存入Index。如果索引值大于等于0（即TLB 中存在Key 对应的表项），我们向EntryHi 和EntryLo 中写入0, 随后使用tlbwi 指令，将EntryHi 和EntryLo 中的值写入索引指定的表项。此时旧表项的Key 和Data 被清零，实现将其无效化。

```
#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      /*使用tlbp 指令，根据EntryHi 中的Key 查找对应的旧表项，将表项的索引存入Index

    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     /*使用tlbwi 指令，将EntryHi 和EntryLo 中的值写入索引指定的表项。此时旧表项的Key 和Data 被清零

.set reorder

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

2.9

完成kern/tlbex.c 中的_do_tlb_refill 函数。

> 提示: 尝试在循环中调用'page_lookup'以查找虚拟地址va。 在当前进程页表中对应的页表项'pte'如果'page_lookup'返回'NULL'，表明'pte'找不到，使用'passive_alloc'为va 所在的虚拟页面分配物理页面，直至'page_lookup'返回不为'NULL'则退出循环。你可以在调用函数时，使用全局变量cur_pgdir作为其中一个实参。

```
/* Overview:
```

```
 *  Refill TLB.
 */
Pte _do_tlb_refill(u_long va, u_int asid) {
    Pte *pte;
    /* Hints:
     *  Invoke 'page_lookup' repeatedly in a loop to find the page table entry 'pte' associated
     *  with the virtual address 'va' in the current address space 'cur_pgdir'.
     *
     *  **While** 'page_lookup' returns 'NULL', indicating that the 'pte' could not be found,
     *  allocate a new page using 'passive_alloc' until 'page_lookup' succeeds.
     */

    /* Exercise 2.9: Your code here. */
    while(page_lookup(cur_pgdir, va, &pte) == NULL) {
        passive_alloc(va, cur_pgdir, asid);
        //cur_pgdir是一个在kern/pmap.c 定义的全局变量，其中存储了当前进程一级页表基地址位于kseg0 的虚拟地址。

    }

    return *pte;
}

```

2.10

完成kern/tlb_asm.S 中的do_tlb_refill 函数。

```
.data
tlb_refill_ra:
.word 0
.text
NESTED(do_tlb_refill, 0, zero)
    mfc0    a0, CP0_BADVADDR
    mfc0    a1, CP0_ENTRYHI
    srl     a1, a1, 6
    andi    a1, a1, 0b111111
    sw      ra, tlb_refill_ra
    jal     _do_tlb_refill
    lw      ra, tlb_refill_ra
    mtc0    v0, CP0_ENTRYLO0
    // See <IDT R30xx Family Software Reference Manual> Chapter 6-8
    nop
    /* Hint: use 'tlbwr' to write CP0.EntryHi/Lo into a random tlb entry. */
    /* Exercise 2.10: Your code here. */
    tlbwr

    jr      ra
END(do_tlb_refill)
```