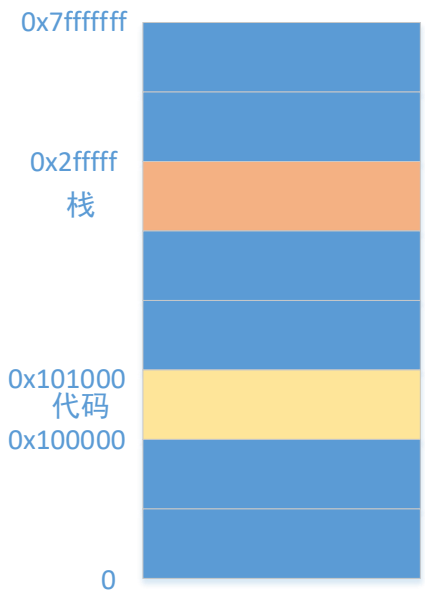
Project 5 Virtual Memory 设计文档

中国科学院大学
[姓名] 袁峥
[日期] 2017.12.20

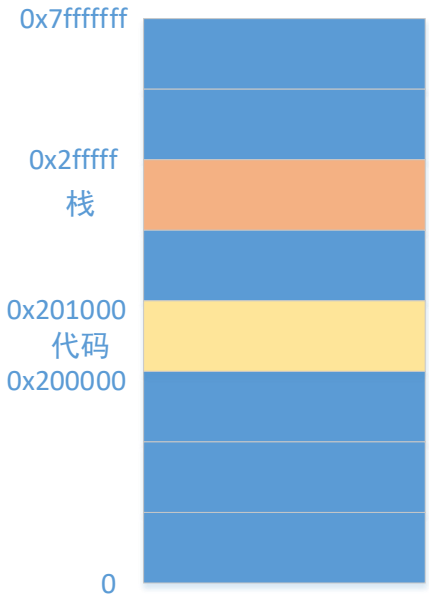
1. 用户态进程内存管理设计

（1）测试的用户态进程虚存布局是怎样的？请说明

进程 1：



进程 2：



（2）你设计的页表项结构是怎样的，包含哪些标记位？

其中 G 为全局比较位，对应与 TLB 中的 G 位，在此次实验中全部置为 0，D 和 V 位也和 TLB 中的对应标志位一致，分别表示可写和有效。

（3）任务 1 中用户态进程页表初始化做了哪些操作？使用了多少个页表项（PTE），以及使用了多少个物理页保存页表？

首先分配一个物理页用来放页表，并将其 pin 住，不允许进行页替换，同时为进程的程序段分配物理页，将其物理页的地址写入页表中对应虚地址的表项，由于是在初始化时就分配了物理页，因此其 D 和 V 标志位全部置 1。由于任务 1 中进程的用户态栈使用的仍然是内核栈，因此不需要为用户态栈分配物理页。

每个进程需要一个页表项，对应进程的程序段，因为编译后发现进程代码段大小小于一个物理页大小。页表只需要 1 个物理页保存。每个进程使用了两个物理页，一个为页表，另一个为代码段。

（4）任务 2 中用户态进程页表初始化做了哪些操作？使用了多少个页表项（PTE），以及使用了多少个物理页保存页表？

首先将页表全部清空，由于任务 2 中的物理页是按需分配，因此初始化时不需要分配物理页。任务 2 中的用户栈也放在了虚拟空间，在我的设计中将用户栈的最高地址设为 0x2fffff。每个进程使用了 2 个页表项，一个为代码段，一个为栈段，但两个表项的 V 标志位全部为 0，还没有分配物理页。一张页表可以管理 4K*1024 的虚拟空间，而 0x2fffff 在这段虚拟空间内，因此每个进程只需要一张物理页用来保存页表。

（5）物理内存使用什么数据结构进行管理，描述物理内存的元数据信息有哪些，各有什么用途？此处的物理页分配策略是什么？

物理内存用一个 page_map 数组来管理。数组中每个表项的结构如下：

```
/* TODO: Structure of an entry in the page map */
typedef struct {
    // design here
    node_t node;
    uint32_t vaddr;
    uint32_t index;
    bool_t used;
    bool_t pinned;
    bool_t readed;
    int pid;
} page_map_entry_t;
```

node 是用队列管理物理页时使用的节点。

pid 是该物理页保存的数据为第几个进程的内容。

vaddr 是该物理页所保存的数据在对应进程中的虚地址，用于在页替换时计算替换内容的地址，即内存拷贝时的目标地址。

index 是该物理页的序号，在页替换时使用，用于计算内存拷贝时的源地址。

used 是记录该物理页是否被使用。

pinned 是记录该物理页是否可以被替换。

readed 为 bonus1 中采用 FIFO 二次机会算法时，记录该物理页近期是否被访问过。

物理页的分配策略，在任务 1 中为进行页表初始化时就同时将进程所需物理页分配好，并填入页表中，在任务 2 中为按需分配，进程访问时首先触发 TLB miss，然后再触发 page fault，此时再进行物理页分配。

（6）TLB miss 何时发生？你处理 TLB miss 的流程是怎样的？

TLB miss 在程序访问的地址需要经过 MMU 转换，且在 TLB 中没有找到时触发。

在 handle_tlb 中，首先通过 tlbp 查找 TLB，如果 CP0_INDEX 返回的最高位为 1 则说明 TLB miss。

在 TLB miss 中首先通过 CP0_BADVADDR 的地址，到页表中找到对应虚地址的表项，并将其附近奇偶两项分别填入 CP0_ENTRYLO0 和 CP0_ENTRYLO1，在通过 tlbwr 将其填入 TLB 中，完成 TLB miss 的处理。

（7）设计或实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

在实验中，首先根据计算机体系结构课程中对 MIPS 的了解，认为 TLB refill 异常的处理程序入口地址应该为 0xbfc00200，因此一开始将 TLB miss 的处理程序放在了该地址。后

来在执行时发现触发 TLB refill 时会跳转到该地址执行，但是执行的结果很奇怪，反复折腾后决定放弃了，将 TLB refill 和 TLB invalid 的处理放在 handle_tlb 中，通过一个 tlbp 指令来区分。

另外，在任务 2 中用户栈使用虚地址空间后，引发了许多问题。首先是在进程切换的上下文保存中。如果是一个进程进行上下文保存，在 SAVE_CONTEXT(USER)后，此时的 sp 寄存器中存放的仍然是进程的用户栈，这样导致如果是进行的时钟中断，再进行 SAVE_CONTEXT(KERNEL)时，进程的内核栈会被用户栈覆盖。同时，还会引发的另一个问题是，如果 SAVE_CONTEXT(USER)后，在内核函数的执行过程中仍然使用用户栈的话，在进程切换时调用 set_pt 函数后 CP0_ENTRYHI 寄存器中的 ASID 会改变，此时进程栈的虚地址和 ASID 号不匹配，会触发新的缺页中断，从而发生错误。

因此在 SAVE_CONTEXT(USER)结束后，需要将此时的 sp 置成内核栈。

此外，在 handle_tlb 中的 SAVE_CONTEXT 也应该区分进线程，不然也会发生线程的内核栈发生异常的情况。

2. 缺页中断与 swap 处理设计

（1）任务 1 和任务 2 中是否有缺页中断？若有，何时发生缺页中断？你设计的缺页中断处理流程是怎样的？

任务 1 由于物理页在建立页表时就分配了，因此不会发生缺页中断。任务 2 中的物理页是按需分配，因此会发生缺页中断。

当访问的虚地址在 TLB 中无法查找到对应项时，首先会触发 TLB miss，将页表中的对应项放入 TLB 中，再次访问该虚地址时，在 TLB 中会找到对应项，但是如果事先没有分配物理页，则 V 位为 0，此时会触发缺页中断。

缺页中断的处理时，首先判断触发缺页中断的虚地址的范围，如果在栈段的范围，则直接为其分配一个空的物理页，如果在代码段的范围，则为其分配一个物理页后，还需要根据虚地址将其程序内容拷贝到物理页中。同时需要更新该 page_map 中该物理页的信息。

如果没有空的物理页可以分配，则进行页替换处理。

（2）你设计中哪些页属于 pinning pages？你实现的页替换策略是怎样的？

我的设计中，页表页和栈页属于 pinning pages。

页替换使用 FIFO 策略，在物理页分配时，如果该页不需要被 pin 住，那么将其放入物理页队列。当没有空的物理页用来分配时，将 FIFO 队列中的第一项取出，同时将其内容存入对应 swap 空间，并将物理页清空，然后将该页分配给新的物理页请求。

（3）设计或实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

在这部分实验中碰到的问题主要在页替换时由于需要内存块的拷贝，因此可能比较费时，同时该部分内容是不能被打断的，因此在处理前关了时钟中断。这样可能会导致如果访问一个没有分配物理页的地址时，会先后触发 TLB miss 和 PAGE fault，这期间如果要进行页替换还需要先后将物理页中原来的内容换出，同时再换入新的内容。这一系列内容执行结束后，所花费的时间可能已经超过了原先一个时间片的时间，所以打开中断后立刻响应时钟中断，并进行进程切换，换到新的进程时，如果此时访问的虚地址也没有分配物理页，则也需要重复以上操作，同时分配结束后可能也会立刻进行中断响应而无法真正执行程序内容。如果两个进程的代码段只有一共只有一个物理页可以使用时，两个进程会来回切换，且相互都不能真正执行程序内容，从而导致看上去进程都没有执行，而是一直在进行页替换。

为了解决这个问题，只需要将时钟中断的间隔稍微变大，这样便可以使得在缺页替换处理完成后还有时间可以用来真正执行程序。

3. Bonus 设计

（1）Bonus 中你限制物理内存到多大？Bonus 任务 1 中的页替换策略是怎样的？和常规任务中的页替换策略比，Bonus 中的策略能减少页替换数量么？

在 Bonus 任务 1 中，为了显示出新的页替换策略的优越性，因此修改了 process2，在其中增加了一个 4096 项的数组，并每次循环时都遍历一遍，同时再另外访问一次下标较低的项。代码如下：

```
int array[4096];

void __attribute__((section(".entry_function"))) _start(void)
{
    int i,j, res;

    for (i = 0; i <= 100; i++) {
        res = rec(i);

        array[i] = i;

        for (j=0; j<4096; j++)
            array[j] = 1;

        printf(LINE, 0, "PID %d : 1 + ... + %d = %d", getpid(), i, res);
        sleep(300);
        yield();
    }
    exit();
}
```

从代码中可以看出进程访问的地址同时有局部性和全局性，因此可以用来测试新的替换算法。在 process2 编译后，发现其代码段大小占用 5 个物理页框，因此由 2 个页表页、2 个栈页和进程 1 的 1 个代码页以及进程 2 的 5 个代码页，一共需要 10 个页面，测试时限制物理内存为 7 个物理页框，用来测试替换效果。

此处所采用的新的替换算法是 FIFO 第二次机会算法，具体实现如下。由于在实验环境中比较难准确地记录访问了哪些页面，因此在此采用略为简略的办法。在每次时钟中断时，检查 CP0_EPC 的地址，这个地址可以认为是访问的地址，同时在 page_map 中查找相应页框，将其标记成访问过的（readed=TRUE）。

```
void timer_irq(uint32_t epc) {
  enter_critical();
  time_elapsed++;
  int i;
  if (current_running->nested_count)
    return;

  for (i=0; i<PAGEABLE_PAGES; i++)
    if (((epc&0xfffff000) == (page_map[i].vaddr&0xfffff000)) && current_running->pid == page_map[i].pid)
      page_map[i].readed = TRUE;

  printf(37,1,"time int epc is 0x%x   pid is %d",epc,current_running->pid);
  print_page_map();
  reset_timer(TIMER_INTERVAL);
  put_current_running();
  scheduler_entry();
  enter_critical();
}
```

在分配页框时，如果没有现成的空页框，那么采用 FIFO 第二次机会算法来进行替换，首先从队列头中取下一个页框，如果其 readed=TRUE，表示该页近期被访问过，则将其 readed

改为 FALSE，再将该页重新放入队尾，直到从队头找到一个 readed 为 FALSE 的页框，并将其替换。

```
page_map_entry_t *FIFO_second_replace(){
    page_map_entry_t * exchange_node;
    while (1){
        exchange_node = (page_map_entry_t *)dequeue(&phy_page_queue);
        printf(38,1,"fifo pid %d  readed %d",exchange_node->pid,exchange_node->readed);

        if (exchange_node->readed == TRUE){
            exchange_node->readed = FALSE;
            enqueue(&phy_page_queue,(node_t*)exchange_node);
        }
        else break;
    }
    ASSERT(exchange_node != NULL);
    return exchange_node;
}
```

```
IN SWAP_OUT phy_entry vaddr is 0x00201000, to 0xa0d01000
happened have no phy_pages,exchange 6
PAGE 0 pinned 1         pid 1   vaddr 0xa0908000         readed 0
PAGE 1 pinned 1         pid 2   vaddr 0xa0909000         readed 0
PAGE 2 pinned 0         pid 2   vaddr 0x00200054         readed 1
PAGE 3 pinned 0         pid 1   vaddr 0x00100230         readed 0
PAGE 4 pinned 1         pid 2   vaddr 0x002fffec         readed 0
PAGE 5 pinned 1         pid 1   vaddr 0x002ffff4         readed 0
PAGE 6 pinned 0         pid 2   vaddr 0x00202000         readed 0
```

在程序运行时，从 process2 中可以看出虚地址 0x00200000 所在页的访问频率比其他页要高，从上图中可以看到其被分配在第 2 个物理页框，同时 readed 标记被标记为 TRUE。在发生页替换时，优先选择了 readed 没有被标记为 TRUE 的第 6 个物理页。

采用这种设计，从测试情况中可以看出，比原来的 FIFO 算法有所提高。下图分别为采用 FIFO 和 FIFO 第二次机会算法在进行了 10 次循环后的结果。

FIFO 算法：

```
Pid     Type    Prio    Status  Entries TLB miss        Page fault

0       Thread  1       Ready   10      0       0
1       Process 1       Ready   9       10      5
2       Process 1       Ready   9       26      16
```

FIFO 第二次机会算法：

```
Pid     Type    Prio    Status  Entries TLB miss        Page fault

0       Thread  1       Ready   10      0       0
1       Process 1       Ready   9       6       3
2       Process 1       Ready   9       19      10
```

（2）Bonus 任务 2 的二级页表中，你设计了多少个页表项（PTE），使用了多少物理页保存二级页表，和常规任务中的一级线性页表相比，使用物理页数有减少么？二级页表的页目录项结构是怎样的，包含哪些信息？

Bonus 任务 2 完成了设计，但是代码还没有来得及最后完成，在此描述一下实验设计。

每个进程需要一个二级页表的页目录，每一个表项对应一张一级页表，对应 1024*4KB 的虚地址空间。每个页目录项包含对应一级页表的地址，和一个 V 位，表示该项的一级页表是否已经分配。由于虚地址范围小于 32 位，因此加上 V 位正好可以一起放在一个 32 位数据结构中。每个一级页表与之前的设计相同。

对于之前的进程，由于每个进程使用的代码段空间和栈空间较少，因此使用二级页表效

果不明显。使用进程 1 进行分析，需要一个物理页来保存二级页目录，和一个物理页来保存一张一级页表。但是常规任务中一共只需要一张一级线性页表，因此在此例中使用二级页表效果反而不好。

因此额外设计了一个进程来体现二级页表的效果。新的进程中将栈的地址分配到 0xF00000，此地址如果使用一级页表来存储，需要使用第 4 张一级页表。因此一共需要分配 4 个物理页来保存一级页表。但是如果使用二级页表，首先二级页目录需要分配 1 个物理页，然后只需再分配两张一级页表，分别对应二级页目录中的第一个和第四个表项，里面分别放代码段和栈段，这样使用二级页表一共只需要使用 3 张物理页，相比及一级页表，体现了其优越性。

（3）设计或实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

在 Bonus1 中，由于目前的代码框架中很难精确的判断是否有访问某一个虚地址，因此很难真正实现 R 位的效果，最后采用了利用时钟中断时的 CP0_EPC 寄存器的值来替代，也在一定程度上达到了效果。

在 Bonus2 中，由于和常规实验相比，几乎要将整体架构全部重写，因此任务量比较大，没能在最后的期限内调试完成。此外，本来的两个进程使用的代码段和栈段的地址空间都很小，反而使用一级页表的效果更好。在设计一个有效的进程来体现二级页表的效果上，也花费了一些时间。

4. 关键函数功能

请列出上述各项功能设计里，你觉得关键的函数或代码块，及其作用

1、 handle_tlb(entry.S)

```
NESTED(handle_tlb,0,sp)
    SAVE_CONTEXT(USER)
    tlbp
    mfc0 k0, CP0_INDEX
    bltz k0, tlb_miss
    nop

tlb_invalid:
    mfc0 a0, CP0_BADVADDR
    jal  do_page_fault
    nop
    j    handle_tlb_finish
    nop
tlb_miss:

    li   t0, 0xffffe000
    li   t1, 0x00000fff
    li   t2, 0xfffff000
    mfc0 k0, CP0_BADVADDR

    and  t3, k0, t0
    mfc0 k1, CP0_ENTRYHI
    and  t4, k1, t1
    or   k1, t3, t4
    mtc0 k1, CP0_ENTRYHI

    lw   t4, current_page_table
    srl  t3, t3, 13
    sll  t3, t3, 3
    add  t4, t4, t3
    lw   t5, 0(t4)
    andi t6, t5, 0xf
    and  t7, t5, t2
    srl  t7, t7, 6
    or   t7, t7, t6
    mtc0 t7, CP0_ENTRYLO0

    lw   t5, 4(t4)
    andi t6, t5, 0xf
    and  t7, t5, t2
    srl  t7, t7, 6
    or   t7, t7, t6
    mtc0 t7, CP0_ENTRYLO1
    mtc0    zero,CP0_PAGEMASK
    tlbwr

    jal do_tlb_miss
    nop
handle_tlb_finish:
    RESTORE_CONTEXT(USER)
    j return_from_exception
    nop
END(handle_tlb)
```

首先先保存用户态上下文，然后通过一个 tlbp 指令来判断是 TLB miss 还是 page fault，然后进入对应的处理程序，再处理结束前再恢复用户态上下文。

2、 set_pt(entry.S)

```
LEAF(set_pt)
    lw   k0, current_running
    lw   k1, PID(k0)

    mtc0 k1, CP0_ENTRYHI

    lw   k1, PAGETABLE(k0)
    sw   k1, current_page_table
    jr   ra
    nop
END(set_pt)
```

在进程切换时调用该函数，将新进程的 PID 填入 CP0_ENTRYHI 的 ASID 域，并将当前页表切换成新进程的页表，用于在 TLB miss 和 page fault 时使用。

3、 init_memory(memory.c)

```
uint32_t init_memory( void ) {

    // initialize all pageable pages to a default state
    int i;
    lock_init(&page_fault_lock);
    queue_init( &phy_page_queue );

    for (i = 0; i < PAGEABLE_PAGES; i++){
        page_map[i].reaped = FALSE;
        page_map[i].index = i;
        page_map[i].pid = -1;
        page_map[i].pinned = FALSE;
        page_map[i].used = FALSE;
        page_map[i].vaddr = 0;
    }
    return 0;
}
```

初始化物理页队列，同时将所有物理页的管理信息进行初始化。

4、 setup_page_table(memory.c)

```
uint32_t *setup_page_table( int pid ) {
    uint32_t *page_table;

    // alloc page for page table
    int page_index = page_alloc(TRUE);
    page_table = (uint32_t *)page_vaddr(page_index);
    page_map[page_index].pid = pid;
    page_map[page_index].vaddr = page_table;

    // initialize PTE and insert several entries into page tables using insert_page_table_entry

    int proc_pages = (pcb[pid].size - 1) / PAGE_SIZE + 1;
    int i;

    if (on_demand == TRUE){
        for (i = 0; i < proc_pages; i++){
            insert_page_table_entry(page_table, pcb[pid].entry_point + i*PAGE_SIZE ,0,0,pid);
        }
        insert_page_table_entry(page_table, pcb[pid].user_tf.regs[29],0,0,pid);
    }
    else{
        for (i = 0; i < proc_pages; i++){
            page_index = page_alloc(FALSE);
            page_map[page_index].vaddr = pcb[pid].entry_point + i*PAGE_SIZE;
            bcopy((char *)(pcb[pid].disk_place+i*PAGE_SIZE),(char *) page_vaddr(page_index),PAGE_SIZE);
            insert_page_table_entry(page_table, pcb[pid].entry_point + i*PAGE_SIZE,page_paddr(page_index),PE_V|PE_D,pid);
        }
    }
    print_page_map();
    return page_table;
} « end setup_page_table »
```

该函数进行页表的初始化，首先分配一个物理页用来存放页表，如果是任务 1，则将进程的代码段也分配物理页，并将对应代码拷贝至物理页中，然后将对应页表项插入页表，此时该页有效。如果是任务 2，则将对应代码段的虚地址所在表项的有效位置为 0，同时将栈段所在表项的有效位也置为 0，不为其事先分配物理页，而是等到访问时通过 page fault 处理分配。

5、page_alloc(memory.c)

```
int page_alloc( bool_t pinned ) {

    // code here
    int free_index;
    page_map_entry_t * exchange_node;
    for (free_index = 0; free_index < PAGEABLE_PAGES; free_index++){
        if (page_map[free_index].used == FALSE)
            break;
    }
    if (free_index >= PAGEABLE_PAGES){
        exchange_node = FIFO_second_replace();
        //exchange_node = (page_map_entry_t *)dequeue(&phy_page_queue);
        swap_out(exchange_node);
        free_index = exchange_node->index;
        page_map[free_index].readed = FALSE;
        page_map[free_index].pid = -1;
        page_map[free_index].pinned = FALSE;
        page_map[free_index].used = FALSE;
        page_map[free_index].vaddr = 0;
        printf(30,1,"happened have no phy_pages,exchange %d",free_index);
    }
    if (pinned == FALSE) enqueue(&phy_page_queue,(node_t *)&page_map[free_index]);
    page_map[free_index].readed = FALSE;
    page_map[free_index].pid = current_running->pid;
    page_map[free_index].pinned = pinned;
    page_map[free_index].used = TRUE;

    ASSERT( free_index < PAGEABLE_PAGES );
    bzero((char*)page_vaddr(free_index), PAGE_SIZE);

    return free_index;
} « end page_alloc »
```

首先遍历物理页表管理的数组，如果有未使用的页则直接取出，否则通过替换算法取出一页，同时将该页中原来的内容换出。如果新的页请求不需要 pin 住，那么将该页加入物理页队列，用于页替换时使用。同时更新该物理页的管理信息。

6、do_page_fault(memory.c)

```
void do_page_fault(uint32_t vaddr){

    current_running->nested_count++;
    //lock_acquire(&page_fault_lock);
    current_running->page_fault++;

    int i;
    if (vaddr > 0x20ffff)
    {
        i = page_alloc(TRUE);
        page_map[i].vaddr = vaddr;
        printf(16,1,"alloc phy page %d",i);
        insert_page_table_entry(current_page_table,vaddr,page_paddr(i),PE_V|PE_D,current_running->pid);
    }
    else
    {
        i = page_alloc(FALSE);
        page_map[i].vaddr = vaddr;
        printf(16,1,"alloc phy page %d",i);
        bcopy((char *)(current_running->disk_place+(get_page_idx(vaddr)<<12)-current_running->entry_point),(char *) page_vaddr(i),PAGE_SIZE);
        insert_page_table_entry(current_page_table,vaddr,page_paddr(i),PE_V|PE_D,current_running->pid);
    }
    //lock_release(&page_fault_lock);
    current_running->nested_count--;
    print_page_map();

} « end do_page_fault »
```

该函数处理缺页中断，首先根据触发异常的虚地址判断是栈缺页还是代码段缺页，如果是栈缺页，那么为其分配一个 pin 住的页，同时更新对应页表项。如果是代码段缺页，则为其分配一个不需要 pin 住的页，同时将代码段内容从 swap 空间换入物理页，并更新页表项。