

## 1 思考题

### 1.1 多级页表的优劣势

多级页表的优势在于：允许页表内存在“空洞”，操作系统可以在虚拟地址被应用进程使用之后再分配和填写相应的页表页，这样当应用进程的虚拟地址空间范围非常大，但是真实使用的虚拟页又非常少的时候，多级页表可以仅仅映射比较少的页数，从而能够大大减少页表本身所占用的物理内存。

多级页表能发挥这样一种优势的前提条件，是应用进程使用的虚拟地址远小于总的虚拟地址空间。如果虚拟地址空间用满，多级页表最末级的页表和单级页表使用的内存大小是一样的，这样多级页表的其他级页表就是多出来的，相比于单级页表占用了更多的内存。

### 1.2 以 4KB 粒度映射时

总地址范围 4GB，故需要  $\frac{4GB}{4KB} = 1M$  个物理页。由于单个页表页中每项占据 8B，一个页表页本身的项数为  $\frac{4KB}{8B} = 2^9$ ，即一个最末级页表页可以指向  $2^9$  个物理页。这样就需要  $\frac{1M}{2^9} = 2^{11}$  个 L3 页表页。同理就需要  $\frac{2^{11}}{2^9}$  个，即 4 个 L2 页表页。这时候只需要 1 个 L0、1 个 L1 页表页。所以一共需要  $2048+4+1+1=2054$  个页表页。

### 1.3 以 2MB 粒度映射时

此时需要  $\frac{4GB}{2MB} = 2K = 2^{11}$  个物理页。然而，由于物理内存的映射粒度发生了变化，我们需要重新计算每个页表页中能够容纳的项数。由于单个页表页本身此时也占据 2MB 的空间，一个页表页中含有  $\frac{2MB}{8B} = 2^{18}$  项。所以只需要一张最末级页表就足够包含这  $2^{11}$  个物理页。

在此基础上我们需要考虑，此时我们能够容纳多少级页表。由于粒度为 2MB，物理页内偏移为  $2^{21}$ ，由刚才的计算知，单个页表页含有  $2^{18}$  项，故单个页表页索引长度为 18。由于我们虚拟地址只有 64 位，最多也只能容纳两级页表（即占用  $18+18+21=57$  位）。这样我们需要 1 个 L0 页表页，1 个 L1 页表页，所以一共是 2 个页表页。

（当然此时我们也可以只用单级页表来进行索引，毕竟一张页表的  $2^{18}$  项就已经足够覆盖我们需要的  $2^{11}$  个不同的物理页，这时候我们实际上只需要 1 个页表页。）

## 2 练习题

代码见附录 1。我们在配置 `ttbr1` 的时候，填入的物理地址是和 `ttbr0` 一样的，不同之处仅仅是虚拟地址的差别。所以我们只需要在第 78 行设定配置 `ttbr1` 虚拟地址的开始，即

```
vaddr = KERNEL_VADDR(0xfffff00000000000)+PHYSMEM_START
```

在本代码定义的每个页表的 `PTP_ENTRIES` 为 512 的情况下，此处我们以 2MB 为粒度映射 1GB 内存恰好需要 512 个页表项，也就是说，一个 L2 页表页就足够了。从而对应的 L0 和 L1 页表页里也暂时仅仅需要填写一项，即下一级页表的首地址。这一部分我们在 `step1` 中填写。

之后，我们还需要在 87 行，填写物理地址前，设定

```
vaddr = PHYSMEM_START
```

然后只需要用循环体将对应的 L2 页表页每项填写上对应的物理地址即可。

### 3 思考题

相关图片见附录1(a)1(b)。从 ChCore 启动到 `el1_mmu_activate` 函数启用之前，ChCore 使用物理地址，MMU 启动之后，CPU 会对 PC 中的地址进行翻译，从这时候开始 PC 中的地址将被视为虚拟地址。为了保证指令的正常运行，必须为低虚拟地址段配置相应的页表，并将低地址区域的虚拟地址映射为完全相同的物理地址。因为在调用 `start_kernel` 函数之后，内核会跳转到高地址区域运行，但是在启动过程中调用过的函数仍然可能用到低虚拟地址，如果不提前将这部分低虚拟地址的页表配置好，MMU 将无法翻译这部分虚拟地址，导致使用低虚拟地址空间的函数访问出现异常。

我猜测如果不设置低虚拟地址的页表，在 `el1_mmu_activate` 执行过程中，访问低虚拟地址时会出现异常。为了验证这一猜想，我将 `mmu.c` 中有关 `ttbr0` 配置已经给出的代码全部注释掉，然后使用 gdb 调试，在 `el1_mmu_activate` 函数处设置断点，之后 `continue` 到此函数处1(a)，并进行单步调试，发现确实出现了地址访问异常1(b)。

### 4 练习题

代码见附录2。下面按照函数调用的逻辑，简单介绍函数实现的过程。

由于在 `init_buddy` 函数的最后调用了 `buddy_free_pages`，我首先实现了此函数。`buddy_free_pages` 只需要将对应 `page` 标记为未分配，然后将其插入到对应 `order` 的空闲链表中，之后，调用 `merge_page` 函数进行块合并即可。在 `buddy_free_pages` 函数中并不需要关心对应块是否能被合并，这部分逻辑可以在 `merge_page` 中处理。

`merge_page` 首先判断能否合并。之后对于能够合并的情况，永远让指针指向较小的地址，然后将被合并的伙伴块标记为已分配，之后将被合并的两块从空闲链表中删除，再将合并后的块插入升阶之后的空闲链表中。由于合并之后的块可能仍然可以继续被合并，在末尾需要递归调用函数自身。

从逻辑上来讲，需要先实现 `split_page`，以便供 `buddy_get_pages` 调用。在实现 `split_page` 函数时我并没有采用递归的方法。参数 `order` 规定了欲分裂得到的块大小，在当前块阶数高于 `order` 时，不断对当前块降阶，取其伙伴块插入对应阶的空闲链表，直到待分裂块满足阶数需求即可。由于我们返回的是一个供 `buddy_get_pages` 使用的块，所以在函数返回前要将返回的块标记为已分配。

对于我们想要获取的某个特定 `order` 的块，`buddy_get_pages` 函数只需要从空闲链表里找到满足阶数需求的最小块，然后调用 `list_entry` 宏来得到一个对应阶数的块，并将块分裂请求发给 `split_page` 即可。在当前阶数的空闲链表里已经有满足需求的块存在的情况下，`split_page` 函数并不会真的进行分裂，而是直接返回对应的块。在这种情况下，`split_page` 和 `buddy_get_pages` 的逻辑是自洽的。

### 5 练习题

代码见附录3。`query_in_pgtbl` 只需要逐级调用 `get_next_ptp` 函数，直到找到对应的页表块或者项，然后根据虚拟地址设定相应的物理地址即可。对于 `map` 和 `unmap` 这一对操作，只需要针对在地址区间内的每一个地址，还是逐级调用 `get_next_ptp` 函数，然后将所需的标志位设置或者重置。

## 6 练习题

代码见附录4。基本思路是和练习题 5 一致的，区别在于这里需要首先处理大小为 1GB 和 2MB 的页。对于不同大小的页面，调用 `get_next_ptp` 的级数不同，设定的页表项（块）参数也略有差别。在 `huge` 函数的末尾再调用练习题 5 中的函数将大小为 4KB 的页实现 `map` 和 `unmap` 即可。

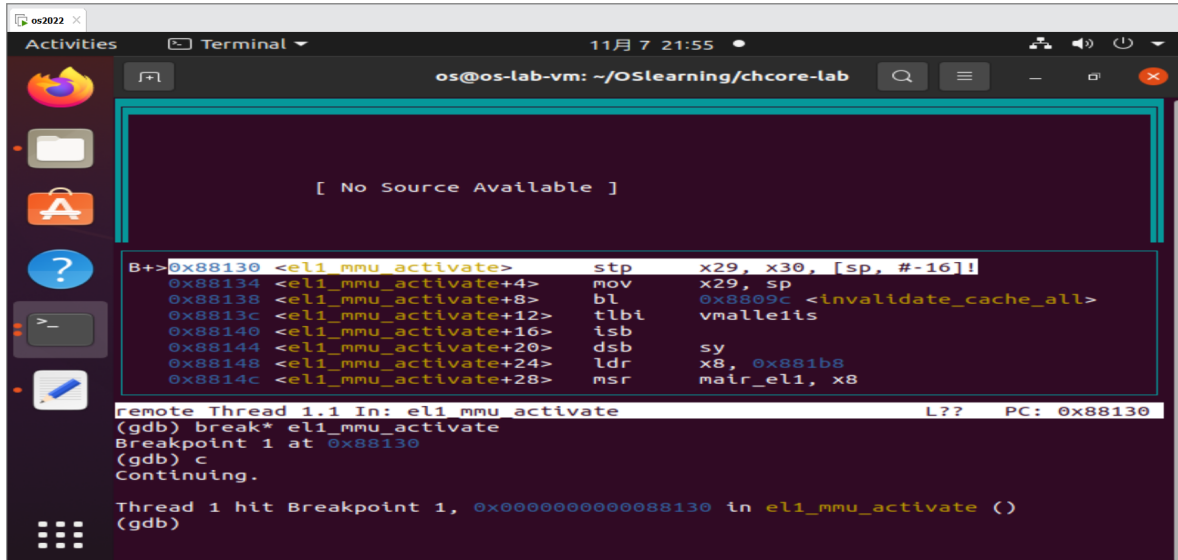
## 7 思考题

需要配置页表是否可写字段。发生缺页异常时，如果操作系统发现当前的缺页异常并不是物理页没有映射，而是由于应用程序尝试写入只读内存且相应的内存区域被标记成写时拷贝，这时操作系统会将物理页重新拷贝一份，并且将新拷贝的物理页以可读写的方式重新映射给触发异常的进程，再将控制流返回对应的进程。

## 8 思考题

内核实际运行时有时可能仅仅需要比较小的物理内存段。如果仅仅实现 2MB 粒度的页表，在没有实现 slab 机制的情况下，有可能会造成比较严重的内部碎片问题。

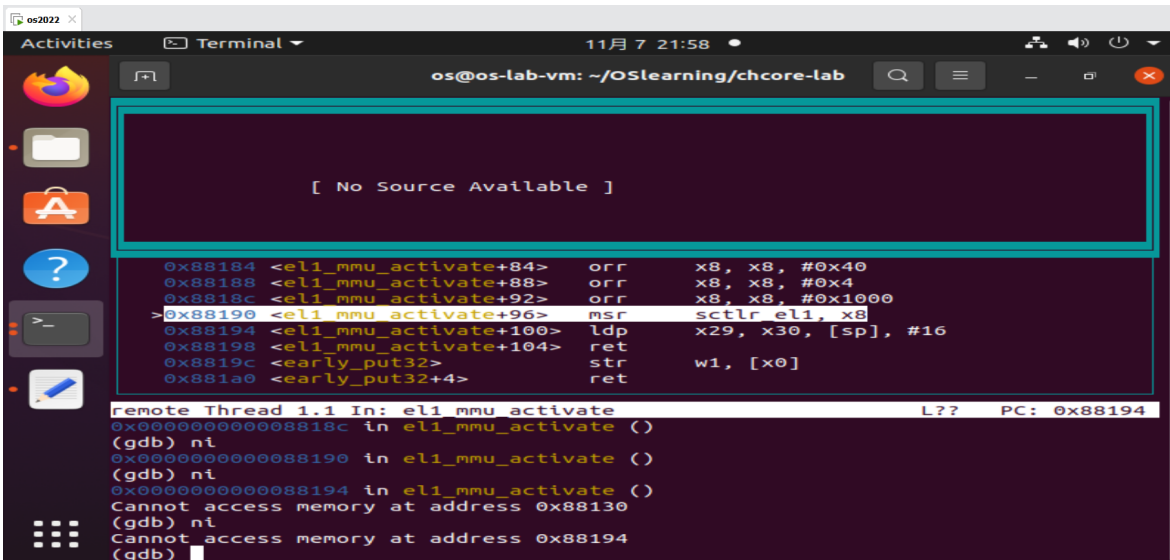
## 附录



The screenshot shows a terminal window with a GDB session. The user has set a breakpoint at address 0x88130 in the `el1_mmu_activate` function. The GDB output shows the program has hit the breakpoint. The assembly code for the function is displayed, starting with `stp x29, x30, [sp, #-16]!` and ending with `mair_el1, x8`. The GDB prompt is at the bottom, showing the current thread and PC.

```
B+>0x88130 <el1_mmu_activate> stp x29, x30, [sp, #-16]!  
0x88134 <el1_mmu_activate+4> mov x29, sp  
0x88138 <el1_mmu_activate+8> bl 0x8809c <invalidate_cache_all>  
0x8813c <el1_mmu_activate+12> tlbi vmalle1is  
0x88140 <el1_mmu_activate+16> isb  
0x88144 <el1_mmu_activate+20> dsb sy  
0x88148 <el1_mmu_activate+24> ldr x8, 0x881b8  
0x8814c <el1_mmu_activate+28> mair_el1, x8  
  
remote Thread 1.1 In: el1_mmu_activate L?? PC: 0x88130  
(gdb) break* el1_mmu_activate  
Breakpoint 1 at 0x88130  
(gdb) c  
Continuing.  
  
Thread 1 hit Breakpoint 1, 0x000000000000088130 in el1_mmu_activate ()  
(gdb)
```

(a) 设置断点并执行到 `el1_mmu_activate` 处



The screenshot shows a terminal window with a GDB session. The user has stepped through the `el1_mmu_activate` function. The GDB output shows the program has hit the breakpoint. The assembly code for the function is displayed, starting with `orr x8, x8, #0x40` and ending with `ret`. The GDB prompt is at the bottom, showing the current thread and PC. The output also shows memory access errors at addresses 0x88130 and 0x88194.

```
0x88184 <el1_mmu_activate+84> orr x8, x8, #0x40  
0x88188 <el1_mmu_activate+88> orr x8, x8, #0x4  
0x8818c <el1_mmu_activate+92> orr x8, x8, #0x1000  
>0x88190 <el1_mmu_activate+96> msr sctlr_el1, x8  
0x88194 <el1_mmu_activate+100> ldp x29, x30, [sp], #16  
0x88198 <el1_mmu_activate+104> ret  
0x8819c <early_put32> str w1, [x0]  
0x881a0 <early_put32+4> ret  
  
remote Thread 1.1 In: el1_mmu_activate L?? PC: 0x88194  
0x00000000000008818c in el1_mmu_activate ()  
(gdb) nl  
0x000000000000088190 in el1_mmu_activate ()  
(gdb) nl  
0x000000000000088194 in el1_mmu_activate ()  
Cannot access memory at address 0x88130  
(gdb) nl  
Cannot access memory at address 0x88194  
(gdb)
```

(b) 单步执行直到发现地址访问异常

图 1: 地址访问异常测试

```
77  /* first set the virtual address for kernel mode*/
78  vaddr = KERNEL_VADDR + PHYSMEM_START;
79  /* Step 1: set L0 and L1 page table entry */
80  boot_ttbr1_l0[GET_L0_INDEX(vaddr)] = ((u64)boot_ttbr1_l1) | IS_TABLE
81                                     | IS_VALID | NG;
82  boot_ttbr1_l1[GET_L1_INDEX(vaddr)] = ((u64)boot_ttbr1_l2) | IS_TABLE
83                                     | IS_VALID | NG;
84
85
86  /* Step 2: map PHYSMEM_START ~ PERIPHERAL_BASE with 2MB granularity */
87  vaddr = PHYSMEM_START;
88  for (; vaddr < PERIPHERAL_BASE; vaddr += SIZE_2M) {
89      boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
90          (vaddr) /* here we have modified vaddr, va = pa */
91          | UXN /* Unprivileged execute never */
92          | ACCESSED /* Set access flag */
93          | NG /* Mark as not global */
94          | INNER_SHARABLE /* Sharebility */
95          | NORMAL_MEMORY /* Normal memory */
96          | IS_VALID;
97  }
98
99  /* Step 2: map PERIPHERAL_BASE ~ PHYSMEM_END with 2MB granularity */
100 for (vaddr = PERIPHERAL_BASE; vaddr < PHYSMEM_END; vaddr += SIZE_2M) {
101     boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
102         (vaddr) /* here we have modified vaddr, va = pa */
103         | UXN /* Unprivileged execute never */
104         | ACCESSED /* Set access flag */
105         | NG /* Mark as not global */
106         | DEVICE_MEMORY /* Device memory */
107         | IS_VALID;
108 }
```

Listing 1: 练习题 2 代码

```
79  static struct page *split_page(struct phys_mem_pool *pool, u64 order,
80                                struct page *page)
81  {
82      /* LAB 2 TODO 2 BEGIN */
83      /*
84       * Hint: Recursively put the buddy of current chunk into
85       * a suitable free list.
86       */
87      if(page->allocated){
88          return NULL;
89      }
90
91      //delete the page from current free list
92      list_del(&page->node);
93      pool->free_lists[page->order].nr_free --;
94
95      while(page->order > order){
```

```
96     page->order --;
97     struct page *buddy = get_buddy_chunk(pool, page);
98     buddy->allocated = 0;
99     buddy->order = page->order;
100    //always need to add buddy to free list, might need to split original chunk recursively
101    list_add(&buddy->node, &pool->free_lists[buddy->order].free_list);
102    pool->free_lists[buddy->order].nr_free++;
103 }
104
105 //mark the page splited as allocated before return
106 page->allocated = 1;
107 return page;
108 /* LAB 2 TODO 2 END */
109 }
110
111 struct page *buddy_get_pages(struct phys_mem_pool *pool, u64 order)
112 {
113     /* LAB 2 TODO 2 BEGIN */
114     /*
115      * Hint: Find a chunk that satisfies the order requirement
116      * in the free lists, then split it if necessary.
117      */
118     u64 curr_order = order;
119     while(pool->free_lists[curr_order].nr_free == 0){
120         curr_order++;
121     }
122     if(curr_order >= BUDDY_MAX_ORDER){
123         return NULL;
124     }
125     struct page *page = list_entry(pool->free_lists[curr_order].free_list.next, struct page, node);
126     return split_page(pool, order, page);
127
128     /* LAB 2 TODO 2 END */
129 }
130
131 static struct page *merge_page(struct phys_mem_pool *pool, struct page *page)
132 {
133     /* LAB 2 TODO 2 BEGIN */
134     /*
135      * Hint: Recursively merge current chunk with its buddy
136      * if possible.
137      */
138
139     //check if current page can be merged
140     if(page->order >= BUDDY_MAX_ORDER-1 || page->allocated){
141         return page;
142     }
143     //check if its buddy page can be merged
144     struct page *buddy = get_buddy_chunk(pool, page);
145     if(buddy == NULL || buddy->allocated || buddy->order != page->order){
146         return page;
147     }
148
149     //after merge, the page must be pointing lower address
```

```
150     if(page > buddy){
151         struct page *tmp = buddy;
152         buddy = page;
153         page = tmp;
154     }
155     buddy->allocated = 1; //this might not be necessary but we still modify it for safety...
156
157     //delete the merged pages from the free lists
158     u64 order_before_merged = page->order;
159     pool->free_lists[order_before_merged].nr_free -= 2;
160     list_del(&page->node);
161     list_del(&buddy->node);
162
163     //
164     page->order++;
165     page->allocated = 0;
166     pool->free_lists[order_before_merged + 1].nr_free ++ ;
167     list_add(&page->node, &pool->free_lists[order_before_merged + 1].free_list);
168
169     return merge_page(pool, page);
170     /* LAB 2 TODO 2 END */
171 }
172
173 void buddy_free_pages(struct phys_mem_pool *pool, struct page *page)
174 {
175     /* LAB 2 TODO 2 BEGIN */
176     /*
177      * Hint: Merge the chunk with its buddy and put it into
178      * a suitable free list.
179     */
180
181     //if the page has been allocated, just return
182     if(page->allocated == 0){
183         return;
184     }
185
186     //always put the page into a free list, then call merge_page to check if it can be merged
187     page->allocated = 0;
188     u64 cur_order = page->order;
189     list_add(&page->node, &pool->free_lists[cur_order].free_list);
190     pool->free_lists[cur_order].nr_free++;
191
192     //once a page is freed, always need to check if we can merge with its buddy(recursively)
193     merge_page(pool, page);
194
195     /* LAB 2 TODO 2 END */
196 }
```

Listing 2: 练习题 4 代码

```
201     int query_in_pgtbl(void *pgtbl, vaddr_t va, paddr_t *pa, pte_t **entry)
202     {
203         /* LAB 2 TODO 3 BEGIN */
204         /*
```

```
205     * Hint: Walk through each level of page table using `get_next_ptp`,
206     * return the pa and pte until a L0/L1 block or page, return
207     * `-ENOMAPPING` if the va is not mapped.
208     */
209
210     ptp_t *cur_ptp = (ptp_t *)pgtbl;
211     pte_t *pte = NULL;
212     int result=0;
213
214     for(int level=0;level <=3; ++level){
215
216         result = get_next_ptp(cur_ptp, level, va, &cur_ptp, &pte, false);
217         if (result == -ENOMAPPING) {return result;}
218
219         if(level == 0){continue;}
220         if(level < 3){
221             if(result == BLOCK_PTP){//with L1 and L2 only return when result is a block ptp
222                 if( entry != NULL){
223                     *entry =pte;
224                 }
225                 switch (level) {
226                     case 1:
227                         *pa = virt_to_phys(cur_ptp) + GET_VA_OFFSET_L1(va);
228                         break;
229                     case 2:
230                         *pa = virt_to_phys(cur_ptp) + GET_VA_OFFSET_L2(va);
231                         break;
232                 }
233                 return 0;
234             }
235         }
236         if(level == 3){
237             if( entry != NULL){
238                 *entry =pte;
239             }
240             *pa = virt_to_phys(cur_ptp) + GET_VA_OFFSET_L3(va);
241         }
242     }
243     return 0;
244     /* LAB 2 TODO 3 END */
245 }
246
247 int map_range_in_pgtbl(void *pgtbl, vaddr_t va, paddr_t pa, size_t len,
248                       vmr_prop_t flags)
249 {
250     /* LAB 2 TODO 3 BEGIN */
251     /*
252     * Hint: Walk through each level of page table using `get_next_ptp`,
253     * create new page table page if necessary, fill in the final level
254     * pte with the help of `set_pte_flags`. Iterate until all pages are
255     * mapped.
256     */
257     ptp_t* ptp = NULL;
258     pte_t* pte = NULL;
```



```
259     int result = 0;
260     const vaddr_t end_va = va + len;
261     vaddr_t cur_va = va;
262     vaddr_t cur_pa = pa;
263
264     while(cur_va < end_va){
265         ptp = (ptp_t *) pgtbl;
266         for(int level=0; level<3; ++level){
267             result = get_next_ptp(ptp, level, cur_va, &ptp, &pte, true);
268             if(result == -ENOMAPPING){return result;}
269         }
270
271         pte = &(ptp->ent[GET_L3_INDEX(cur_va)]);
272         pte->l3_page.is_valid = 1;
273         pte->l3_page.is_page = 1;
274         pte->l3_page.pfn = cur_pa >> PAGE_SHIFT;
275         set_pte_flags(pte, flags, USER_PTE);
276
277         cur_va += PAGE_SIZE;
278         cur_pa += PAGE_SIZE;
279     }
280     return 0;
281     /* LAB 2 TODO 3 END */
282 }
283
284 int unmap_range_in_pgtbl(void *pgtbl, vaddr_t va, size_t len)
285 {
286     /* LAB 2 TODO 3 BEGIN */
287     /*
288      * Hint: Walk through each level of page table using `get_next_ptp`,
289      * mark the final level pte as invalid. Iterate until all pages are
290      * unmapped.
291      */
292     ptp_t* ptp = NULL;
293     pte_t* pte = NULL;
294     int result = 0;
295     const vaddr_t end_va = va + len;
296     vaddr_t cur_va = va;
297
298     while (cur_va < end_va) {
299         ptp = (ptp_t *) pgtbl;
300         for(int level=0; level<3;++level){
301             result = get_next_ptp(ptp, level, cur_va, &ptp, &pte, true);
302             if(result == -ENOMAPPING){return result;}
303         }
304         pte = &(ptp->ent[GET_L3_INDEX(cur_va)]);
305         pte->l3_page.is_valid = 0;
306
307         cur_va += PAGE_SIZE;
308     };
309
310     return 0;
311     /* LAB 2 TODO 3 END */
312 }
```

Listing 3: 练习题 5 代码

```
314 int map_range_in_pgtbl_huge(void *pgtbl, vaddr_t va, paddr_t pa, size_t len,
315                             vmr_prop_t flags)
316 {
317     /* LAB 2 TODO 4 BEGIN */
318     ptp_t* ptp=NULL;
319     pte_t* pte=NULL;
320     int result = 0;
321     const vaddr_t end_va = va + len;
322
323     const u64 SIZE_1G = PAGE_SIZE * L1_PER_ENTRY_PAGES;
324     const u64 SIZE_2M = PAGE_SIZE * L2_PER_ENTRY_PAGES;
325     const u64 SHIFT_1G = PAGE_SHIFT + PAGE_ORDER + PAGE_ORDER;
326     const u64 SHIFT_2M = PAGE_SHIFT + PAGE_ORDER;
327
328     while (va + SIZE_1G < end_va) {
329         ptp = (ptp_t *)pgtbl;
330         result = get_next_ptp(ptp, 0, va, &ptp, &pte, true);
331         if(result == -ENOMAPPING){return result;}
332
333         pte = &(ptp->ent[GET_L1_INDEX(va)]);
334         pte->l1_block.is_valid = 1;
335         pte->l1_block.is_table = 0;
336         pte->l1_block.pfn = pa >> SHIFT_1G;
337         set_pte_flags(pte, flags, USER_PTE);
338
339         va += SIZE_1G;
340         pa += SIZE_1G;
341         len -= SIZE_1G;
342
343     };
344
345     while (va + SIZE_2M < end_va) {
346         ptp = (ptp_t *)pgtbl;
347         for(int level=0; level<2;++level){
348             result = get_next_ptp(ptp, level, va, &ptp, &pte, true);
349             if(result == -ENOMAPPING){return result;}
350         }
351
352         pte = &(ptp->ent[GET_L2_INDEX(va)]);
353         pte->l2_block.is_valid = 1;
354         pte->l2_block.is_table = 0;
355         pte->l2_block.pfn = pa >> SHIFT_2M;
356         set_pte_flags(pte, flags, USER_PTE);
357
358         va += SIZE_2M;
359         pa += SIZE_2M;
360         len -= SIZE_2M;
361     };
362
363     map_range_in_pgtbl(pgtbl, va, pa, len, flags);
```

```
364
365     return 0;
366     /* LAB 2 TODO 4 END */
367 }
368
369 int unmap_range_in_pgtbl_huge(void *pgtbl, vaddr_t va, size_t len)
370 {
371     /* LAB 2 TODO 4 BEGIN */
372     ptp_t* ptp=NULL;
373     pte_t* pte=NULL;
374     int result = 0;
375     const vaddr_t end_va = va + len;
376
377     const u64 SIZE_1G = PAGE_SIZE * L1_PER_ENTRY_PAGES;
378     const u64 SIZE_2M = PAGE_SIZE * L2_PER_ENTRY_PAGES;
379     const u64 SHIFT_1G = PAGE_SHIFT + PAGE_ORDER + PAGE_ORDER;
380     const u64 SHIFT_2M = PAGE_SHIFT + PAGE_ORDER;
381
382     while (va + SIZE_1G < end_va) {
383         ptp = (ptp_t *)pgtbl;
384         result = get_next_ptp(ptp, 0, va, &ptp, &pte, true);
385         if(result == -ENOMAPPING){return result;}
386
387         pte = &(ptp->ent[GET_L1_INDEX(va)]);
388         pte->l1_block.is_valid = 0;
389
390         va += SIZE_1G;
391         len -= SIZE_1G;
392     };
393
394     while (va + SIZE_2M < end_va) {
395         ptp = (ptp_t *)pgtbl;
396         for(int level=0; level<2;++level){
397             result = get_next_ptp(ptp, level, va, &ptp, &pte, true);
398             if(result == -ENOMAPPING){return result;}
399         }
400         pte = &(ptp->ent[GET_L2_INDEX(va)]);
401         pte->l2_block.is_valid = 0;
402
403         va += SIZE_2M;
404         len -= SIZE_2M;
405     };
406
407     unmap_range_in_pgtbl(pgtbl, va, len);
408
409     return 0;
410     /* LAB 2 TODO 4 END */
411 }
```

Listing 4: 练习题 6 代码