

Lab 2: Memory Management

习题解答

0. 一些宏的定义

```
1 // inc/mmu.h
2
3 /*
4  * A virtual address 'va' has a four-part structure as follows:
5  * +-----9-----+-----9-----+-----9-----+-----9-----+-----12-----+
6  * | Level 0 | Level 1 | Level 2 | Level 3 | Offset within Page |
7  * | Index  | Index  | Index  | Index  |
8  * +-----+-----+-----+-----+-----+
9  * \PTX(va, 0)/\PTX(va, 1)/\PTX(va, 2)/\PTX(va, 3)/
10 */
11
12 #define PGSIZE 4096
13 #define PGSHIFT 12
14 #define L0SHIFT 39
15 #define L1SHIFT 30
16 #define L2SHIFT 21
17 #define L3SHIFT 12
18 #define ENTRYSZ 512
19
20 #define PTX(level, va) (((uint64_t)(va) >> (39 - 9 * level)) & 0x1FF)
21 #define L0X(va) (((uint64_t)(va) >> L0SHIFT) & 0x1FF)
22 #define L1X(va) (((uint64_t)(va) >> L1SHIFT) & 0x1FF)
23 #define L2X(va) (((uint64_t)(va) >> L2SHIFT) & 0x1FF)
24 #define L3X(va) (((uint64_t)(va) >> L3SHIFT) & 0x1FF)
25
26 /* accessibility */
27 #define PTE_P (1<<0) /* valid */
28 #define PTE_BLOCK (0<<1)
29 #define PTE_PAGE (1<<1)
30 #define PTE_TABLE (1<<1) /* entry gives address of the next level of translation table */
31 #define PTE_KERNEL (0<<6) /* privileged, supervisor EL1 access only */
32 #define PTE_USER (1<<6) /* unprivileged, EL0 access allowed */
33 #define PTE_RW (0<<7) /* read-write */
34 #define PTE_RO (1<<7) /* read-only */
35 #define PTE_AF (1<<10) /* P2066 access flags */
36 // Address in page table or page directory entry
37 #define PTE_ADDR(pte) ((uint64_t)(pte) & ~0xFFF)
38 #define PTE_FLAGS(pte) ((uint64_t)(pte) & 0xFFF)
```

```

1 // inc/memlayout.h
2
3 #define EXTMEM 0x80000          /* Start of extended memory */
4 #define PHYSTOP 0x3F000000    /* Top physical memory */
5
6 #define KERNBASE 0xFFFF000000000000 /* First kernel virtual address */
7 #define KERNLINK (KERNBASE + EXTMEM) /* Address where kernel is linked */
8
9 #define V2P_WO(x) ((x) - KERNBASE) /* Same as V2P, but without casts */
10 #define P2V_WO(x) ((x) + KERNBASE) /* Same as P2V, but without casts */
11
12 #ifndef __ASSEMBLER__
13
14 #   include <stdint.h>
15 #   define V2P(a) (((uint64_t) (a)) - KERNBASE)
16 #   define P2V(a) ((void *)(((char *) (a)) + KERNBASE))
17
18 #endif

```

1. 物理内存分配器

完成物理内存分配器的分配函数 `kalloc` 以及回收函数 `kfree`。

由源代码可知，物理页表位于 `kmem.free_list` 这个链表里。

```

1 // kern/kalloc.c
2 struct {
3     struct run* free_list; /* Free list of physical pages */
4 } kmem;

```

对于函数 `kalloc`，我们所需要的就是从 `free_list` 链表中取出头节点返回。因此不难得到函数 `kalloc` 的代码如下：

```

1 // kern/kalloc.c
2
3 /*
4  * Allocate one 4096-byte page of physical memory.
5  * Returns a pointer that the kernel can use.
6  * Returns 0 if the memory cannot be allocated.
7  */
8 char*
9 kalloc()
10 {
11     struct run* p;
12     p = kmem.free_list;
13     if (p) kmem.free_list = p->next;
14     return (char*)p;
15 }

```

对于函数 `kfree`，则是函数 `kalloc` 的逆过程，即将 free 后的物理页插回 `free_list` 链表头部。

```

1 // kern/kalloc.c
2
3 /* Free the page of physical memory pointed at by v. */
4 void
5 kfree(char* v)
6 {
7     struct run* r;
8
9     if ((uint64_t)v % PGSIZE || v < end || V2P(v) ≥ PHYSTOP)
10         panic("kfree: invalid address: 0x%p\n", V2P(v));
11
12     /* Fill with junk to catch dangling refs. */
13     memset(v, 1, PGSIZE);
14
15     r = (struct run*)v;
16     r->next = kmem.free_list;
17     kmem.free_list = r;
18 }

```

这一部分并不复杂。

2. 页表管理

完成物理地址的映射函数 `map_region` 以及回收页表物理空间函数 `vm_free`。

本过程中，我们需要构建 `ttbr0_el1` 页表，并将其映射到虚拟地址（高地址）。

在完成映射函数 `map_region` 之前，我们需要先按照要求完成函数 `pgdir_walk`。根据注释，函数 `pgdir_walk` 所做的事情是根据提供的虚拟地址 `va` 找到相应的页表，如果途径的页表项（PDE, Page Directory Entry）不存在，则分配（allocate）一个新的页表项。

这里我将分配新页表项的逻辑单独封装成一个函数 `pde_validate`，以提升代码的可读性。

```

1 // kern/vm.c
2
3 // If the page is invalid, then alloc a new one. Return NULL if failed.
4 static uint64_t*
5 pde_validate(uint64_t* pde, int64_t alloc)
6 {
7     if (!(*pde & PTE_P)) { // if the page is invalid
8         if (!alloc) return NULL;
9         char* p = kalloc();
10        if (!p) return NULL; // allocation failed
11        memset(p, 0, PGSIZE);
12        *pde = V2P(p) | PTE_P | PTE_PAGE | PTE_USER | PTE_RW;
13    }
14    return pde;
15 }

```

需要注意的是，PDE 中前半段保存的地址应当为物理地址（低地址）。

函数 `pgdir_walk` 中我们进行了 3 次循环。每次循环，我们根据当前所在层级（level）的 PDE 所保存的物理地址，将其转换为虚拟地址后，再以 `va` 相应的片段为索引，找到下一级 PDE 所在的虚拟地址。过程中如果 PDE 不存在，则通过函数 `pde_validate` 分配一个新的。

```
1 // kern/vm.c
2
3 /*
4  * Given 'pgdir', a pointer to a page directory, pgdir_walk returns
5  * a pointer to the page table entry (PTE) for virtual address 'va'.
6  * This requires walking the four-level page table structure.
7  *
8  * The relevant page table page might not exist yet.
9  * If this is true, and alloc == false, then pgdir_walk returns NULL.
10  * Otherwise, pgdir_walk allocates a new page table page with kalloc.
11  * - If the allocation fails, pgdir_walk returns NULL.
12  * - Otherwise, the new page is cleared, and pgdir_walk returns
13  *   a pointer into the new page table page.
14  */
15 static uint64_t*
16 pgdir_walk(uint64_t* pgdir, const void* va, int64_t alloc)
17 {
18     uint64_t sign = ((uint64_t)va >> 48) & 0xFFFF;
19     if (sign != 0 && sign != 0xFFFF) return NULL;
20
21     uint64_t* pde = pgdir;
22     for (int level = 0; level < 3; ++level) {
23         pde = &pde[PTX(level, va)]; // get pde at the next level
24         if (!(pde = pde_validate(pde, alloc))) return NULL;
25         pde = (uint64_t*)P2V(PTE_ADDR(*pde));
26     }
27     return &pde[PTX(3, va)];
28 }
```

为什么 4 级页表只进行了 3 次循环，是因为最后一级我们只需要返回 PDE 中地址所指向的页表（PTE, Page Table Entry）地址即可。

对于回收页表物理空间函数 `vm_free`，我们需要遍历 4 级页表，并将其中的节点全部 free 掉。

```

1 // kern/vm.c
2
3 /*
4  * Free a page table.
5  *
6  * Hint: You need to free all existing PTEs for this pgdir.
7  */
8 void
9 vm_free(uint64_t* pgdir, int level)
10 {
11     // cprintf("vm_free: currently at 0x%p at level %d.\n", pgdir, 4 - level);
12     if (!pgdir || level < 0) return;
13     if (PTE_FLAGS(pgdir)) panic("vm_free: invalid pgdir.\n");
14     if (!level) {
15         // cprintf("vm_free: free 0x%p at level %d.\n", pgdir, level);
16         kfree((char*)pgdir);
17         return;
18     }
19     for (uint64_t i = 0; i < ENTRYSZ; ++i) {
20         // cprintf("[%lld]: 0x%llx\n", i, pgdir[i]);
21         if (pgdir[i] & PTE_P) {
22             uint64_t* v = (uint64_t*)P2V(PTE_ADDR(pgdir[i]));
23             // cprintf("vm_free: free 0x%p at level %d.\n", v, 5 - level);
24             vm_free(v, level - 1);
25         }
26     }
27     kfree((char*)pgdir);
28 }

```

这里我们使用了递归的写法。需要注意的是，`pgdir` 中 PDE 保存的地址为物理地址，在传给函数 `kfree` 前需要先转换为虚拟地址。代码中，`ENTRYSZ` 的值为 `512`，表示 4 KB 页表中的 PDE 项数（每项的大小为 $64 \text{ bit} = 8 \text{ B}$ ）。