Lab 2: Memory Management

习题解答

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。

```
// kern/kalloc.c

struct {
    struct run* free_list; /* Free list of physical pages */
} kmem;
```

对于函数 `kalloc` ，我们所需要做的就是从 `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：

```
// kern/kalloc.c

/*
 * Allocate one 4096-byte page of physical memory.
 * Returns a pointer that the kernel can use.
 * Returns 0 if the memory cannot be allocated.
 */
char*
kalloc()
{
    struct run* p;
    p = kmem.free_list;
    if (p) kmem.free_list = p->next;
    return (char*)p;
}
```

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。

```c
// kern/kalloc.c

/* Free the page of physical memory pointed at by v. */
void
kfree(char* v)
{
    struct run* r;

    if ((uint64_t)v % PGSIZE || v < end || V2P(v) ≥ PHYSTOP)
        panic("kfree: invalid address: 0x%p\n", V2P(v));

    /* Fill with junk to catch dangling refs. */
    memset(v, 1, PGSIZE);

    r = (struct run*)v;
    r->next = kmem.free_list;
    kmem.free_list = r;
}
```

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。

```c
// kern/vm.c

/*
 * If the page is invalid, then allocate a new one. Return NULL if failed.
 */
static uint64_t*
pde_validate(uint64_t* pde, int64_t alloc)
{
    if (!(*pde & PTE_P)) {  // if the page is invalid
        if (!alloc) return NULL;
        char* p = kalloc();
        if (!p) return NULL;  // allocation failed
        memset(p, 0, PGSIZE);
        *pde = V2P(p) | PTE_P | PTE_PAGE | PTE_USER | PTE_RW;
    }
    return pde;
}
```

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。

```
// kern/vm.c

/*
 * Given 'pgdir', a pointer to a page directory, pgdir_walk returns
 * a pointer to the page table entry (PTE) for virtual address 'va'.
 * This requires walking the four-level page table structure.
 *
 * The relevant page table page might not exist yet.
 * If this is true, and alloc == false, then pgdir_walk returns NULL.
 * Otherwise, pgdir_walk allocates a new page table page with kalloc.
 *   - If the allocation fails, pgdir_walk returns NULL.
 *   - Otherwise, the new page is cleared, and pgdir_walk returns
 *     a pointer into the new page table page.
 */
static uint64_t*
pgdir_walk(uint64_t* pgdir, const void* va, int64_t alloc)
{
    uint64_t sign = ((uint64_t)va >> 48) & 0xFFFF;
    if (sign != 0 && sign != 0xFFFF) return NULL;

    uint64_t* pde = pgdir;
    for (int level = 0; level < 3; ++level) {
        pde = &pde[PTX(level, va)];  // get pde at the next level
        if (!(pde = pde_validate(pde, alloc))) return NULL;
        pde = (uint64_t*)P2V(PTE_ADDR(*pde));
    }
    return &pde[PTX(3, va)];
}
```

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。

```c
// kern/vm.c

/*
 * Free a page table.
 */
void
vm_free(uint64_t* pgdir, int level)
{
    if (!pgdir || level < 0) return;
    if (PTE_FLAGS(pgdir)) panic("vm_free: invalid pgdir.\n");
    if (!level) {
        kfree((char*)pgdir);
        return;
    }
    for (uint64_t i = 0; i < ENTRYSZ; ++i) {
        if (pgdir[i] & PTE_P) {
            uint64_t* v = (uint64_t*)P2V(PTE_ADDR(pgdir[i]));
            vm_free(v, level - 1);
        }
    }
    kfree((char*)pgdir);
}
```

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 $4\ \mathrm{KB}$ 页表中的 PDE 项数（每项的大小为 $64\ \mathrm{bit} = 8\ \mathrm{B}$）。

测试环境

- OS: Ubuntu 18.04.5 LTS (WSL2 4.4.0-19041-Microsoft)
- Compiler: gcc version 8.4.0 (Ubuntu/Linaro 8.4.0-1ubuntu1~18.04)
  - Target: aarch64-linux-gnu
- Debugger: GNU gdb 8.2 (Ubuntu 8.2-0ubuntu1~18.04)
  - Target: aarch64-linux-gnu
- Emulator: QEMU emulator version 5.0.50
- Using GNU Make 4.1