Lab 2: Memory Management

18307130024 Jimmy Tan

1. `AArch64` Virtual Memory System Architecture(`VMSAv8-64`)

The reference of this part is `D5` of `ARMv8 Reference Manual`.

The accessible virtual memory ranges from `0x0` to `0x0000FFFFFFFFFFFF` for a `48-bit` VA.

1.1 Memory Translation granule size

`VMSAv8-64` supports translation granule sizes of `4KB`, `16KB`, and `64KB`. The memory translation granule size defines the maximum size of a translation table and the memory page size.

Each entry in the translation table is 64 bits large (`8B`), and one translation table is stored in a single page. Therefore, the maximum number of entries in a translation table is $\frac{page\ size}{8}$.

This means a single translation table lookup can resolve only a limited number of address bits, the process of translation involves multiple **levels** of translation.

1.2 `VMSAv8-64` translation table format descriptors

This section briefly shows the meaning of each bit in a page table entry.

Block descriptors are skipped in this section, for the lab is focused on pages.

1.2.1 Level 0, 1, 2 page table

- `Bits[0]` shows the validity.
- `Bits[1]` is 1 if the page table entry points to a next-level page table and 0 if the page table entry points to a block.
- `Bits[47:12]` are `bits[47:12]` of the address of the next-level table.

- `Bits[59]` is the `PXN` limit for subsequent levels of lookup and is valid only for a stage 1 translation that can support two VA ranges.
- `Bits[60]` is the `XNTable` bit and restricts the execute-never control.
- `Bits[62:61]` are the access permissions limit for subsequent levels of lookup.
- `Bits[63]` is the `NSTable` bit and specifies the Security state for subsequent levels of lookup.

1.2.2 Level 3

- `Bits[0]` shows the validity.
- `Bits[1]` is 1 if the page table entry points to the page and 0 is the reserved value.
- `Bits[47:12]` are `bits[47:12]` of the address of the page.
- `Bits[4:2]` are Stage 1 memory attributes index field, for the `MAIR_ELx`.
- `Bits[5]` is the non-secure bit.
- `Bits[7:6]` are the data access permissions bits (`AP[2:1]`). `Bits[6]` is `RES1` when stage 1 translations can support only one VA range.
  - `AP[2]` = 1 shows it's read-only.
  - `AP[2]` = 0 shows it's read/write.
- `Bits[9:8]` are share-ability field.
- `Bits[10]` is the access flag.
- `Bits[11]` is the not global bit.
- `Bits[16]` is the block translation entry.
- `Bits[50]` is the guarded page bit.
- `Bits[51]` is the dirty bit modifier.
- `Bits[52]` is the a hint bit indicating that the translation table entry is one of a contiguous set or entries, that might be cached in a single `TLB` entry.

2. Physical Memory Allocator

The kernel maintains the free pages by a **linked list**. The first `64bit` of the `4KB` free page stores the pointer to the next free page.

The page at physical address `0x0` cannot be free, so `NULL` is used to identify the last free page in the linked list.

2.1 Page Allocation

The `kalloc` function has no parameters and returns a pointer of the first byte of the page.

Page allocation involves two steps:

- Remove the free page from the linked list mentioned above. (Lock is needed).
- Fill in the page with junk data.

Take the first item of the linked list costs $O(1)$ time, while take the last item of the linked list costs $O(n)$ time where n is the size of the linked list.

2.2 Physical Memory Free

The `kfree` function has a pointer at the page to be freed as the parameter and returns nothing.

Freeing a page involves two steps:

- Check the validity of the address pointing at the page to be freed
  - Whether the address is aligned
  - Whether the address is above `bss` section
  - Whether the physical address is above the limit.
- Add the pointer to the linked list. (Lock is needed)

Adding the pointer to the head of the linked list costs $O(1)$ time, but it may happen that a single page exists twice in the free list. This implementation requires the kernel programmer to think twice before calling the `kfree` function.

Adding the pointer to the tail of the linked list costs $O(n)$ time, where n is the size of the linked list. While searching for the tail of the linked list, it can check whether the page has existed in the linked list.

3. Page Table Manager

In this lab, block memory (`2MB` or `1GB`) is not supported. All table entries in level 0, 1 and 2 points to another level of page table.

3.1 Walk

The parameters of `pgdir_walk` function includes a pointer to a page directory, a virtual address and an allocation enable signal.

First, check whether the virtual address is above the maximum virtual address limit.

Then, iterate three times:

- Get the page table entry according to the index bits in the virtual address, `VA[47:39]`, `VA[38:30]`, `VA[29:21]` for level 0, 1, 2 page table.
- If the valid bit of the page table entry is 0,
  - If allocation enable is 1, allocate a new page for the page table, and modify the page table entry. If this allocation failed, the function return 0.
  - If allocation enable is 0, the function return 0.
- Step into the next page table.

When the level 3 page table is reached, return the physical page pointer.

This lab does not support block memory. If a page table entry is found to point to a block instead of the next level page table (`pte[1] == 0 && pte[0] == 1`), a error is reported.

3.2 Map

The parameters of `map_region` function includes a pointer to a page directory, a virtual address with a physical address, a size and the permission of the page table entry to be created.

First, calculate the address of the first page and the last page to be allocated according to the provided virtual address and the size.

Then, create the map between the physical address and the virtual address from the first page to the last page. If the map had been created, report an error.

3.3 Page Table Free

The parameters of `vm_free` function includes a pointer to a page directory and the page table level.

It's implemented by recursive function calls.

For level 0, 1, 2 page table, iterating all its valid page table entry and call `vm_free` to free the next page table, then free itself.

For level 3 page table, iterating all its valid page table entry and call `kfree`, and then free itself.

From my perspective, the `vm_free` function should be private function. If someone calls the function with the `level` parameter as 1, the kernel cannot find the page table entry in level 0 page table pointing to the given page table. It can be improved to:

```
static void
vm_free_level(uint64_t* pgdir, int level);

void
vm_free(uint64_t* pgdir)
{
    vm_free_level(pgdir, 0);
}
```

This improvement can ensure more security than previous.

3.4 Test for `walk` and `map`

The following code shows a test for function `map_region` and `pgdir_walk`.

```
void
test_map_region()
{
    // Write data to physical address
    *((int64_t*)P2V(0)) = 0xac;

    // Create a page table
    char* p = kalloc();

    // The page is initially filled with junk data.
    // Zero initializaion is required.
    memset(p, 0, PGSIZE);

    // set a map between the selected virtual address and physical address
    map_region((uint64_t*)p, (void*)0x1000, PGSIZE, 0, 0);

    // Set the ttbr0_el1 register
    asm volatile("msr ttbr0_el1, %[x]": : [x] "r"(V2P(p)));

    // Getting the value through virtual address
    if (*((int64_t*)0x1000) == 0xac) {
        cprintf("Test_Map_Region Pass!\n");
    }
    else {
        cprintf("Test_Map_Region Fail!\n");
    }
```

```
}
```