Lab 2: Memory Management

Sriram V - CS11B058

Exercise 1

- Implemented the code for `boot_alloc()`, `mem_init()`, `page_init()`, `page_alloc()` and `page_free()` in `kern/pmap.c`.
- `assert()` statements are used wherever necessary to ensure that assumptions made for a function are true.

1. `boot_alloc()`:
    - `boot_alloc()` is used for memory allocation only when JOS is setting up its virtual memory system. As paging is enabled, memory is supposed to be allocated only in terms of pages, using the `page_alloc()` function.
    - `boot_alloc()` initializes `nextfree` (a pointer to the virtual address of the next byte of free memory) the first time it is called, by pointing to the first page (of size `PGSIZE`) immediately after the kernel's bss segment (whose end is pointed to by `end`, a magic symbol automatically generated by the linker).
    - If `n>0`, `boot_alloc()` allocates n bytes of memory (rounded up to the nearest multiple of `PGSIZE`) and returns a pointer to the first byte of this chunk of allocated memory.
    - If `n=0`, it returns a pointer to the address pointed to by `nextfree`.
    - While allocating memory, if the virtual address exceeds `0xf0400000` (The rudimentary mapping done in `kern/entry.S` is from `[0x0, 0x00400000)` to `[0xf0000000, 0xf0400000)`), then the kernel panics and prints an `out of memory` message.
2. `mem_init()`:
    - The page directory is allocated memory first, using `boot_alloc()`, and its permissions are set. It is aloso recursively inserted in itself as a page table. Its permissions are set as well.
    - The first line of code we had to add in was to use `boot_alloc()` to allocate memory for `npages` of `PageInfo` structures. This array is pointed to by the `pages` variable.
    - The kernel has a `PageInfo` structure associated with every physical page, to store the page's details such as the physical address and the number of references to the page.
    - `npage` and `npage_basemem` are variables that give us the ammount of base and physical memory (respectively) in terms of number of pages. These values are calculated in `i386_detect_memory()`.
    - Then `page_init()` is called to initialise `pages` with the data associated with the currently allocated pages.
    - Checks are then performed.
3. `page_init()`:
    - The first page is ignored as it is in use. (It contains the real-mode IDT and BIOS structures. We preserve them in case we ever need them.)
    - The rest of the base memory is then marked as free by adding them to `page_free_list`. Page refs are set to zero.
    - We don't allocate pages for the IO hole, which is memory meant for IO devices.
    - We query the virtual address of the next byte of free memory (using `boot_alloc()`), and set all the pages from this one onwards, right up to the last page.
4. `page_alloc()`:
    - This function first gets the address of the first free page (using the pointer to the linked list of free pages - `page_free_list`).

- It shifts this pointer to the next free page as the topmost one has now been allocated.
- If `alloc_flags` has a value of `ALLOC_ZERO`, then the allocated page is filled with `\0`s using the `memset()` function.
- `memset()` requires the kernel virtual address of the page, which is obtained by using the `page2kva()` function.
- A `PageInfo` structure is returned.

5. `page_free()`:
   - This function returns a page back to the free list.
   - We first ensure that the `PageInfo` exists, and that the number of references to it are zero, so that we don't accidentally free up a page that still has references to it.
   - Once these two points are asserted, we add the page to the list of free pages (`page_free_list`).

Exercise 2

- **Page Translation:**
  - It is in effect only when the `PG` bit of `CR0` is set.
  - A virtual address is translated into a linear address by the segmentation mechanism into a linear address.
  - This linear address maps to a physical address by means of a page translation mechanism.
  - The first 10 bits of the 32-bit linear address points gives the offset of the entry in the Page Directory. This entry points to a Page Table.
  - The next 10 bits of the linear address gives the offset of the Page Table Entry, which points to the physical address of the page being referred to.
  - The last 12 bits of the linear address gives the offset of the physical address within the page.
- **Page-based Protection:**
  - Page-level protection is implemented by setting certain bits in the 32-bit entries in the Page Directory and Page Tables.
  - It is used for both inter and intra-process protection.

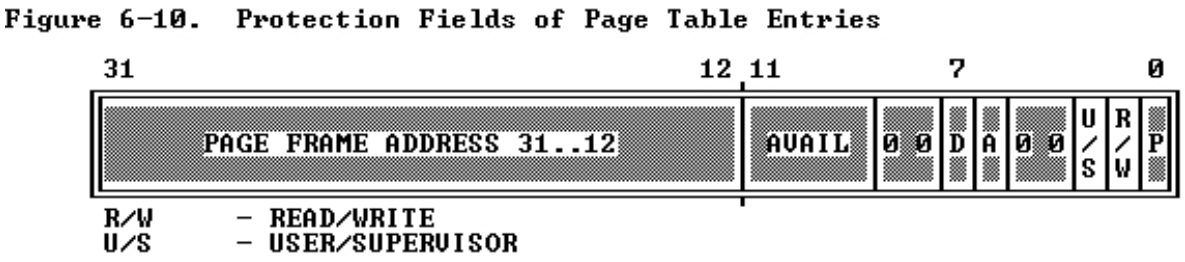

Figure 1: Protection Fields of Page Table Entries

- **Segmentation:**
  - Segmentation is used for intra-process protection.
  - Different portions of the program are put into different segments.
  - While converting the virtual address to a linear address, the 16 bits are used to select the segment, and the 32 bits are used to offset into the segment.
  - The segment selector points to a segment descriptor that contains the base address of the segment, as well as other details such as protection and permission bits.

Exercise 3

- Used `info pg` in the QEMU monitor (accessed by using `Ctrl-a c`) to view the mapping between the virtual addresses and the physical addresses, and the permissions of the pages.

- Used `xp/i <physical_address>` in the QEMU monitor, and `x/i <corresponding_virtual_address>` in GDB to ensure that both the addresses accessed the same memory location.

Questions

1. **Assuming that the following JOS kernel code is correct, what type should variable `x` have, `uintptr_t` or `physaddr_t`?**

```
mystery_t x;
char* value = return_a_pointer();
*value = 10;
x = (mystery_t) value;
```

- `x` should be of the type `uintptr_t`, since all addresses referred to in C code are virtual addresses and not physical addresses.

Exercise 4

- Implemented code in `kern/pmap.c` for `pgdir_walk()`, `boot_map_region()`, `page_lookup()`, `page_remove()` and `page_insert()`.

1. `pgdir_walk()`:
   - This function returns a pointer to the Page Table Entry, given a pointer to a page directory and a linear address.
   - Macros from `inc/mmu.h` are made use of to manipulate the page directory and table entries.
   - `PDX` is used to get the Page Directory Index for the virtual address `va`.
   - This index is used to then obtain the Page Directory Entry at that index.
   - A check is done if a Page Table exists at that entry. If it does, we get its kernel virtual address.
   - If it doesn't and the `create` parameter is `1`, then a new page is allocated using `page_alloc()` and filled with `\0`s.
   - This created page is then inserted into the Page Directory at the Page Directory Index with the appropriate permissions, and its reference count is incremented by one.
   - In either case, the Page Table Index (obtained by using `PTX` on `va`) is used to index into the page table, to access the corresponding Page Table Entry.
   - A pointer to this entry is returned.
   - In any other case, a `NULL` is returned.
   - We use `pte_t` and `pde_t` (`typedef`s of `uint_32`) to explicitly specify if we are dealing with a page directory or table entry.
2. `boot_map_region()`:
   - This function maps a virtual address range to a physical address region with certain permissions for the pages.
   - Assertions are made that the virtual address and physical address starts at a multiple of `PGSIZE`, and that the range is a multiple of `PGSIZE` as well.
   - Starting from the lower value of the virtual address range, we identify the corresponding page table entry (using `pgdir_walk` on the Page Directory, and create a Page Table for the entry if one doesn't exist), set the entry to associate with the appropriate physical address for that page, set the permissions for that entry, and move on to the next page.
   - We do this until the entire virtual address range specified has been mapped on to the provided physical address range.
3. `page_lookup()`:
   - This function returns the `PageInfo` structure located at a virtual address `va`, given the Page Directory `pgdir`.

- `pgdir_walk` is used to fetch the pointer to the page table entry for the virtual address. Since it is a lookup, the `create` parameter is set to zero.
- If the page table doesn't exist or a page doesn't exist at the corresponding PTE, we return `NULL`.
- If it does exist, we find the physical address of the PTE (using the macro `PTE_ADDR()`) and use the `pa2page()` function to find the corresponding `PageInfo` structure, which we then return as the result.

4. `page_remove()`:

   - This function unmaps the page located at the virtual address `va`. If a physical page doesn't exist at this location, it fails silently.
   - `page_lookup` is used to identify the `PageInfo` structure associated with the page at the virtual address `va`.
   - If a Page Table or Page Table Entry doesn't exist, the funtion `return`s.
   - Else, we decrement the reference count for the structure. This is managed using the `page_decref()` function, which also frees up the page (using `page_free()`) if the reference count drops to zero.
   - The corresponding Page Table Entry for this page is then set to zero, and the TLB is invalidated for the passed virtual address by the `tlb_invalidate()` function.

5. `page_insert()`:

   - This function maps a `PageInfo` object `pp` to a virtual address `va`. This is an alternative to the `boot_map_region()` function, which maps physical address ranges to virtual address ranges. This can be used when the page object is known, but its physical address isn't.
   - First we assert that the Page Directory and the `PageInfo` object exist.
   - `pgdir_walk()` is then used (with `create=1`) to find the Page Table Entry for the passed virtual address.
   - `-E_NO_MEM` is returned if the Page Table doesn't exist. (This can happen only if we run out of memory, as we are explicitly requesting the `pgdir_walk()` function to create a page table if it doesn't exist.)
   - If the Page Table exists and the Page Table Entry exists as well, we check if the physical address of the PTE is the same as the physical address of the `PageInfo` object to be inserted.
   - This is a corner case. If we handle it like any other page and remove it before adding it back in, we may end up deleting the `PageInfo` object itself.
   - This can happen if the object has its reference count equal to 1. In such a case, deleting the PTE using the `page_remove()` function decrements the reference count for the object to zero, which returns the page to the list of free pages, even though it is to be reallocated to the same virtual address.
   - Hence, as the object has been marked as a free page, we can no longer access the page to increment its reference count and assign permissions to it.
   - If it is, it is a case of reinsertion of the same page at the same virtual address in the same Page Directory. This means that only a permission change for the page is to be effected. Hence, we use a `goto` to jump to the permission change portion.
   - If the physical address of the PTE doesn't match that of the page to be inserted, then the page that exists at the virtual address is a different page from the one to be inserted. Hence, we call `page_remove()` to remove the page at this location and to invalidate the TLB.
   - We now set the Page Table Entry to point to the physical address of the `PageInfo` object and set the permissions of the PTE. For the corner case, this is the only step that happens, i.e., only a permission change occurs.
   - A `0` is returned to signal a successful page insertion.

Exercise 5

- Now that all the helper functions have been filled in, we can duly fill in the rest of the `mem_init()` function after the call to `check_page()`.
- The code we write now initializes the kernel address space.
- The kernel virtual address space is mapped to corresponding physical address regions by making use of the `map_boot_region()` function.
- The `pages` array that contains all the `PageInfo` objects is mapped to the virtual address given by the macro `UPAGES`.

- This region is provided as Read Only for both the kernel and the user, as it is a means for the user to access the kernel data structures.
- The actual physical address that this virtual address maps to has RW permissions for the kernel, so that the kernel can modify the data structures here. This region is inaccessible by the user.
- The changes made here are reflected at the mapped virtual address, which can be read by the user. Hence, a protected interface for kernel methods is provided to the user.
- The next mapping maps the kernel stack (physical address is given by the `bootstack` variable set in `kern/entry.S`) to the virtual address `KSTACKTOP`. The region mapped is of size `KSTKSIZE`, rather than `PTSIZE`.
- This is because the region `[KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE)` is meant to be a 'guard page' - this page doesn't have a corresponding physical page (not backed by the physical memory), so if the kernel overflows its stack, a fault will occur rather than overwriting of memory (since virtual memory to write into exists, but corresponding physical memory doesn't).
- Since it is the kernel stack, the kernel has RW permissions, but the user has none.
- The final mapping is to map the virtual address range `[KERNBASE, 2^32)` to the physical address range `[0, 2^32 - KERNBASE)`. This is done to replace the rudimentary mapping that was done in `kern/entry.S`, before the page directory and tables were allocated and initialised.
- Also, when we switch from the rudimentary mapping to the full-fledged page directory that has just been created, we will not have any issues, as the virtual addresses pointed to by the old page directory (in range `[0xf0000000, 0xf0400000)`) will still map to the same physical address range `[0x00000000, 0x00400000)` as before.

Questions

2. **What entries (rows) in the page directory have been filled in at this point? What addresses do they map and where do they point? In other words, fill out the table as much as possible:**

- Filled table:

Entry	Base Virtual Address	Points to (logically)
1023	0xffc00000	Page table for top 4MB of phys memory
.	.	.
.	.	.
.	.	.
960	0xf0000000	KERNBASE
959	0xefc00000	VPT,KSTACKTOP
958	0xef800000	ULIM
957	0xef400000	UVPT
956	0xef000000	UPAGES (RO PAGES)
955	0xeec00000	UTOP,UENVS/UXSTACKTOP (RO ENVS)
.	.	.
.	.	.
.	.	.
2	0x00800000	UTEXT
1	0x00400000	UTEMP
0	0x00000000	Real-mode IDT, BIOS etc.

3. **We have placed the kernel and user environment in the same address space. Why will user programs not be able to read or write the kernel's memory? What specific mechanisms protect the kernel memory?**

- The virtual memory is divided into segments by ULIM and UTOP - `(ULIM, 4GB)` is Kernel only, `(UTOP, ULIM]` can be accessed by both the Kernel and User environment (Read Only) and `[0x0, UTOP]` is user environment space.
- These memory spaces are protected by the permission bits, such as `PTE_W` (writeable) and `PTE_U` (user), which are set in the page table/directory entries.
- The Current Privilege Level (CPL) is used to identify whether in privileged mode (`CPL=0` - the kernel) or not (`CPL=3` - user processes). Accordingly, certain pages in the memory are (or are not) accessible.

4. **What is the maximum amount of physical memory that this operating system can support? Why?**

- Since `KERNBASE` is equal to `0xf0000000`, the virtual address range mentioned above has a size of approximately 256MB. Hence, the physical range it maps to has to have a size of 256MB as well.
- If we have physical memory greater than 256MB, then there will be a region that is unmapped.
- Reverse mappings from certain physical addresses to virtual addresses would not exist. The unmapped region would not be usable.
- Thus, this operating system can support only upto 256MB.

5. **How much space overhead is there for managing memory, if we actually had the maximum amount of physical memory? How is this overhead broken down?**

- Memory management is handled by means of the Page Directory and the associated page tables.
- We have a single page directory, and this accounts for 4KB.
- Since the page directory can have a maximum of 1024 entries, if we had the maximum amount of physical memory, all these entries would be filled up.
- Hence, we would also be having 1024 page tables, each of size 4KB. Therefore the total space overhead is `4KB * 1024 + 4KB`, which is 4KB more than 4MB i.e., the space overhead would be a little over 4MB.

6. **Revisit the page table setup in `kern/entry.S` and `kern/entrypgdir.c`. Immediately after we turn on paging, `EIP` is still a low number (a little over 1MB). At what point do we transition to running at an `EIP` above `KERNBASE`? What makes it possible for us to continue executing at a low `EIP` between when we enable paging and when we begin running at an `EIP` above `KERNBASE`? Why is this transition necessary?**

- We transition to running at an EIP above KERNBASE once paging is enabled and just before entering the C code (`line 68` in `kern/entry.S`).
- A trivial page directory (defined in `/kern/entrypgdir.c`) is loaded into CR3 before paging is enabled. This code defines the initial PD and PT that are used in `kern/entry.S`
- This code maps both `[0, 4MB)` and `[KERNBASE, KERNBASE+4MB)` sets of VAs to the `[0, 4MB)` set of PAs.
- So after paging is enabled, the low virtual addresses are still being mapped to the correct low physical addresses.
- Hence, execution at a low `EIP` after enabling paging makes it possible to continue execution.
- This transition is necessary so that the kernel can be run at a high virtual address. It must run at a high virtual address because the lower address space is meant for user code in the x86 atchitecture, so by accessing it from a large virtual address, the lower addresses are free for user processes.