Operating Systems (0368-2162)

Multi-Level Page Tables Assignment

Due date (via moodle): **November 24th, 23:55**

Individual work policy

The work you submit in this course is required to be the result of your individual effort only. Feel free to discuss concepts and ideas with others, but **you should never observe, or have possession of, another student's code** (from this or previous semesters).

Students violating this policy will receive a 250 grade ("did not complete course duties") in the course. *If you're under pressure, contact the course staff or even don't submit, rather than cheat and risk having to repeat the course.*

1 Introduction

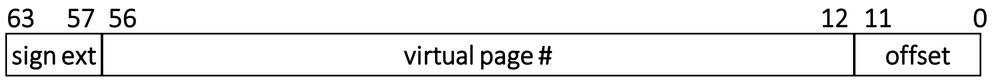
The goal in this assignment is to implement simulated OS code that handles a multi-level (trie-based) page table. You will implement two functions. The first function creates/destroys virtual memory mappings in a page table. The second function checks if an address is mapped in a page table. (The second function is needed if the OS wants to figure out which physical address a process virtual address maps to.)

Your code will be a simulation because it will run in a normal process. We provide two files, `os.c` and `os.h`, which contain helper functions that simulate some OS functionality that your code will need to call. For your convenience, there's also a `main()` function demonstrating usage of the code. However, the provided `main()` only exercises your code in trivial ways. We recommend that you change it to thoroughly test the functions that you implemented.

1.1 Target hardware

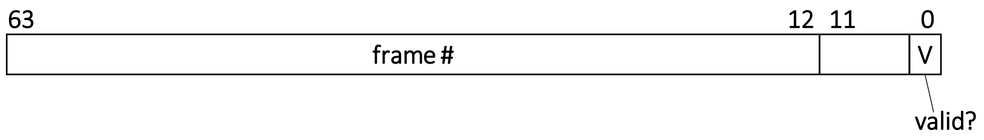
Our simulated OS targets an imaginary 64-bit x86-like CPU. When talking about addresses (virtual or physical), we refer to the least significant bit as bit 0 and to the most significant bit as bit 63.

Virtual addresses The virtual address size of our hardware is 64 bits, of which only the lower **57** bits are used for translation. The top 7 bits are guaranteed to be identical to bit 56, i.e., they are either all ones or all zeroes. The following depicts the virtual address layout:



Physical addresses The physical address size of our hardware is also 64 bits.

Page table structure The page/frame size is 4 KB (4096 bytes). Page table nodes occupy a physical page frame, i.e., they are 4 KB in size. The size of a page table entry is 64 bits. Bit 0 is the valid bit. Bits 1–11 are unused and must be set to zero. (This means that our target CPU does not implement page access rights.) The top 52 bits contain the page frame number that this entry points to. The following depicts the PTE format:



Number of page table levels To successfully complete the assignment, you must answer to yourself: how many levels are there in our target machine's multi-level page table? As mentioned, assume that only the lowest 57 bits of the virtual address are used for translation.

1.2 OS physical memory manager

To write code that manipulates page tables, you need to be able to perform the following: (1) obtain the page number of an unused physical page, which marks it as used; and (2) obtain the kernel virtual address of a given physical address. The provided `os.c` contains functions simulating this functionality:

1. Use the following function to allocate a physical page (also called *page frame*):

   ```
   uint64_t alloc_page_frame(void);
   ```

   This function returns the **physical page number** of the allocated page. In this assignment, you do not need to free physical pages. If `alloc_page_frame()` is unable to allocate a physical page, it will exit the program. The content of the allocated page frame is all zeroes.

2. Use the following function to obtain a pointer (i.e., virtual address) to a **physical address**:

   ```
   void* phys_to_virt(uint64_t phys_addr);
   ```

   The valid inputs to `phys_to_virt()` are addresses that reside in physical pages that were previously returned by `alloc_page_frame()`. If it is called with an invalid input, it returns NULL.

2 Assignment description

Implement the following two functions in a file named `pt.c`. This file should `#include "os.h"` to obtain the function prototypes.

1. A function to create/destroy virtual memory mappings in a page table:

   ```
   void page_table_update(uint64_t pt, uint64_t vpn, uint64_t ppn);
   ```

   This function takes the following arguments:

   (a) `pt`: The **physical page number** of the page table root (this is the physical page that the page table base register in the CPU state will point to). You can assume that `pt` has been previously returned by `alloc_page_frame()`.

(b) `vpn`: The **virtual page number** the caller wishes to map/unmap.

(c) `ppn`: Can be one of two cases. If `ppn` is equal to a special `NO_MAPPING` value (defined in `os.h`), then `vpn`'s mapping (if it exists) should be destroyed. Otherwise, `ppn` specifies the **physical page number** that `vpn` should be mapped to.

2. A function to query the mapping of a **virtual page number** in a page table:

```
uint64_t page_table_query(uint64_t pt, uint64_t vpn);
```

This function returns the **physical page number** that `vpn` is mapped to, or `NO_MAPPING` if no mapping exists. The meaning of the `pt` argument is the same as with `page_table_update()`.

You can implement helper functions for your code, but make sake sure to implement them in your `pt.c` file. You may not submit additional files (not even header files).

IMPORTANT: A page table node should be treated as an array of `uint64_t`s.

2.1 Something to think about (no need to submit)

How would your code change if you were required to free a page table node once all of the PTEs it contains become invalid? How would you detect this condition? How efficient would your approach be (i.e., how much overhead would it add on every page table update)?

3 Submission instructions

1. Submit just your `pt.c` file. (We will test it with our own `main` function.)

2. The program must compile cleanly (no errors or warnings) when the following command is run in a directory containing the source code files:

```
gcc -O3 -Wall -std=c11 os.c pt.c
```

4 Grading tips (will be removed from assignment)

The idea is to check this assignment by linking against the submitted code and calling the functions directly. Here are potential pitfalls worth checking for.

Checking `page_table_update()` To check this function, we perform updates of various page tables and check that the submitted code creates the page table correctly.

1. Basic sanity check that the page tables have a correct format. In particular, (1) the number of levels is 5 and (2) the PTEs contains physical page numbers. Usage of `malloc` in the student's code is a big hint that it's incorrect.

2. Correct handling of physical page number 0 (that they don't assume it's NULL).

3. Changing a mapping (i.e., overwriting of a previous mapping) works.

4. If address v is mapped and we create a mapping for address v' that shares a prefix with v, then the high-level PTEs shouldn't get overwritten.

5. Destroying a mapping (`NO_MAPPING` argument) marks the relevant leaf PTE as invalid. A possible wrong implementation will set the physical page # to `NO_MAPPING` but keep the PTE valid.
6. Correct usage of a page numbers vs. addresses. In particular, (1) some might not use `phys_to_virt()` and (2) some might mistakenly pass a page number to `phys_to_virt()` (this won't crash, but will corrupt the page table).
7. That they don't assume the root node is on physical page 0, just because that's what the supplied `main()` does.
8. **Don't check** passing of invalid VPNs whose MSBs aren't sign extension of bit 56.
9. **Don't check** unmapping of a virtual page that is already unmapped.
10. **Don't check** passing a root page that wasn't previously returned by `alloc_page_frame()`.

Checking `page_table_query()` To check this function, we build the page tables and check that the submitted code traverses them correctly. (It's important to use our page tables, in case the student made the same mistake in both building and traversing the page table. Using our page table will catch such bugs.)

1. Basic sanity check that they return correct responses (for example, if they don't parse PTEs correctly, nothing will work), including handling of valid/invalid entries. In particular, that they don't treat the physical page # field in the PTE as a virtual address or as a physical address. Also, this function should never update the page table.
2. Querying an unmapped address whose translations ends (i.e., has a PTE marked invalid) at level i, for $i = 0, 1, 2, 3, 4$.
3. That they don't assume the root node is on physical page 0, just because that's what the supplied `main()` does.
4. That the code handles PTEs whose physical page number field is $> 2^{20}$ (which is what the provided code limits PPNs to).
5. **Don't check** passing of invalid VPNs whose MSBs aren't sign extension of bit 56.
6. **Don't check** passing a root page that wasn't previously returned by `alloc_page_frame()`.