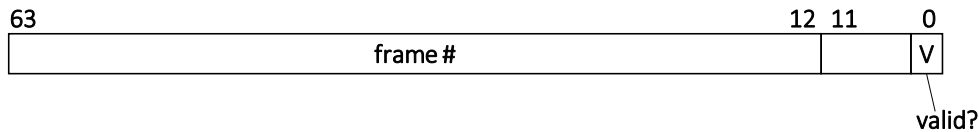


Page table structure The page/frame size is 4 KB (4096 bytes). Page table nodes occupy a physical page frame, i.e., they are 4 KB in size. The size of a page table entry is 64 bits. Bit 0 is the valid bit. Bits 1–11 are unused and must be set to zero. (This means that our target CPU does not implement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

- (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 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 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_ts**.

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
```