Assignment 2

Memory Management

AUTHOR
Operating Systems Teaching and Research Unit

PUBLISHED
09.05.2025

C Reminder: Fixed-width integer types

In C programming, the size of types like `int` or `long` can vary depending on the system architecture. However, C also provides fixed-width integer types that ensure consistent sizes across different platforms.

To ensure your variables have a precise size, you can use these fixed-width types. This is particularly important in scenarios like memory management on a 64-bit system.

For this assignment, we need to handle 64-bit addresses, so we'll use the `uint64_t` type. This type represents a 64-bit unsigned integer, making it ideal for storing addresses accurately and consistently.

C Reminder: Type conversion

For this tutorial, you will need to cast some values to pointers for the page walk.

While going through the different page tables, you will have to interpret some values as pointers.

Let's take this example: `value` is of type `uint64_t`.

```
uint64_t value = page_table_pointer[index];
```

If you think that this value contains an address to another memory location, you can interpret the value as a pointer:

```
uint64_t *new_pointer = (uint64_t *) value;
```

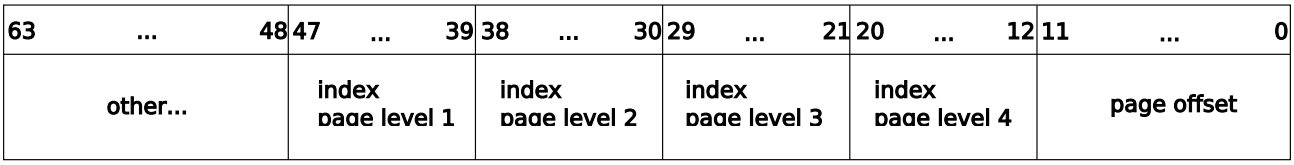
Casting in C is not limited to pointers; it can be used to change the type of any basic type. It can be used to convert chars to int, double to int etc…

See the wikipedia page on type conversion for more information.

Assignment

This week's assignment will focus on implementing the page walk for virtual to physical address translation & eviction policies.

First, we will be simulating a 64-bit system, with a page size of 4 KiB. The virtual address is 64 bits, and there are 4 levels of page table. The index for each page table is decribed in the following figure:



Virtual Address Format

We have in this architecture:

- 12 bits for the page offset (because a page is 2^12 bits).
- 9 bits for each of the page table
- 16 bits reserved that are not used for address translation

The Page Table Entry is 12 bits of flags, 40 bits of page frame and finally 12 reserved bits by the CPU.

The useful flags on the PTE are the following:

```
enum flags {
  PRESENT,        // 1 if page frame present in memory (bit 0 of PTE)
  READWRITE,      // 1 if read/write, 0 if readonly (bit 1 of PTE)
  USERSUPERVISOR, // 1 if accessible in user/supervisor mode, 0 if only
                  // supervisor mode (bit 2 of PTE)
};
```

Page Table Entry

Task 1: Bitwise manipulation

Start by implementing the following functions. They will be useful for checking the permissions and the present bit of the page.

- `set_bit(uint64_t value, int bit)`: set the nth `bit` to 1
- `test_bit(uint64_t value, int bit)`: test the nth `bit`
- `clear_bit(uint64_t value, int bit)`: set the nth `bit` back to 0

Testing the bits of the PTE will be as simple as:

```
uint64_t *pte = ...;
if (!test_bit(*pte, PRESENT)) {
    // handle the fault
}
```

Make sure that your implementation is correct by evaluating your code before proceeding to the next task.

Task 2: Break down the virtual address

> **Tip**
>
> In this task, you'll need to use bitmasks to isolate the relevant bits from the virtual address.
>
> You will need to use the `&` operator to extract the desired bits and the shift `>>` operator to move the bits to the right.
>
> For example, if you want to isolate the bits from 6 to 11, you would use the following code
>
> ```
> (address & 0b00011111100000) >> 5
> ```

Index of the page tables

Implement the following functions:

```
uint64_t index_page_level_1(uint64_t virtual_address);
uint64_t index_page_level_2(uint64_t virtual_address);
uint64_t index_page_level_3(uint64_t virtual_address);
uint64_t index_page_level_4(uint64_t virtual_address);
```

1. return the index for the first level page table
2. return the index for the second level page table
3. return the index for the third level page table
4. return the index for the last level page table (it should give you the PTE)

Offset within the page

How would you get the offset from the virtual address?

```
uint64_t offset(uint64_t virtual_address);
```

Task 3: PTE to Physical Address

Given a PTE and a Virtual Address, return the associated Physical Address

```
uint64_t pte_to_physical(uint64_t pte, uint64_t virtual_address)
```

Task 4: Eviction Policies

In this task, you will implement two eviction policies: FIFO and CLOCK. The skeleton codes are available in Moodle, and you can follow the comments on the files for your implementation.

Your designs include three main functions for each policy (**check VPL files for exact inputs and return types**):

```
init_list();
/*
Allocating memory for struct memory,
properly initializing the values,
and returning the proper pointer.
*/
```

```
map_page();
/*
Mapping a virtual address to a page frame.
*/
```

```
access_page();
/*
Handling the access to an address. Note that the
address might not present, so you need to do a mapping first.
*/
```

The header files included in each policy (mmem_clock.h & mmem_fifo.h) are also available at Moodle, however, you don’t need to change them.

```
/*
mmem_fifo.h
*/
#ifndef MMEM_FIFO_H__
#define MMEM_FIFO_H__

#include "mmem.h"

/**
 * \brief Initializes the list with the page frames
 */
memory* stud_fifo_init_list();


/**
 * \brief maps a page to a page frame
 */
void stud_fifo_map_page(memory* mem, uint64_t virtual_address);


/**
 * \brief function is called, when a page is accessed
 * The page that is accessed may or may not be in a page frame
 */
void stud_fifo_access_page(memory* mem, uint64_t virtual_address);

#endif
```

```
/*
mmem_clock.h
*/
#ifndef MMEM_CLOCK_H__
#define MMEM_CLOCK_H__

#include "mmem.h"

/**
 * \brief Initializes the list with the page frames
 */
memory* stud_clock_init_list();


/**
 * \brief maps a page to a page frame
 */
void stud_clock_map_page(memory* mem, uint64_t virtual_address);


/**
 * \brief function is called, when a page is accessed
 *  The page that is accessed may or may not be in a page frame
 */
void stud_clock_access_page(memory* mem, uint64_t virtual_address);



#endif
```

- Note that the memory struct manages all page frames of a given system. There is no predetermined way for how to manage the array of page frames and the concentration is on loading and evicting pages according to the policy.
- You don’t need to implement the body of evict_page() and load_page() for your submissions. Only call them when it is needed in your code. However, if you want to locally test your codes, you need to implement them.

Notes:

1- In case a part of the tasks is not clear, please use Moodle to ask your questions.

2- Some of the functions are defined for testing and evaluation.

3- You should not modify some parts of the skeleton code. Please read the comments carefully. If some parts of the comments were unclear, please contact us via Moodle.