CMPE 260 - Principles of Programming Languages
Spring 2024 - Project 2

Deadline: 28 May 2024 23:59

Assistants: Deniz Baran Aksoy
Ramazan Onur Acar, Mert Cengiz, Hasan Kerem Şeker
`deniz.aksoy@bogazici.edu.tr`

1 Introduction

In this project, you are expected to implement a simple Memory Management Unit (MMU). The Central Processing Unit (CPU) is primarily responsible for executing the instructions in a computer system. However, the CPU does not store either the instructions or the data. The instructions and the data are stored in the main memory, and the CPU fetches them from the main memory as needed. For example, a basic instruction execution cycle consists of the following steps: CPU fetches the instruction from the main memory, decodes the instruction, fetches the data from the main memory, executes the instruction, and writes the result back to the main memory. In order to communicate with the main memory, the CPU generates memory addresses and sends them to the main memory. Thus, the memory unit sees only a stream of memory addresses; it doesn't know how they are generated or their purpose. However, these memory addresses are logical(virtual) addresses, and they need to be translated into physical addresses before they can be used to access the main memory. MMU converts the logical addresses to physical addresses.[1]

2 Memory Management Unit

MMU is a hardware unit that sits between the CPU and the main memory. It translates the logical (virtual) addresses generated by the CPU to physical addresses.

An address generated by the CPU is known as a logical address, while an address seen by the memory unit is defined as a physical address. The Memory Management Unit (MMU) is responsible for mapping logical addresses to physical addresses. One mechanism for this operation involves the use of registers. The MMU utilizes a relocation register, which holds the smallest physical address, and a limit register, which defines the range of logical addresses. This configuration ensures that each logical address falls within a predefined range and makes accurate and secure address mapping. For example, if the logical address is 256, the value of the limit register is 350 and the value of the relocation register is 1200, then the logical address is mapped to 1456 (256 + 1200).

Another mechanism (and the most commonly used one) is paging. While the above method requires the physical address space of a process to be contiguous, paging allows a process's address space to be noncontiguous by dividing it into fixed-size blocks called pages. This approach enables

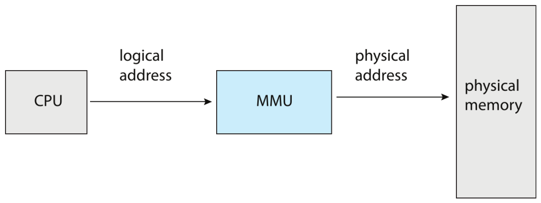


Figure 1: Memory Management Unit (MMU)
[1]

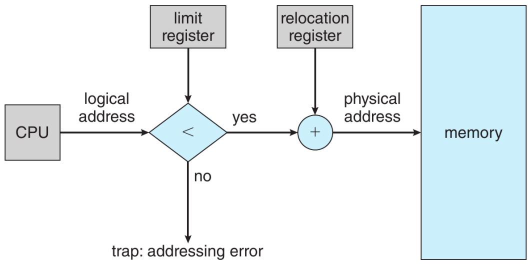


Figure 2: Relocation and Limit Registers
[1]

each page of the process's virtual address space to be independently mapped to any available frame in physical memory. In order to implement paging, every address generated by the CPU is divided into two parts, a page number (p) and page offset (d).

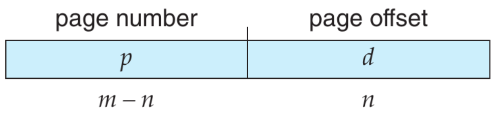


Figure 3: Division of Logical Address Space
[1]

The page size is determined by the hardware and is a power of 2 (generally between 4KB and 1GB). If the size of the logical address space is 2^m and the page size is 2^n bytes, then $m - n$ bits of the logical address are used to access a specific page number and the remaining n bits of the logical address are used as page offset, pointing to a specific location within the page.

The page number is used by the MMU as an index to access the page table. The page table contains the base address of each frame in physical memory and the offset is the location within the frame that being referenced. For example, if the logical address space consists of 16 bits and

the page size is 4KB, then the $m = 16$ and $n = 12$. Then, the first $[m - n = 4]$ bits will be used as page number and the remaining $n(12)$ bits will be used as page offset. So, for this example there will be $2^4 or 16$ pages.
The MMU follows the steps listed below to translate the logical address space to a physical address.

1. Extract the page number p and use it as index into the page table
2. Extract the corresponding page number fro the page table
3. Replace the page number p in the logical address with the frame number f

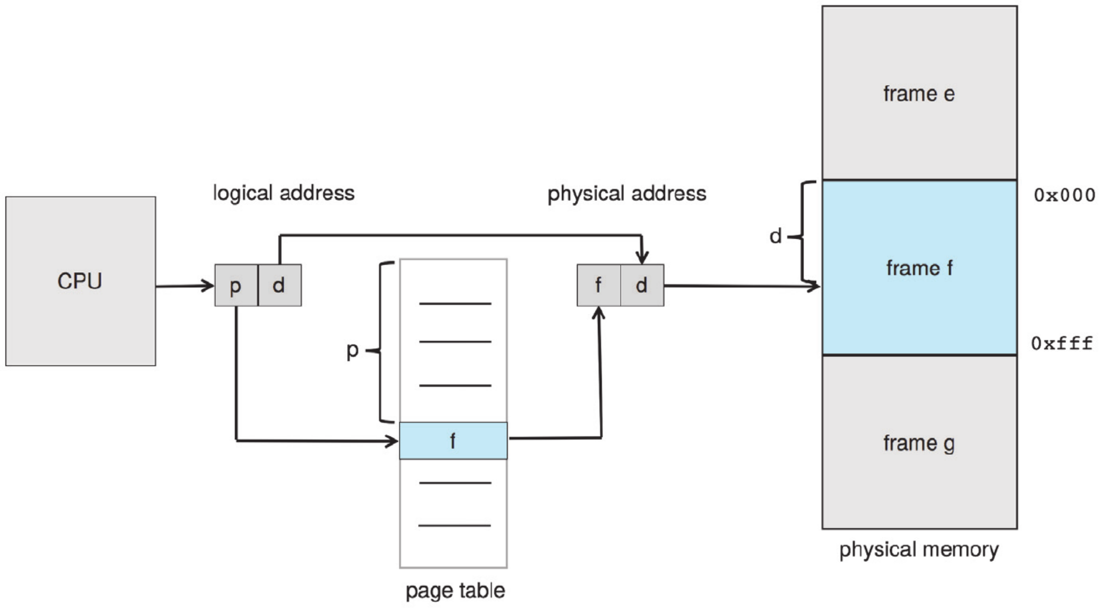


Figure 4: Paging Process
[1]

3 Procedures

In this project you are expected to implement nine procedures to simulate a Memory Management Unit (MMU).

3.1 (binary_to_decimal *binary*)

This procedure returns the decimal value of the binary number `binary` which is given as a string.

```
> (binary_to_decimal "100101")
37
```

```
> (binary_to_decimal "11111101000")
2024
```

3.2 (relocator *args limit base*)

This procedure returns the corresponding physical addresses for a list of logical addresses in binary format, given in the list `args`. The procedure should first convert each binary number to decimal. Then, if the value of the address exceeds the value of the `limit`, the procedure should return -1. If the value of the address is less than or equal to the `limit`, the procedure should make the necessary mapping.

Examples:

```
> (relocator '("000010100111" "010000110001" "100100111101" "100110010001" "101111011000")
3500 1200 )
'(1367 2273 3565 3649 4232 )
> (relocator '("0010010110001000" "1011111000100111" "0101010100000101" "0101011101001111")
25000 400 )
'(10008 -1 22165 22751)
```

3.3 (divide_address_space *num page_size*)

This procedure returns a list consisting of the page number and page offset by splitting a logical address according to the given *page_size* in KB ($1KB = 2^{10}B$).

Examples:

```
> (divide_address_space "11011011011000" 4)
'("11" "011011011000" )
> (divide_address_space "1111101010110000000000" 512 )
'( "111" "1101010110000000000" )
> (divide_address_space "10110111010010000011101110011011" 256 )
'( "10110111010010" "000011101110011011" )
```

3.4 (page *args page_table page_size*)

This procedure returns the list of physical addresses of the given `args` with using `page_table` and `page_size` in KB.

Examples:

```
> (page '("110010111011001" "000001111111010" "010001100000100" "101001011011101")
'( "100" "000" "010" "110" "011" "001" "111" "101") 4 )
'("111010111011001" "100001111111010" "010001100000100" "001001011011101")
```

```
> (page '("01101000101111110") '("11" "00" "10" "01") 32 )
'("00101000101111110")
```

3.5 (find_sin *value num*)

This procedure returns the sine of an angle given in `value` by using the Taylor series expansion, up to a predefined number `num`. For example, if the `num` is five, the procedure should use the first five terms.

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ... \tag{1}$$

Examples:

```
> (find_sin 45 5)
0.7071067829368671
> (find_sin 30 2)
0.49967417939436376
```

3.6 (myhash *arg table_size*)

This procedure returns the hash value of a given binary number. The procedure first finds the decimal value of the given `arg`. Then, it calculates the sin of this decimal number using the first `n` terms of the Taylor series, where `n` is equal to $(number \mod 5) + 1$. Finally, the procedure sums the first ten digits after the decimal point and takes the modulus of `table_size` of the result.

Examples:

```
> (myhash "1101" 8)
3
> (myhash "0110101" 12)
11
```

3.7 (hashed_page *arg table_size page_table page_size*)

This procedure returns the physical address for a given logical address `arg` by using a hashed page table. A hashed page table first computes the hash value of the page number. Then, it compares this page number with the heads of the lists at the corresponding hash table index. If there a match is found, the tail of the list(frame number) is concatenated with the page offset to generate physical address.

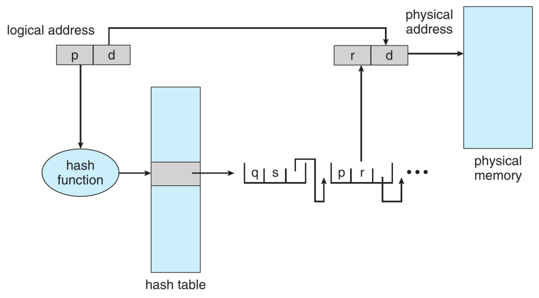
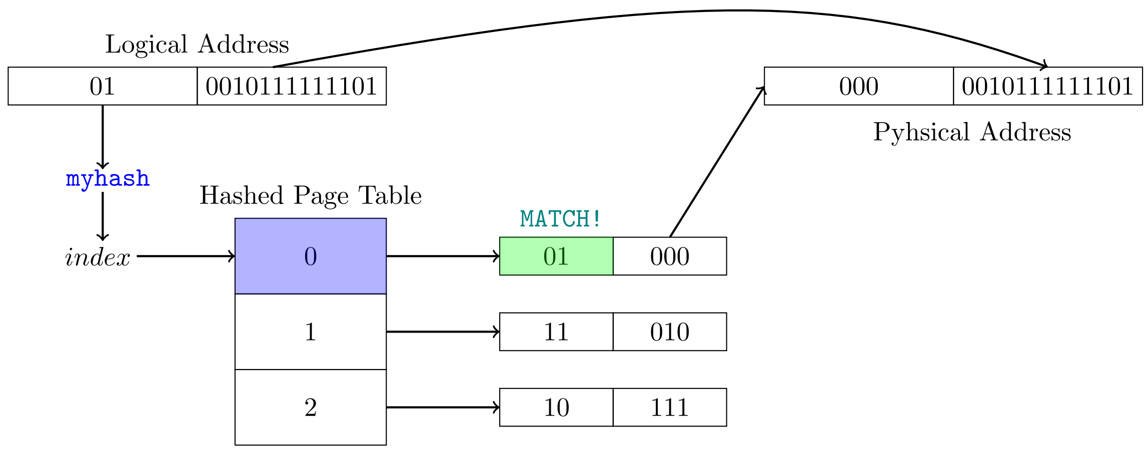


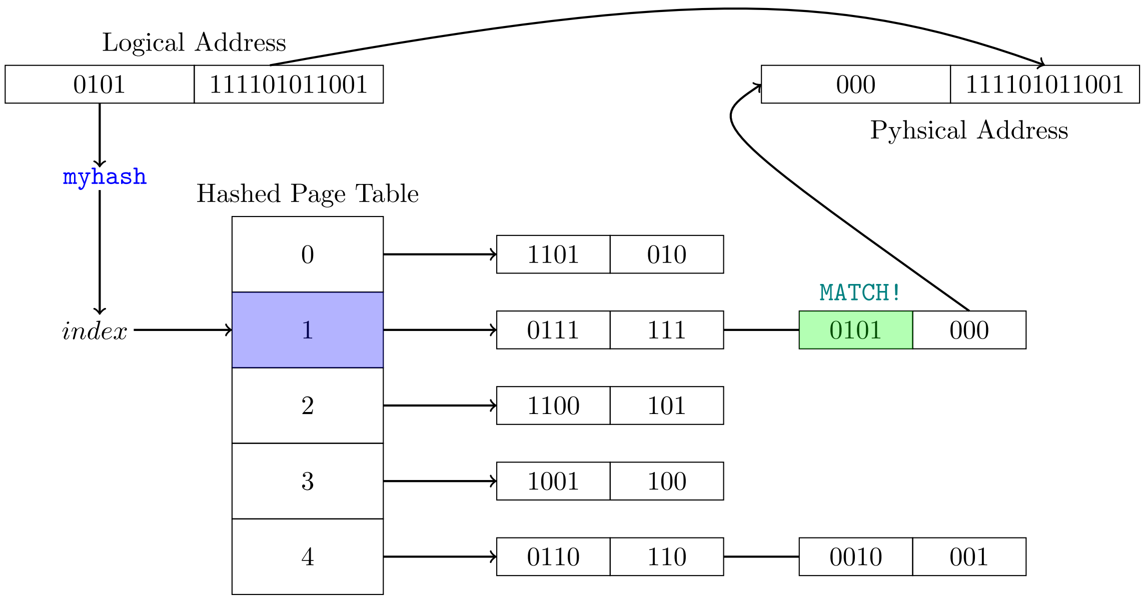
Figure 5: Hashed Page Table

Examples:

```
> (hashed_page "010010111111101" 3 '( ( ("01" "000") ) ( ("11" "010") ) ( ("10" "111")
) ) 8)
"0000010111111101"
```



```
> (hashed_page "0101111101011001" 5 '( ( ("1101" "010") ) ( ("0111" "111") ("0101" "000")
) ( ("1100" "101") ) ( ("1001" "100") ) ( ("0110" "110") ("0010" "001") ) ) 4)
"000111101011001"
```



3.8 (split_addresses *args size*)

Given a stream of logical addresses `args` and the size of the logical address space `size`, this procedure returns a list of logical addresses.

Examples:

```
> (split_addresses "1110110101000000100100101011000101110011" 8)
'(("11101101") ("01000000") ("10010010") ("10110001") ("01110011"))
> (split_addresses "10101110101111010010101011111101" 16)
'(("1010111010111101")("0010101011111101"))
> (split_addresses "011110001101" 4)
'(("0111")("1000")("1101"))
```

3.9 (map_addresses *args table_size page_table page_size address_space_size*)

This procedure returns a list of physical addresses for a given stream of logical addresses by using a hashed page table.

Examples:

```
> (map_addresses "001010000011001011000010100000011001011101001010" 5 '( ( ("1101" "010")
) ( ("0111" "111") ("0101" "000") ) ( ("1100" "101") ) ( ("1001" "100") ) ( ("0110"
"110") ("0010" "001") ) ) 4 16)
'("001100000110010" "101001010000001" "100011101001010")
```

4 Submission

You should submit your code in a single file named `main.rkt`. The first four lines of your `main.rkt` file must have exactly the lines below since it will be used for compiling and testing your code automatically:

```
; name surname
; student id
; compiling: yes
; complete: yes
#lang racket
```

The third line denotes whether your code compiles correctly, and the fourth line denotes whether you completed all of the project, which must be `no` if you are doing a partial submission. This whole part must be lower case and include only the English alphabet. Example:

```
; deniz baran aksoy
; 2021234567
; compiling: yes
; complete: yes
#lang racket
```

5 Prohibited Constructs

The following language constructs are ***explicitly prohibited***. You *will not get any points* if you use them:

- Any function or language element that ends with an `!`.
- Any of these constructs: `begin`, `begin0`, `when`, `unless`, `for`, `for*`, `do`, `set!-values`.
- Any language construct that starts with `for/` or `for*/`.

6 Tips and Tricks

- You can use higher-order functions `apply`, `map`, `foldl`, `foldr`. You are also encouraged to use anonymous functions with the help of `lambda`.
- You can use Racket reference, either from DrRacket's menu: Help, Racket Documentation, or from the following link `https://docs.racket-lang.org/reference/index.html`.
- A useful link for the difference between `let`, `let*`, `letrec`, and `define`: `https://stackoverflow.com/questions/53637079/when-to-use-define-and-when-to-use-let-in-racket`.

References

[1] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts.* Wiley Publishing, 9th edition, 2012.