Computing platforms (Spring 2025)
week 3

Exercise 1

Address Translation: Segmentation

32-bit address space, leftmost 4 bits represent segment number and remaining 28 bits offset within that segment. The segmentation table for a process is as follows:

Index	Base Address	Segment Length
0	0x00001000	0x0500
1	0x00020000	0x5000
2	0x00030000	0x10000
3	0x00045000	0x5000

(a) Translating Virtual Address `0x20001111`

The virtual address can be split into:

Segment Number	Offset
0x2 (Segment 2)	0x0001111

For segment 2:

Segment	Base Address	Segment Length
2	0x00030000	0x10000

Since `0x0001111` is within the segment length `0x10000`, the physical address is:

$$\text{Physical Address} = \texttt{Base Address} + \texttt{Offset} = \texttt{0x00030000} + \texttt{0x0001111} = \texttt{0x00031111}$$

Answer for (a): The virtual address `0x20001111` translates to the physical address `0x00031111`.

(b) Translating Virtual Address `0x00001000`

The virtual address can be split into:

Segment Number	**Offset**
`0x0` (Segment 0)	`0x0001000`

For segment 0:

Segment	**Base Address**	**Segment Length**
0	`0x00001000`	`0x0500`

Since the offset `0x1000` exceeds the segment length `0x0500`, segmentation fault occurs.

Answer for (b): The virtual address `0x00001000` is invalid because the offset exceeds the segment's size, causing segmentation fault.

Exercise 2

Address Translation: Paging

We have a system with:

- 32-bit virtual addresses
- 4 KB pages (4 KB = 2^{12} bytes)
- A single-level page table

(a) Layout of Virtual Addresses

Since each page is 2^{12} bytes in size, the **offset** occupies the lower 12 bits of the address. The remaining upper 20 bits specify the **page number**. Thus, a 32-bit virtual address is divided as follows:

$$\underbrace{\text{Page Number}}_{\text{20 bits}} \mid \underbrace{\text{Offset}}_{\text{12 bits}}$$

```
[ 5 hex digits ] [ 3 hex digits ]
```

(b) Translating Virtual Address `0x00001234`

Assume a process has single-level page table shown below:

Index	Frame Number
0	3
1	2
2	7
3	1

Step 1: Split the Virtual Address.

$$\texttt{0x00001234} \quad \longrightarrow \quad \begin{cases} \text{Page Number} = 1 \\ \text{Offset} = 0x234 \end{cases}$$

Step 2: Look Up the Frame. Using page number = 1 $\longrightarrow$ Frame Number = 2

Step 3: Construct the Physical Address. Each frame is also 4 KB = 2^{12} bytes hence frame number 2 starts at

$$\underbrace{2 \times 2^{12}}_{\text{Frame Number}\times\text{Frame Size}} = 2 \times 0x1000 = \texttt{0x2000}.$$

and add offset 0x234:

$$\text{Physical Address} = \texttt{0x00002000} + \texttt{0x00000234} = \texttt{0x00002234}.$$

Exercise 3

Virtual Memory with Paging and TLB

We have:

- 32-bit virtual addresses
- 4 KB pages (4 KB = 2^{12} bytes)
- A single-level page table and a Translation Lookaside Buffer (TLB)

Page Table and TLB Contents

single-level page table:

Page table				**TLB**				
Index	**Frame Number**	**M**	**P**	**Index**	**Page #**	**Frame #**	**M**	**P**
0	3	0	1	0	3	1	1	1
1	2	0	1	1	0	3	0	1
2	7	0	0	2				0
3	1	1	1	3				0

- `M = 1` indicates the cached page is modified.
- `P = 1` indicates the TLB entry is valid (present).
- Entries 2 and 3 are either empty or invalid (`P = 0`).

Address Format (32-bit, 4 KB Pages)

A 32-bit address is split into:

$$\underbrace{\text{Page Number}}_{\text{20 bits}} \mid \underbrace{\text{Offset}}_{\text{12 bits}}$$

Hence, the lower 12 bits (3 hex digits) are the *offset*, and the upper 20 bits are the *page number*.

(a) Translating Address `0x00001123`

Step 1: Split the Address.

`0x00001123`

- Offset = `0x123` (the lowest 12 bits)
- Page Number = `0x1` (the upper 20 bits)

Step 2: TLB Lookup. We check whether the TLB has an entry for Page 1.

- Index 0 has Page 3.
- Index 1 has Page 0.
- Index 2,3 are invalid.

No match $\Rightarrow$ **TLB miss**.

Step 3: Page Table Lookup. From the Page Table:

$$\text{Page 1} \longrightarrow \text{Frame 2},\ M = 0,\ P = 1$$

Since `P=1`, the page is in memory (no page fault).

Step 4: Construct Physical Address. Frame 2 means the base of this frame is:

$$2 \times 2^{12} = \texttt{0x2000}.$$

Adding the offset `0x123`:

$$\text{Physical Address} = \texttt{0x2000} + \texttt{0x0123} = \texttt{0x2123}.$$

Step 5: Update TLB. We add a new TLB entry for Page 1:

$$\begin{aligned} \text{Page No.} &= 1, \\ \text{Frame No.} &= 2, \\ M &= 0 \text{ (no writes so far)}, \\ P &= 1 \text{ (valid entry)}. \end{aligned}$$

This typically occupies an empty TLB slot (say Index 2).
The virtual address `0x00001123` translates to `0x00002123`. The TLB is updated to include Page 1 $\rightarrow$ Frame 2.

(b) Translating Address `0x00002333`

Step 1: Split the Address.

`0x00002333`

- Offset = `0x333` (lowest 12 bits)
- Page Number = `0x2` (upper 20 bits)

Step 2: TLB Lookup. We check for Page 2 in the TLB:

- Index 0 has Page 3,
- Index 1 has Page 0,
- Index 2 (updated above) has Page 1,
- Index 3 is invalid/empty.

No match $\Rightarrow$ **we got TLB miss again.**

Step 3: Page Table Lookup. From the Page Table:

$$\text{Page } 2 \longrightarrow \text{Frame } 7,\ M = 0,\ P = 0$$

`P=0` means the page is not present in main memory $\Rightarrow$ **page fault**.

Step 4: Handle Page Fault. The OS must load Page 2 from disk into a free frame. Assume the OS chooses some free frame `F`. After loading:

$$\begin{aligned}&\text{Page 2 now mapped to Frame } \texttt{F},\\ &M = 0 \text{ (still no writes)}, \quad P = 1 \text{ (now present)}.\end{aligned}$$

The page table is updated accordingly:

$$\text{Page Table}[2] \leftarrow (\texttt{Frame F}, M = 0, P = 1).$$

Step 5: Construct Physical Address. Once the page is loaded, the physical base address is:

$$\texttt{F} \times 2^{12},$$

and we add the offset `0x333` to get the final physical address.

Step 6: Update TLB. We add a valid entry for Page 2 in the TLB:

$$\text{Page } 2 \longrightarrow \text{Frame F},\ M = 0,\ P = 1.$$

The virtual address `0x00002333` causes a page fault. After the OS loads Page 2 into some free frame `F`, the physical address is `(F * 0x1000) + 0x333`. The page table and TLB entries for Page 2 are updated to reflect `Frame = F` and `P=1`.

Exercise 4

Page Replacement Comparison

We have:

- Main memory of 3 frames
- Reference stream: `7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2`
- We count page faults only after the first 3 references have initialized the frames.

Ref.	**7**	**0**	**1**	**2**	**0**	**3**	**0**	**4**	**2**	**3**	**0**	**3**	**2**
FIFO	7 - -	7 0 -	7 0 1	2 0 1 **F**	2 0 1	2 3 1 **F**	2 3 0 **F**	4 3 0 **F**	4 2 0 **F**	4 2 3 **F**	0 2 3 **F**	0 2 3	0 2 3
LRU	7 - -	7 0 -	7 0 1	2 0 1 **F**	2 0 1	2 0 3 **F**	2 0 3	4 0 3 **F**	4 0 2 **F**	4 3 2 **F**	0 3 2 **F**	0 3 2	0 3 2
Clock	7 - -	7 0 -	7 0 1	2 0 1 **F**	2 0 1	2 0 3 **F**	2 0 3	4 0 3 **F**	4 2 3 **F**	4 2 3	4 2 0 **F**	4 2 3	4 2 3
OPT	7 - -	7 0 -	7 0 1	7 0 2 **F**	7 0 2	7 3 2 **F**	7 3 0	2 3 0 **F**	2 3 0	2 3 0	2 3 4 **F**	2 3 4	2 3 4

Table 1: Comparison of FIFO, LRU, OPT, and Clock. "F" marks a fault.

Fault Counts and Miss Rates

- **FIFO:** 7 faults over 10 references ⇒ 70% miss rate.
- **LRU:** 6 faults ⇒ 60% miss rate.
- **Clock:** 4 or 5, depending on the initial hand and use-bit detail.
- **Optimal:** 5 faults if you consider extended reference stream $(\ldots, 1, 2, 0, 1, 7, 0, 1)$ to decide which page is used farthest in the future.

Exercise 5

Consider “matrix-rowwise.py” and “matrix-columnwise.py”

(a) Memory Organisation of the Two-Dimensional Table

In Python “list of lists” is stored as a top-level list, each element of which is a separate Python list object. Therefore, only the references in row `j` are contiguous within that row’s list structure; there is no guarantee that row `j+1` is adjacent to row `j` in memory. Each integer is also a Python object on the heap, so even within a single row, the values themselves need not be stored contiguously as raw bytes.

(b) Run both of the codes and measure how much time their execution takes

```
$ time python3 ./matrix-columnwise.py
171475500000

real    0m6.811s
user    0m6.607s
sys     0m0.199s
```

vs.

```
$ time python3 ./matrix-rowwise.py
171475500000

real    0m3.967s
user    0m3.778s
sys     0m0.185s
```

Columnwise reading in this case is slower at reading the memory.

(c) Explain the difference in the running times.

Row-wise traversal (`array[j][i]`) accesses elements from same inner list meaning it follows consecutive memory addresses for that row’s pointers, improving cache locality. **Column-wise traversal** (`array[j][i]`) accesses elements from different lists (one list per row), meaning each lookup requires fetching a new row’s list reference, which may be far apart in memory. This leads to more cache misses.

Comparison to NumPy Arrays If the same operation were done using a **NumPy array**, where elements are stored in a **contiguous block of memory**, cache strides would be effective, and column-wise traversal would be much faster. This demonstrates why NumPy is preferred for numerical computing in Python.

```
import numpy as np

cols = 7000
rows = 7000
array = np.arange(cols, dtype=np.int64).repeat(rows).reshape(rows, cols)

# axis 0 or 1 control column/row reading
summa = sum(np.sum(array, axis=0))

print(summa)
---
$ time python3 ./numpy-arr.py
171475500000

real    0m0.129s
user    0m0.075s
sys     0m0.049s
```