ECS518U Operating Systems

Lab 7: Multi-Level Paging, Page Faults & Memory Allocation

This exercise is assessed (Must be assessed the latest by your Week 11 lab but hopefully you will get it done before ☺).

Aim

This lab has three parts:

1. The aim of the first part of the lab exercise is to compile the C code `virtual.c` that outputs virtual addresses, run the non-interactive program, understand what is being outputted and what it entails.
2. The aim of the second part is to compile the C code `stack.c` that calls a recursive function, run the program and use a disassembler to analyse the machine code to understand what is being outputted by the program (and why).
3. The aim of the third part is to compile the C code `paging.c` that simulates x86-64 paging using 4-level page tables, run the interactive program and analyse how page allocation affects the overhead of the simulated page table tree and how the simulation's memory allocations on the heap affect the number of page faults.

You can assess this lab **only one time**, either during your Week 9, Week 10 or Week 11 lab slot. See QMPlus for details on assessment rules and marking criteria.

You should be able to:

1. Run shell commands, particularly:
    a. Compile C code with `GCC` (e.g., ''`gcc -o code code.c`'')
    b. Execute machine code on Ubuntu/MacOSX machine (e.g., ''`./code`'')
    c. Read machine code with disassembler (e.g., ''`objdump -d code`'')
    d. Monitor memory-related statistics of processes (e.g., using `ps` or `top`)
2. Understand some basics of C code, particularly useful to understand are:
    a. pointer-related operators such as `&x` (address), `*x` (dereference) and `x[i]`
    b. pointer types `void*` (generic) and `long*` (signed 64-bit integer pointer)
    c. bit-wise operators `x << y` (left shift) and `x & y` (bit-wise AND)
3. Compile/execute/use the interactive simulator from `paging.c` to simulate memory usage for a set of allocated pages/frames.

Part A: Virtual Addresses

The code of `virtual.c` outputs the virtual address (of the first byte) of various variables as hexadecimal (most commonly used for addresses) and decimal notation. The code can be compiled by running from a Unix/Linux shell by going to the folder with the code and shell command:

```
gcc -o virtual virtual.c
```

The code can then be executed by the following shell command:

```
./virtual
```

The task is to analyse the C code to:

1. **understand the outputs**
2. **reason about virtual vs physical address spaces**
3. **answer the questions Q1.1, Q1.2, Q1.3 in the answer sheet**

The output will be dependent on the machine, but just to give an example it could look something like the following:

```
michaelshekelyan@Supports-MacBook-Pro lab7 % gcc -o virtual virtual.c
michaelshekelyan@Supports-MacBook-Pro lab7 % ./virtual

this system uses 14 offset bits in each virtual address
that means the page size is 2^14 = 16384 bytes

---------

virtual address pointing to 1st byte of global offsetBits variable:
hexadecimal: 0x102d00000
decimal: 265024 * 16384 + 0 = 4342153216

virtual address pointing to 1st byte of printAddress function:
hexadecimal: 0x102cfb8d4
decimal: 265022 * 16384 + 14548 = 4342134996

virtual address pointing to 1st byte of array on stack:
hexadecimal: 0x16d107640
decimal: 373825 * 16384 + 13888 = 6124762688

---------

virtual address pointing to 1st byte of arrayOnHeap[0]:
hexadecimal: 0x600000259100
decimal: 6442451094 * 16384 + 4352 = 105553118728448

virtual address pointing to 1st byte of arrayOnHeap[1]:
hexadecimal: 0x600000259108
decimal: 6442451094 * 16384 + 4360 = 105553118728456

virtual address pointing to 1st byte of arrayOnHeap[2]:
hexadecimal: 0x600000259110
decimal: 6442451094 * 16384 + 4368 = 105553118728464

virtual address pointing to 1st byte of arrayOnHeap[3]:
hexadecimal: 0x600000259118
decimal: 6442451094 * 16384 + 4376 = 105553118728472
```

Part B: Stack

The code of `stack.c` calls a recursive function. Use a disassembler (suggestions for tools can be found in the code; e.g., `objdump`) to inspect the executable file. The code can be compiled by running from a Unix/Linux shell by going to the folder with the code and one of the following shell commands:

```
gcc -O0 -o stack0 stack.c
gcc -O3 -o stack3 stack.c
```

Note: *The flag* "`-O0`" *is the default and leads to less optimised machine code and* "`-O3`" *to more optimised machine code (may run noticeably slower for larger code bases).*

The machine code can then be executed by the following shell commands:

```
./stack0
./stack3
```

The code can then be disassembled by the following shell commands:

```
objdump -d stack0
objdump -d stack3
```

MacOSX only (requiring XCode command line tools):

```
otool -vt stack0
otool -vt stack3
```

The task is:

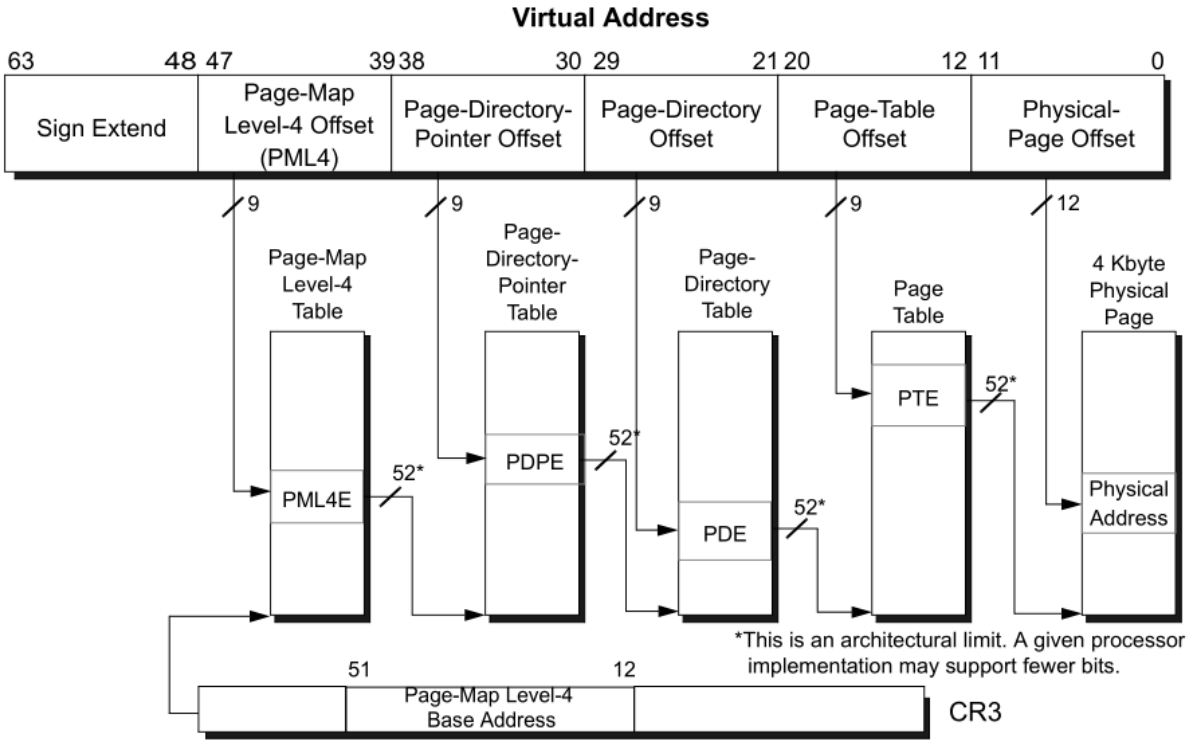
- **use a "disassembler" to understand how function calls work in machine code**
- **analyse the outputs and link it to the observations in the machine code**
- **answer the questions Q2.1, Q2.2 & Q2.3 in the answer sheet.**
- The output could look something like this:

```
michaelshekelyan@supports-mbp lab7 % gcc -o stack3 stack.c;./stack3

stack pointer moves by 48 bytes from 1 recursive calls
stack pointer moves by 96 bytes from 2 recursive calls
stack pointer moves by 144 bytes from 3 recursive calls
stack pointer moves by 192 bytes from 4 recursive calls
stack pointer moves by 240 bytes from 5 recursive calls
stack pointer moves by 288 bytes from 6 recursive calls
stack pointer moves by 336 bytes from 7 recursive calls
stack pointer moves by 384 bytes from 8 recursive calls
stack pointer moves by 432 bytes from 9 recursive calls
stack pointer moves by 480 bytes from 10 recursive calls
...
stack pointer moves by 4800 bytes from 100 recursive calls
...
stack pointer moves by 48000 bytes from 1000 recursive calls
...
stack pointer moves by 480000 bytes from 10000 recursive calls
michaelshekelyan@supports-mbp lab7 % objdump -d stack3

stack3: file format mach-o arm64

Disassembly of section __TEXT,__text:

0000000100003dd8 <_getStackPointer>:
100003dd8: fd 7b bf a9  stp     x29, x30, [sp, #-16]!
100003ddc: fd 03 00 91  mov     x29, sp
100003de0: e0 03 1d aa  mov     x0, x29
100003de4: fd 7b c1 a8  ldp     x29, x30, [sp], #16
100003de8: c0 03 5f d6  ret

0000000100003dec <_f>:
```

Part C: Paging using 4-Level Page Tables and Page Faults

The program has an interactive shell that asks a user input for a page number, or range of page numbers, and then allocates each of those page numbers with a “frame” using the above scheme. After each user input it outputs how much memory is used for the page tables (overhead) and “frames”.

The code of `paging.c` simulates the following 4-level page table scheme (AMD; Intel is similar except that it has 40-bit instead of 52-bit addresses in the entries) that has 48-bit virtual addresses (36-bit page number and 12-bit offset; the page number bits are evenly split between the 4 levels into per-level indices) which is very common on x86-64 bit architectures:



For sake of simplicity, we deviate from the above naming scheme (AMD) as follows:

Intel	AMD	Name throughout Lab
Page-Map Level-4 Table (PML4)		Level-1 Page Table
Page-Directory-Pointers Table (PDPT)	Page-Directory-Pointers (PDP)	Level-2 Page Table
Page-Directory (PD)		Level-3 Page Table
Page-Table (PT)		Level-4 Page Table

Note: *The program runs as a user process and therefore cannot translate virtual to physical addresses directly and instead simulates it by translating page numbers to virtual addresses of "frames" that are allocated using a system call (malloc). It is only meant to simulate the memory usage involved (which does not require direct knowledge of the physical addresses).*

The code can be compiled by running from a Unix/Linux shell by going to the folder with the code and shell command:

```
gcc -o paging paging.c
```

The code can then be executed by the following shell command:

```
./paging
```

The task is to:

1. **monitor number of page faults after allocating more and more page numbers**
2. **monitor how much overhead is caused by the 4-level page table scheme**
3. **answer the questions Q3.1, Q3.2, Q3.3 in the answer sheet.**

Exemplary interaction with paging simulator:

```
[General] Enter a number (e.g., 123) or a range (e.g., 1:1000) to simulate allo
 to display this again and 'exit' to quit.


> 1
[Simulator] allocating page 1 ... done.

[Virtual Memory]
        4-level page tables: 16 kilobytes
        1 frames: 4 kilobytes

> 2:1000
[Simulator] allocating pages between 2 and 1000 ... done.

[Virtual Memory]
        4-level page tables: 20 kilobytes
        1000 frames: 4000 kilobytes

> exit
michaelshekelyan@Supports-MacBook-Pro lab7 %
```

Testing, Demonstration and Assessment

You are expected to:

- Answer the questions in the **Answer Sheet** and be ready to explain your answers.
- Be ready to show, compile and execute the code and interpret its outputs (that can vary between executions).
- Be ready to link your answers to material in the lectures and how the provided code aims to clarify them.

Answer Sheet
PART A

Virtual Addresses

Answer the following questions.

Q & A	`virtual.c`
Basic Question **Q1.1**	The C program `virtual.c` outputs virtual addresses in decimal notation written as `A*B+C = D`. After inspecting the code, what does each of the variables `A`,`B`,`C` and `D` describe (e.g., what would be a good name for each of them)?
Your answer for **Q1.1**	
Intermediate Question **Q1.2**	The C program `virtual.c` outputs virtual addresses both in decimal and hexadecimal notation. One could also manually translate between both (e.g., 0xEFF1 = $14*16^3 + 15*16^2 + 15*16^1 + 1*16^0$). Pick any printed virtual address (copy the related outputs): how would one manually translate the hexadecimal value into a decimal one?

Your answer for **Q1.2**	
Bonus Question **Q1.3**	`virtual.c` outputs multiple virtual addresses. Going through all printed virtual addresses, what are the lowest and highest virtual addresses? (copy printed lines of those) Considering that virtual addresses point to a single byte, how many bytes are addressable based on the smallest and highest address? (does the system have that much RAM?) How many frames do contain some bytes of arrayOnHeap and presuming the same frame size what is the maximal number of frames that could theoretically contain some bytes of arrayOnHeap?

Your answer for **Q1.3**	

PART B

Call stack

Answer the following questions about the behaviour of `stack.c`.

Q & A	`stack.c`
Basic Question **Q2.1**	A stack pointer (SP) is the virtual address of the first byte of the stack. How many bytes does the SP move down per each recursive call? Does this number change when compiling either with the "-O3" or "-O0" flags?
Your answer for **Q2.1**	
Intermediate Question **Q2.2**	Use a disassembler tool (e.g., Ubuntu/Mac: objdump or Mac: otool) to analyse the compiled machine code of the program. In which instruction (should correspond to a disassembler output line) of the machine code does it jump/branch from the main function into the f function? (copy that line in your answer)

Your answer for **Q2.2**	
Bonus Question **Q2.3**	Which machine code instructions of the f function explain by how much the stack pointer moves exactly as observed in Q2.1? (copy the whole f function and prepend a few asteriks (***) at any explanatory instruction)
Your answer for **Q2.3**	

PART C

Page Faults and Address Translation using 4-Level Page Tables

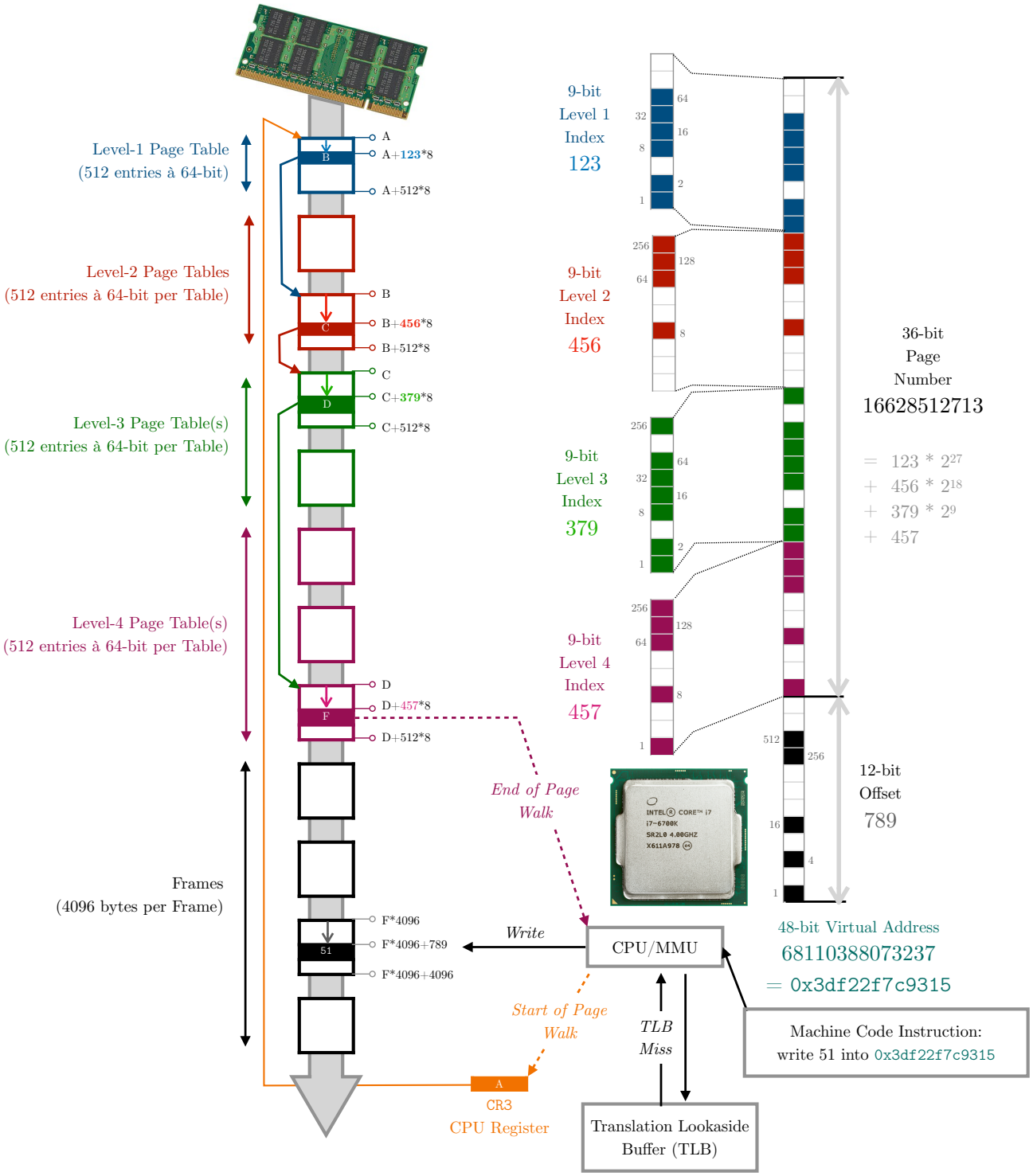
Answer the following questions about the behaviour of `paging.c`

Q & A	`paging.c`
Basic Question **Q3.1**	Use a tool of your choice to monitor (major) page faults caused by a specific process (the code prints some suggested commands). Allocate pages in the simulator and observe if it impacts the number of page faults. How many pages/frames do you need to allocate on the paging simulator running on your system such that a certain number of page faults are attributed to this process (Ubuntu: 100 Major Page Faults, MacOSX: 100000 “Faults”; alternatively: obvious signs of thrashing)?

Your answer for **Q3.1**	
Intermediate Question **Q3.2**	How much larger would the overhead of a (flat) 1-level page table have been compared to the (hierarchical) 4-level page table scheme/tree? (for a similar number of pages/frames allocations as in the answer to **Q3.1**)
Your answer for **Q3.2**	
Bonus Question **Q3.3**	What is an example of a set of page numbers (to allocate a frame for) that would lead to the observed number of page tables and subsequent behaviour of Figure 1 in the Appendix? Entering those exemplary page numbers into the simulator, how much memory usage does it report?
Your answer for **Q3.3**	

APPENDIX

Figure 1: *Example of 4-Level Page Tables*



"OS not needed it is"

Role of OS beyond this example:

- Manages page tables for all processes
- Give CPU/MMU enough information for hardware page walks
  - Updates CR3 register in case of context switches
- Handles page faults
  - TLB miss —> page walk —> page not in memory —> OS
  - TLB hit —> dirty page —> page not in memory —> OS

APPENDIX

Explanation of Figure 1

- *Physical addresses* point directly to a single byte, but are hidden away from user processes both for sake of convenience and security.
- *Virtual addresses* point only indirectly to a single byte, but can be used by user processes as if they were physical addresses on extremely large memory (2^{48} bytes = 256 TB).
- *Address translation* on the CPU/MMU replaces virtual addresses in instructions to physical ones during execution (user process is oblivious to this).
- *Example*:
  - Machine code instructs setting byte value at 48-bit virtual address (`0x3df22f7c9315`) to a new value (51) .
    - CPU/MMU splits virtual address into 36-bit page number (16628512713) and 12-bit offset (789) within the page/frame.
  - CPU/MMU checks if TLB knows frame number for page number 16628512713.
    - TLB miss: Translation Lookaside Buffer (TLB) does not know it.
  - Hardware Page Walk by CPU/MMU:
    - *Note:* page number 16628512713 = 123*2^{27} +456*2^{18} + 379*2^{9}+457
    - Split page number into it's four 9-bit indices (123, 456, 379, 457)
    - Read physical address `A` from `CR3` register on the CPU/MMU.
    - Read 8 bytes (64-bit value) starting from physical address `A+123*8`
      - Check least significant 12 bits to rule out OS is needed (not needed here)
        - Interpret the most significant 52 bits as physical address `B`
    - Read 8 bytes (64-bit value) starting from physical address `B+456*8`
      - Check least significant 12 bits to rule out OS is needed (not needed here)
        - Interpret the most significant 52 bits as physical address `C`
    - Read 8 bytes (64-bit value) starting from physical address `C+379*8`
      - Check least significant 12 bits to rule out OS is needed (not needed here)
        - Interpret the most significant 52 bits as physical address `D`
    - Read 8 bytes (64-bit value) starting from physical address `D+457*8`
      - Check least significant 12 bits to rule out OS is needed (not needed here)
        - Interpret the most significant 52 bits as frame number `F`
  - CPU/MMU sets byte at physical address `F*4096+789` to the new value 51.
- *Caveats:*
  - Number of most significant bits for physical addresses is hardware dependent (Typically: AMD uses 52 bits while Intel uses 40 bits)
  - The figure implies A < B < C < D < F and does not depict any gaps for sake of simplicity (it is not impossible, but not likely)
  - There can be more than one TLB cache (akin to other cache hierarchies), but one could think of all the TLB caches as the "TLB". TLB entries can be outdated and even TLB hits may require OS intervention ("dirty" pages etc).
  - On some systems multi-level page tables do not always have to be in physical memory (which requires using virtual instead of physical addresses in entries). Some tables must always be kept in main memory to handle the page faults related to page tables.