# 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** | A: Page number  B: Page size in bytes  C: Offset in the page  D: Actual virtual address |
| 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\*163+ 15\*162 + 15\*161 + 1\*160). 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** | Hex: 0x555a2ab30014  Starting from the right side, for each digit in the hex, convert it into a decimal and multiply with 16^n-1 where n is the n-th digit.  4\*16^0 + 1\*16^1 + 3\*16^4 + 11\*16^5 + 10\*16^6 + 2\*16^7 + 10\*16^8 + 5\*16^9 + 5\*16^10 + 5\*16^11  = 93845751791636 |
| 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** | Lowest:  virtual address pointing to 1st byte of printAddress function:  hexadecimal: 0x5570706f31e9  decimal: 22934914803 \* 4096 + 489 = 93941411033577  Highest:  virtual address pointing to 1st byte of array on stack:  hexadecimal: 0x7ffda0636100  decimal: 34357249590 \* 4096 + 256 = 140727294320896  46.79TB addressable RAM, which the system doesn’t have.  Only 1 frame contain the bytes of arrayOnHeap. The maximum number of frames that can contain some bytes is 4 if the page size is only 1B. If the page size is >=4B then it is only 1 frame. |

**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** | -O0: The SP always moves 48 bytes per recursive call.    -O3: The SP doesn’t move for the first 7 recursive calls, moves 64 bytes for the 8th-10th recursive call, moves 928 for 100th, 9568 for 1000th and 95968 for 10000th. |
| 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** | 122d: e8 64 ff ff ff call 1196 <f> |
| 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** | 0000000000001196 <f>:      1196:   f3 0f 1e fa             endbr64      119a:   55                \*\*\* push   %rbp      119b:   48 89 e5                mov    %rsp,%rbp      119e:   48 83 ec 20       \*\*\*   sub    $0x20,%rsp      11a2:   48 89 7d e8             mov    %rdi,-0x18(%rbp)      11a6:   48 83 7d e8 01          cmpq   $0x1,-0x18(%rbp)      11ab:   75 0c                   jne    11b9 <f+0x23>      11ad:   b8 00 00 00 00          mov    $0x0,%eax      11b2:   e8 d2 ff ff ff    \*\*\*   call   1189 <getStackPointer>      11b7:   eb 36                   jmp    11ef <f+0x59>      11b9:   b8 00 00 00 00          mov    $0x0,%eax      11be:   e8 c6 ff ff ff          call   1189 <getStackPointer>      11c3:   48 89 45 f0           mov    %rax,-0x10(%rbp)      11c7:   48 8b 45 e8             mov    -0x18(%rbp),%rax      11cb:   48 83 e8 01             sub    $0x1,%rax      11cf:   48 89 c7                mov    %rax,%rdi      11d2:   e8 bf ff ff ff          call   1196 <f>      11d7:   48 89 45 f8             mov    %rax,-0x8(%rbp)      11db:   48 8b 45 f8             mov    -0x8(%rbp),%rax      11df:   48 3b 45 f0             cmp    -0x10(%rbp),%rax      11e3:   7d 06                   jge    11eb <f+0x55>      11e5:   48 8b 45 f8             mov    -0x8(%rbp),%rax      11e9:   eb 04                   jmp    11ef <f+0x59>      11eb:   48 8b 45 f0             mov    -0x10(%rbp),%rax      11ef:   c9                      leave      11f0:   c3                      ret |

**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** | Ubuntu: 45000, process killed, sometimes typing the ps command lags the terminal. |
| 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** | Assume overhead means the total memory consumption  Assume 48-bit virtual address space and 4KiB page size.  Page Table Entry size: 6 bytes  Size of 1-level page table: 2^36 \* 6 = 412GB  Size of 4-level page table:  11 page tables (45000 / 4096 = 10.9)  Only 1 top-level page directory: 1\*2 (2 bytes for 9 bits)  Only 1 level-2 page directory: 1\*2  Only 1 level-3 page directory: 1\*2  11 level-4 page tables needed. 11\*2^9\*6  Total memory: 33.8kB |
| 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** | 122\*2^27 000\*2^18 000\*2^9 001  123\*2^27 457\*2^18 000\*2^9 001  123\*2^27 456\*2^18 377\*2^9 001  123\*2^27 456\*2^18 378\*2^9 001  Page table memory consumption: 40kB  4 frames used: 16kB |

Diagram

Description automatically generated

