Midterm Exam 2

601.229 Computer Systems Fundamentals

November 5, 2021

Complete all questions.

Time: 50 minutes.

I affirm that I have completed this exam without unauthorized assistance from any person, materials, or device.

Signed: Solution

Print name: ____________________

Date: ____________________

Reference

Powers of 2 ($y = 2^x$):

x	0	1	2	3	4	5	6	7	8	9	10	11	12
y	1	2	4	8	16	32	64	128	256	512	1,024	2,048	4,096

x	13	14	15	16
y	8,192	16,384	32,768	65,536

Note that in all questions concerning C:

- `uint8_t` is an 8-bit unsigned integer type
- `uint16_t` is a 16-bit unsigned integer type
- `uint32_t` is a 32-bit unsigned integer type
- `int8_t` is an 8-bit signed two's complement integer type
- `int16_t` is a 16-bit signed two's complement integer type
- `int32_t` is a 32-bit signed two's complement integer type

x86-64 registers:

Callee-saved: `%rbx`, `%rbp`, `%r12`, `%r13`, `%r14`, `%r15`

Caller-saved: `%r10`, `%r11`

Return value: `%rax`

Arguments: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`

Note that argument registers and return value register are effectively caller-saved.

Registers and sub-registers:

Register	Low 32 bits	Low 16 bits	Low 8 bits
`%rax`	`%eax`	`%ax`	`%al`
`%rbx`	`%ebx`	`%bx`	`%bl`
`%rcx`	`%ecx`	`%cx`	`%cl`
`%rdx`	`%edx`	`%dx`	`%dl`
`%rbp`	`%ebp`	`%bp`	`%bpl`
`%rsi`	`%esi`	`%si`	`%sil`
`%rdi`	`%edi`	`%di`	`%dil`
`%r8`	`%r8d`	`%r8w`	`%r8b`
`%r9`	`%r9d`	`%r9w`	`%r9b`
`%r10`	`%r10d`	`%r10w`	`%r10b`
`%r11`	`%r11d`	`%r11w`	`%r11b`
`%r12`	`%r12d`	`%r12w`	`%r12b`
`%r13`	`%r13d`	`%r13w`	`%r13b`
`%r14`	`%r14d`	`%r14w`	`%r14b`
`%r15`	`%r15d`	`%r15w`	`%r15b`

Stack alignment: `%rsp` must contain an address that is a multiple of 16 when any `call` instruction is executed.

Operand size suffixes: **b** = 1 byte, **w** = 2 bytes, **l** = 4 bytes, **q** = 8 bytes (Examples: `movb`, `movw`, `movl`, `movq`)

Question 1. [10 points] Consider the following C++ function, which rearranges the elements of a vector of `int` elements so that they are in the opposite of their original order:

```
void reverse_int_vec(std::vector<int> &v) {
  size_t size = v.size();
  for (size_t i = 0; i < size; i++) {
    // remove the last element
    int last = v.back();
    v.pop_back();

    // insert the element at position i
    v.insert(v.begin() + i, last);
  }
}
```

The following unit tests explain the expected behavior:

```
std::vector<int> testvec = { 1, 2, 3 };
reverse_int_vec(testvec);
assert(3 == testvec[0]);
assert(2 == testvec[1]);
assert(1 == testvec[2]);
```

(a) Briefly explain why the `reverse_int_vec` function will execute slowly when called on a vector with a large number of elements.

Inserting at an arbitrary position in a vector requires all elements at or past the point of the insertion to be moved. So, the cost is proportional to how many elements need to be moved (on average N/2 for this function.)

(b) Briefly suggest a way of implementing `reverse_int_vec` that would perform better on large vectors.

Possible approaches:

- swap first and last elements, then 2nd and 2nd to last elements, etc., until all pairs are swapped
- copy all elements to temporary vector, then copy back to original vector in reversed order

Question 2. [15 points] Consider the following code:

```
1:      a = b * c;
2:      d = e * f;
3:      g = d - a;
```

Assume that all variables (a – g) are CPU registers, and that each statement can be translated to a single machine instruction.

(a) Which, if any, of these statements can be executed in parallel? Explain briefly.

statements 1 and 2 can be freely reordered, since neither depends on or is affected by the other

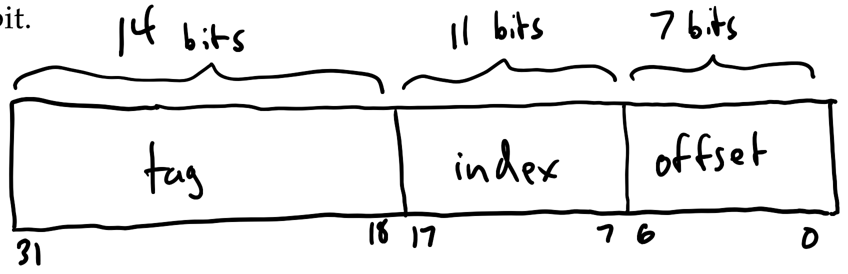
(b) Are there any ordering constraints which would require one statement to be executed after another statement? If so, what are they? Explain briefly.

statement 3 must be executed after both statements 1 and 2, because it uses their results (in a and d) as source operands

Question 3. [15 points] A memory cache for a system with a 32-bit address space has 2048 sets, 4 blocks per set, and the block size is 128 bytes.

Sketch the format of a memory address, showing which bits of the address are the offset, index, and tag. (Reminder: the offset indicates the position of a specific byte in the accessed block, and the index indicates which set of the cache is being accessed.)

Note that bit 0 is the least significant (rightmost) bit and bit 31 is the most significant (leftmost) bit.



block size 128 $= 2^7$ bytes, so offset is 7 bits

2048 sets $= 2^{11}$, so index is 11 bits

tag $= 32 - 11 - 7 = 14$ bits

Question 4. [30 points] In a cache, addresses are 8 bits, blocks are 16 bytes, there are 4 sets, and the cache is 2-way set associative.

low 4 bits are offset

index is 2 bits

Complete the following table. For each address in the *Request* column, indicate the tags of cached blocks after handling the request. Addresses are specified in base-2. Assume each request is a load, and that they execute sequentially (top row is the first in the sequence.) All slots are initially empty. When a block is evicted, select the victim using the LRU (Least Recently Used) replacement policy.

* access

	Set 0		Set 1		Set 2		Set 3	
Request	Slot 0	Slot 1	Slot 0	Slot 1	Slot 0	Slot 1	Slot 0	Slot 1
	empty	empty	empty	empty	empty	empty	empty	empty
10110001							* 10	
00111110								* 00
10111000							*	
00011011			* 00					
10111101							*	
11001110	* 11							↓
11111001								* 11
00001100		* 00						
10011000				* 10				
11001011	*						↓	
00111111		↓					* 00	
01001010		* 01	↓					
11011100	↓	↓	* 11	↓			↓	↓

tag

index

Question 5. [15 points] Consider the following C program:

```
 1:    #include <stdio.h>
 2:
 3:    int main(void) {
 4:      // read a sequence of integer values
 5:      // until a negative value is read,
 6:      // then print the sum
 7:      int sum = 0, done = 0;
 8:      while (!done) {
 9:        int value;
10:        scanf("%d", &value);
11:        if (value < 0) { done = 1; }
12:        else           { sum += value; }
13:      }
14:      printf("Sum is %d\n", sum);
15:      return 0;
16:    }
```

(a) At which point in this program's execution is it most likely that the OS kernel will suspend the execution of the program and allow another program to execute? Explain briefly.

The most likely point where execution will be suspended is the call to scanf at line 10. E.g., if the user has not yet typed any input, the OS kernel will suspend the process until input is available.

(b) For the point you mentioned in (a), briefly explain the most likely reason why the OS kernel would choose to resume execution of the program.

If input arrives (e.g. the user types something and presses enter) the OS kernel will deliver the input and wake up (resume) the suspended process.

Question 6. [15 points] Consider the following C program:

```
 1:     #include <stdio.h>
 2:
 3:     const int arr[] = {
 4:       456, 832, 815, 920, 448, 120, 475, 346, 352, 568,
 5:       486, 70, 594, 9, 111, 908, 871, 188, 159, 527
 6:     };
 7:
 8:     int main(void) {
 9:       const int *p = arr, *end = arr + 20;
10:
11:       int sum = 0;
12:       while (p < end) {
13:         sum += *p;
14:         p++;
15:       }
16:       printf("sum=%d\n", sum);
17:       return 0;
18:     }
```

When the program executes, assume that a page fault occurs in the pointer dereference (`*p`) at line 13.

Briefly explain how the OS kernel will handle the page fault. Be sure to include

- How the OS kernel will find a physical page to allocate, and
- How the OS kernel will initialize the data of the physical page it maps into the process address space

If data is read from and/or written to a storage device (hard disk or SSD), explain.

The OS kernel will choose an unused physical page (if there is one) or will choose a physical page that hasn't been used recently and "steal" it by unmapping it from its current address space. If the stolen page is dirty (i.e., has been modified relative to its data on disk / SSD), its data will be written back to disk / SSD. Then, the OS kernel will read data into the physical page. In the program above, this data will most likely come from the .rodata section of the executable file. Once the physical page contains the correct data, the OS kernel will map it into the address space of the process at the appropriate address and re-execute the instruction which caused the page fault.

[Extra page for answers and/or scratch work.]

[Extra page for answers and/or scratch work.]

[Extra page for answers and/or scratch work.]