# COA1 Exam 3 – Fall 2019

## Name: ________________________________ Computing ID: __________

**Write Letters clearly**: if we are unsure of what you wrote you will get a zero on that problem.
**Bubble and Pledge** the exam or you will lose points.
**Assume** unless otherwise specified:

- all necessary `#include`s have been used
- `char`, `short`, `int`, and `long` are 8-, 16-, 32-, and 64-bits long, respectively
- compilation happens using `clang` on a Linux system
- We use the x86-64 Linux calling convention: arguments are in (in order)
  `rdi`, `rsi`, `rcx`, `rdx`, `r8`, `r9`, and then the stack; and the return value is in `rax`

**Single-select by default**: Multiple select are all clearly marked; answer them by putting 1 or more letters in the box, or writing "`none`" if none should be selected.
**Page-at-a-time Grading**: We scan your exam and grade each page separately. Do not refer to other pages, scrratch paper, etc., in your answer.
**Mark clarifications**: If you need to clarify an answer, do so, and also add a `*` to the top right corner of your answer box.

.....................................................................................

**Question 1 [2 pt]:** What is `0x2501002 & 0xFEDCBA`? Answer in hexadecimal.

______________________________________________________________

**Question 2 [2 pt]:** Suppose we have a 32-bit number consisting only of 0 bits. This number is equal to the value 0 in which of the following number representations?
**Select all that apply** by putting zero or more letters in the box.

**A** 2's complement integer
**B** unsigned integer
**C** biassed integer
**D** IEEE-style floating-point number

Answer:

**Question 3 [2 pt]:** Write a `C` expression equivalent to `~x` without using `~`.

______________________________________________________________

**Question 4 [2 pt]:** Provide a value (in hex or binary) of `x` for which `y = x + 0x3F` sets `y` to `0`. Assume `x` and `y` are both an `unsigned char`.

______________________________________________________________

**Question 5 [2 pt]:** Draw a two-input single-bit mux using basic logic gates; that is, gates that do the same as `(s ? x : y)` for single-bit `s`, `x`, and `y`.

**Question 6 [2 pt]:** What is the value of `y` at the end of this code snippet? Answer in hexadecimal.

```
int x = 0x12345678;
char *p = (char *)&x;
int y = p[1];
```

________________________________________

**Question 7 [2 pt]:** In our toy ISA, the instruction with `icode=6` and `b=3` is called the "indirect load" instruction. Indirect loads are uncessary; write bytes that do the same thing as `63 9A` without using the indirect laod instruction.

________________________________________

**Question 8 [2 pt]:** What value of `i` will **not** cause this code to overflow? Answer as a set of decimal integers: `{}` if none are OK; `{0}` if only 0 is OK; `{0, 1, 2}` if 0, 1, and 2, are OK but 3 is not; etc.

```
int *p = malloc(sizeof(int *) * 2);
p[i] = 2501;
```

________________________________________

**Information for questions 9–12**
Consider the following assembly:

```
foo:
    xorl    %eax, %eax
    testq   %rsi, %rsi
    jle .L2
.L1:
    addq    -8(%rdi,%rsi,8), %rax
    addq    $-1, %rsi
    jg  .L1
.L2:
    retq
```

**Question 9 [2 pt]:** (see above) How many arguments does the function `foo` use?

Answer:

**Question 10 [2 pt]:** (see above) Give an invocation of `foo` that is guaranteed to return `0`. You may use an underscore for an argument whose value does not matter; for example, if `0` is returned when the fifth argument is greater than the sixth, you could say `foo(_, _, _, _, 3, 2)`.

`foo(`

**Question 11 [2 pt]:** (see above) There is an array in this code. What size value (in bytes) are the elements of the array?

Answer:

**Question 12 [2 pt]:** (see above) There is a loop in this code. Which of the following best describes it?

**A** `for(int i=0; i<n; i+=1) { ... }`
**B** `for(int i=n-1; i>=0; i-=1) { ... }`
**C** `do { ... } while(i >= 0);`
**D** `do { ... } while(i < 0);`

Answer:

**Question 13 [2 pt]:** The following function correctly sorts a list of integers, but contains at least one memory leak. Add `free` call(s) to rmeove the leak(s).

```
void mergesort(int *list, int len) {
    if (len <= 1) return;


    int len2 = len>>1;
    int len3 = len-len2;


    int *lst2 = list+len2;


    mergesort(list, len2);
    mergesort(lst2, len3);


    int *lst3 = malloc(4 * len);


    int i=0; int j=0;
    while(i<len2 && j<len3) {
        if (list[i] < lst2[j]) { lst3[i+j] = list[i]; i+=1; }
        else { lst3[i+j] = lst2[j]; j+=1; }
    }


    while(i<len2) { lst3[i+j] = list[i]; i+=1; }
    while(j<len3) { lst3[i+j] = lst2[j]; j+=1; }


    for(i=0; i<len; i+=1) list[i] = lst3[i];


}
```

**Question 14 [1 pt]:** The above code is inefficient, using `malloc` on every recursive invocation. Describe how it could be refactored to only invoke `malloc` once for the entire sorting operation.

**Question 15 [8 pt]:** Implement the `write_long` function documented in the manual page at the end of this exam. Use `write` to display content; you may also use `malloc/calloc/realloc/free` if you want to use heap memory, but must not use any other library functions.

**Information for questions 16–17**
Consider the following C code

```
const char *sptr(const char *a, const char *b) {
    while(*b) {
        int bad = 0;
        for(int i=0; a[i]; i+=1)
            if (a[i] != b[i]) {
                bad = 1;
                break;
            }
        if (!bad) return b;
        b += 1;
    }
    return NULL;
}
```

**Question 16 [2 pt]:** (see above) Write a manual-page-style **description** section for this function.

**Question 17 [2 pt]:** (see above) Write a manual-page-style **example** section for this function.

**Question 18 [2 pt]:** The following code contains a memory error.

```
double f(int n) {
    double *tmp = malloc(8 * n);
    for(int j=n; j>=0; j-=1) tmp[j] = j*j;
    double ans = 0;
    for(int j=0; j<n; j+=1) ans += tmp[j];
    free(tmp);
}
```

What is the error and how would you fix it?

**Question 19 [5 pt]:** Fill in a circle for true statements of the indicated memory region. Note that some rows may be true for both, only one, or neither.

| The stack | The heap | |
|---|---|---|
| ○ | ○ | is used to track recursive calls |
| ○ | ○ | can be leaked if there is a memory leak |
| ○ | ○ | is used by some C programs but not others |
| ○ | ○ | contains both code and data |
| ○ | ○ | can be used for `struct`s and arrays |

## Pledge:

On my honor as a student, I have neither given nor received aid on this exam.

Your signature here

# Our Example ISA

## Instruction Breakdown

Treat each instruction's first byte as having four parts:

| bits | name | meaning |
|---|---|---|
| 7 | reserved | If set, this instruction is reserved for future definition. |
| [4, 7) | `icode` | Specifies what action to take |
| [2, 4) | `a` | The index of a register |
| [0, 2) | `b` | The index of another register, or details about icode |

Some instructions use additional bytes, described below as "memory at `pc + 1`" or the like.

## Instructions

If `reserved` is 0, consult the table below. In it, `rA` means "the value stored in register number `a`" and `rB` means "the value stored in register number `b`."

| `icode` | `b` | Behavior | add to `pc` |
|---|---|---|---|
| 0 | any | `rA = rB` | 1 |
| 1 | any | `rA += rB` | 1 |
| 2 | any | `rA &= rB` | 1 |
| 3 | any | `rA` = read from memory at address `rB` | 1 |
| 4 | any | write `rA` to memory at address `rB` | 1 |
| 5 | 0 | `rA = ~rA` | 1 |
| 5 | 1 | `rA = -rA` | 1 |
| 5 | 2 | `rA = !rA` | 1 |
| 5 | 3 | `rA = pc` | 1 |
| 6 | 0 | `rA` = read from memory at `pc + 1` | 2 |
| 6 | 1 | `rA` += read from memory at `pc + 1` | 2 |
| 6 | 2 | `rA` &= read from memory at `pc + 1` | 2 |
| 6 | 3 | `rA` = read from memory at the address stored at `pc + 1` | 2 |
| 7 | any | if `rA <= 0`, set `pc = rB` | N/A |
| 7 | any | if `rA > 0`, do nothing | 1 |

If `reserved` is 1, the above table does not define what the instruction means, but some other source (such as a question on this exam) might. If it has no defined meaning either here or elsewhere, leave the `pc` and all other registers and memory values unchanged.

# NAME — malloc, free, calloc, realloc

## SYNOPSIS

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

## DESCRIPTION

The **malloc**() function allocates *size* bytes and returns a pointer to the allocated memory. *The memory is not initialized.* If *size* is 0, then **malloc**() returns either NULL, or a unique pointer value that can later be successfully passed to **free**().

The **free**() function frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc**(), **calloc**(), or **realloc**(). Otherwise, or if *free(ptr)* has already been called before, undefined behavior occurs. If *ptr* is NULL, no operation is performed.

The **calloc**() function allocates memory for an array of *nmemb* elements of *size* bytes each and returns a pointer to the allocated memory. The memory is set to zero. If *nmemb* or *size* is 0, then **calloc**() returns either NULL, or a unique pointer value that can later be successfully passed to **free**().

The **realloc**() function changes the size of the memory block pointed to by *ptr* to *size* bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will *not* be initialized. If *ptr* is NULL, then the call is equivalent to *malloc(size)*, for all values of *size*; if *size* is equal to zero, and *ptr* is not NULL, then the call is equivalent to *free(ptr)*. Unless *ptr* is NULL, it must have been returned by an earlier call to **malloc**(), **calloc**(), or **realloc**(). If the area pointed to was moved, a *free(ptr)* is done.

## RETURN VALUE

The **malloc**() and **calloc**() functions return a pointer to the allocated memory, which is suitably aligned for any built-in type. On error, these functions return NULL. NULL may also be returned by a successful call to **malloc**() with a *size* of zero, or by a successful call to **calloc**() with *nmemb* or *size* equal to zero.

The **free**() function returns no value.

The **realloc**() function returns a pointer to the newly allocated memory, which is suitably aligned for any built-in type and may be different from *ptr*, or NULL if the request fails. If *size* was equal to 0, either NULL or a pointer suitable to be passed to **free**() is returned. If **realloc**() fails, the original block is left untouched; it is not freed or moved.

# NAME — write

## SYNOPSIS

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

## DESCRIPTION

**write**() writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.

## RETURN VALUE

On success, the number of bytes written is returned. On error, -1 is returned, and *errno* is set to indicate the cause of the error.

## ERRORS

**EAGAIN** The file descriptor *fd* has been marked nonblocking (**O_NONBLOCK**), and the write would block. See **open**(2) for further details on the **O_NONBLOCK** flag.

**EBADF** *fd* is not a valid file descriptor or is not open for writing.

**EDQUOT** The user's quota of disk blocks on the filesystem containing the file referred to by *fd* has been exhausted.

**EFAULT** *buf* is outside your accessible address space.

**EFBIG** An attempt was made to write a file that exceeds the maximum file size or the process's file size limit, or to write at a position past the maximum allowed offset.

**EINTR** The call was interrupted by a signal before any data was written; see **signal**(7).

**EINVAL** *fd* is attached to an object which is unsuitable for writing; or the file was opened with the **O_DIRECT** flag, and either the address specified in *buf*, the value specified in *count*, or the file offset is not suitably aligned.

**EIO** A low-level I/O error occurred while modifying the inode. This error may relate to the write-back of data written by an earlier **write**(), which may have been issued to a different file descriptor on the same file.

**ENOSPC** The device containing the file referred to by *fd* has no room for the data.

**EPERM** The operation was prevented by a file seal; see **fcntl**(2).

Other errors may occur, depending on the object connected to *fd*.

## NOTES

For the purposes of this exam, you may assume that **write** either writes all of its buffer or none of it, and thus that the return value is either -1 or *count*. That is not true of the actual **write** function, which can sometimes write only part of the buffer.

# NAME — write_long

## SYNOPSIS

```
#include <coa1exam3.h>

ssize_t write_long(int fd, long num);
```

## DESCRIPTION

**write_long**() writes an integer to the the file referred to by the file descriptor *fd*. It does this in base 10, with a leading `-` if the number if negative.

## RETURN VALUE

On success, the number of bytes written is returned. On error, -1 is returned, and *errno* is set to indicate the cause of the error.

## ERRORS

All errors of **write_long** are caused by the underlying call to **write**(2) and are passed through unchanged by **write_long**(). See the manual page for **write**(2) for more.

## EXAMPLE

The following code will write 6 characters to file descriptor 0: in particular, "**2501-2**".

```
    write_long(0, 2501);
    write_long(0, -002);
```

---

# ASCII Table

The following is a subset of the ASCII table:

| Char | Hex | Char | Hex | Char | Hex | Char | Hex | Char | Hex |
|---|---|---|---|---|---|---|---|---|---|
| `' '` | 0x20 | `'A'` | 0x41 | `'N'` | 0x4E | `'a'` | 0x61 | `'n'` | 0x6E |
| `'+'` | 0x2B | `'B'` | 0x42 | `'O'` | 0x4F | `'b'` | 0x62 | `'o'` | 0x6F |
| `'-'` | 0x2D | `'C'` | 0x43 | `'P'` | 0x50 | `'c'` | 0x63 | `'p'` | 0x70 |
| `'0'` | 0x30 | `'D'` | 0x44 | `'Q'` | 0x51 | `'d'` | 0x64 | `'q'` | 0x71 |
| `'1'` | 0x31 | `'E'` | 0x45 | `'R'` | 0x52 | `'e'` | 0x65 | `'r'` | 0x72 |
| `'2'` | 0x32 | `'F'` | 0x46 | `'S'` | 0x53 | `'f'` | 0x66 | `'s'` | 0x73 |
| `'3'` | 0x33 | `'G'` | 0x47 | `'T'` | 0x54 | `'g'` | 0x67 | `'t'` | 0x74 |
| `'4'` | 0x34 | `'H'` | 0x48 | `'U'` | 0x55 | `'h'` | 0x68 | `'u'` | 0x75 |
| `'5'` | 0x35 | `'I'` | 0x49 | `'V'` | 0x56 | `'i'` | 0x69 | `'v'` | 0x76 |
| `'6'` | 0x36 | `'J'` | 0x4A | `'W'` | 0x57 | `'j'` | 0x6A | `'w'` | 0x77 |
| `'7'` | 0x37 | `'K'` | 0x4B | `'X'` | 0x58 | `'k'` | 0x6B | `'x'` | 0x78 |
| `'8'` | 0x38 | `'L'` | 0x4C | `'Y'` | 0x59 | `'l'` | 0x6C | `'y'` | 0x79 |
| `'9'` | 0x39 | `'M'` | 0x4D | `'Z'` | 0x5A | `'m'` | 0x6D | `'z'` | 0x7A |