# COS 217, Fall 2023
# Final Exam

---

This exam consists of 6 questions, and you have 120 minutes – budget your time wisely. ***Do all of your work on these pages (using the back for scratch space), and give the answer in the space provided.*** Note that the exams will be scanned and graded online, so ***ONLY ANSWERS IN THE BOXES WILL BE GRADED.*** Assume the ArmLab/Linux/C/gcc217 environment unless otherwise stated. This is a closed-book, closed-note exam, and only 1 two-sided page of notes is allowed. Please place items that you will not need out of view in your bag or under your working space at this time. Electronic devices such as cell phones, laptops, tablets, etc. may not be used during this exam.

---

**Name:** | **NetID:** | **Precept:**

P01 MW 1:30 Christopher Moretti
P02 MW 3:30 Christopher Moretti
P03 TTh 12:30 Guðni Nathan Gunnarsson
P04 TTh 12:30 Sam Ginzburg
P05 TTh 1:30 Indu Panigrahi
P06 TTh 1:30 Gongqi Huang
P07 TTh 2:30 Nanqinqin Li
P09 TTh 3:30 Jianan Lu
P10 TTh 7:30 Dwaha Daud

This examination is administered under the Princeton University Honor Code. Students should sit one seat apart from each other, and refrain from talking to other students during the exam. All suspected violations of the Honor Code must be reported to honor@princeton.edu.

***Write out and sign the Honor Code pledge before turning in the test:***

*"I pledge my honor that I have not violated the Honor Code during this examination."*

**Pledge, written out exactly as above:**

**Signature:**

## 1. ADT vs. AO

An undirected graph is defined as a collection of nodes with edges. These edges do *not* have a direction: if node A and node B share an edge, A is a neighbor of B, and B is a neighbor of A. We want to design a module for such a graph. Assume that we can have ***only one*** graph in the entire program, and assume that the graph can contain ***more than one*** node.

(a) Would this undirected graph API be better represented as an abstract object (AO) or an abstract data type (ADT)? ***Write only ADT or AO:*** *(3 pts total for a and b)*

AO

(b) Write a brief (a few words or a sentence) ***justification*** for your answer in part (a):

The problem statement indicates that there is only one graph in the program. The AO abstraction reflects this "singleton pattern" better than ADTs, which can have multiple instances instantiated.

(c) Would a node in this undirected graph be better represented as an abstract object (AO) or an abstract data type (ADT)? ***Write only ADT or AO:*** *(3 pts total for c and d)*

ADT

(d) Write a brief (a few words or a sentence) ***justification*** for your answer in part (c):

The problem statement indicates that there may be **many** nodes in the graph. The ADT design reflects this: we can instantiate multiple distinct nodes that coexist.

(e) Assume that each node stores a unique string and can have ***up to 5 neighbors***. Write the definition of a `struct GraphNode` to represent each node in the graph, assuming that ***no storage outside of the struct*** will be allocated to represent the neighbor pointers: *(3 pts)*

```
struct GraphNode {

   char *string;
   struct GraphNode *neighbors[5];
   /* alternative: struct GraphNode *n1, *n2, *n3, *n4, *n5; */









};
```

(f) Now instead assume that each node can have an ***arbitrary number of neighbors*** (still together with a string label) and that storage for the neighbor pointers will be allocated on the heap. Also assume that it is a requirement of the API that the ***number of neighbors*** of a node be available as a ***constant-time*** operation. Write the definition of a `struct GraphNode` to represent each node in the graph, assuming these new requirements: *(3 pts)*

```
struct GraphNode {


   char *s;
   size_t n;
   struct GraphNode **neighbors;









};
```

## 2. DMM errors

Consider the following C program, with one piece of code missing:

```
#include <stdlib.h>
#include <stdio.h>

void square(int i, int **piResult)
{
    *piResult = (int *) malloc(sizeof(int));
    if (*piResult == NULL) {
        fprintf(stderr, "Insufficient memory available\n");
        return;
    }

    **piResult = i * i;
}

int main()
{
    int i = 5;
    int *piResult = NULL;
    int *piExtra = NULL;

    square(i, &piResult);

    /* INSERT CODE HERE */

    printf("%d is one plus the square of %d", *piResult, i);
    return 0;
}
```

***Identify the memory management bugs*** (if any) that occur if the `/* INSERT CODE HERE */` comment is replaced with each of the code snippets in parts a-e. ***Write the letters for ALL relevant bugs in the boxes at right***, or write "None" if none of the issues are present. Consider bugs in the entire resulting program, not just the provided code snippets.

**A**: accesses unallocated memory
**B**: accesses freed memory (dangling pointer)
**C**: leaks memory (does not free allocated memory)
**D**: frees unallocated memory
**E**: double-frees allocated memory
**None**: none of the memory management errors above are present

| | **Code** *(2 pts ea)* | **Bug(s)** |
|---|---|---|
| (a) | `piResult++;` | A C |
| (b) | `piResult++;`<br>`free(piResult);` | A C D |
| (c) | `*piResult = *piResult + 1;` | C<br>(A is also acceptable. See below.) |
| (d) | `*piResult = *piResult + 1;`<br>`free(piResult);` | B<br>(A is also acceptable. See below.) |
| (e) | `piExtra = piResult;`<br>`*piResult = *piResult + 1;`<br>`free(piExtra);` | B<br>(A is also acceptable. See below.) |
| (f) | `free(piResult);`<br>`piExtra = piResult;`<br>`*piExtra = *piExtra + 1;`<br>`free(piExtra);` | B E<br>(A is also acceptable: square should exit() on malloc failure, or main should guard printf.) |

## 3. VM, Bitwhacking, and Hash Tables

The heart of a virtual memory implementation is the differentiation between virtual and physical addresses. In this question, you will write C code that translates from a virtual address to a physical address.

On AArch64, 64-bit addresses can be divided into a ***48-bit page number*** in the most-significant bits, and a ***16-bit offset*** that is identical between virtual and physical addresses. Assume that you are provided with a function `virt_to_phys_page()` that returns the physical page number corresponding to a given virtual page number:

```
size_t virt_to_phys_page(size_t virt_page);
```

(a) ***Implement the following function***, which returns the physical address corresponding to a given virtual address, calling `virt_to_phys_page()` as appropriate. Assume that the virtual address passed in as `virt_addr` is valid and mapped-in to physical memory. *(8 pts)*

```
size_t virt_to_phys_addr(size_t virt_addr)
{
```

There are many correct options. Each will require isolating the virtual page number, saving the offset, getting the physical page number, and putting the physical page number and offset into place in the result to return.

```
    size_t virt_page, offset, phys_page, phys_addr;
    virt_page = virt_addr >> 16;
    offset = virt_addr & 0xFFFF; /* or (virt_addr << 48) >> 48 */
    phys_page = virt_to_phys_page(virt_page);
    phys_addr = phys_page << 16;
    phys_addr = phys_addr | offset; /* or + or ^, but not || */
    return phys_addr;
```

```
}
```

Now assume that the virtual-to-physical page mappings (the "page tables") are implemented as a hash table, using the following data structure:

```
struct Binding {
   size_t virt_page, phys_page;
   struct Binding *next;
};
struct PageTable {
   struct Binding *buckets[BUCKET_COUNT];
};
struct PageTable page_table;  /* Global variable */
extern size_t hash_page(size_t page);  /* Hash function */
```

(b) ***Implement*** a version of `virt_to_phys_page()` that uses the `page_table` global variable, calling `hash_page()` as appropriate. ***If the virtual page number is not found, return 0.*** *(10 pts)*

```
size_t virt_to_phys_page(size_t virt_page)
{


   struct Binding *cur;
   cur = page_table.buckets[hash_page(virt_page) % BUCKET_COUNT];
   while(cur != NULL) {
      if(cur->virt_page == virt_page)
         return cur->phys_page;
      cur = cur->next;
   }
   return 0;









}
```

## 4. Memory Sections

The following questions ask you to identify the most relevant memory section, from the following list: TEXT, RODATA, DATA, BSS, STACK, HEAP. ***Write the (single) section corresponding to the answer in the boxes at right.***

For some of the questions, you will need to refer to the following program:

```
#include <stdio.h>

int result;

int multiply(int left, int right)
{
     return left * right;
}

int main()
{
     const char *greetString = "Choose two numbers!\n";
     int firstNumber;
     static int secondNumber = 0;
     int (*mul)(int, int) = &multiply;

     printf("%s", greetString);
     scanf("%d %d", &firstNumber, &secondNumber);

     result = (*mul)(firstNumber, secondNumber);

     printf("The result is: %d\n", result);
     return 0;
}
```

| | **Question** *(2 pts ea)* | **Memory section: TEXT, RODATA, DATA, BSS, STACK, HEAP** |
|---|---|---|
| (a) | In AArch64 assembly language, which section of memory does the stack pointer (`sp`) register ***point to***? | STACK |
| (b) | In AArch64 assembly language, which section of memory does the program counter (`pc`) register ***point to***? | TEXT |
| (c) | In the program shown above, in which memory section is the `result` variable ***stored***? | BSS (process duration, not initialized in def.) |
| (d) | In the program shown above, in which memory section is the `greetString` variable ***stored***? | STACK (the variable, not the string contents) |
| (e) | In the program shown above, in which memory section is the `secondNumber` variable ***stored***? | DATA (process duration, definition initializes) |
| (f) | In the program shown above, which memory section does the `mul` variable ***point to***? | TEXT (the target, not the pointer mul itself) |
| (g) | In the program shown above, in which memory section is the string literal `"The result is: %d\n"` ***stored***? | RODATA |
| (h) | Given a `const char` array (also known as a string) that has been initialized to `"Hello, world!\n"`, in which memory section could that array variable ***NOT be stored***? | TEXT (just machine code, not strings) |

## AArch64 Assembly Language Reference for Questions 5 and 6

| Registers / Instructions | Description |
|---|---|
| `x0..x30, xzr / w0..w30, wzr` | 8-byte / 4-byte registers; `xzr` and `wzr` hold 0 |
| `x0..x7 / w0..w7` | ***Caller**-saved* scratch registers, hold parameters |
| `x0 / w0` | Holds return value |
| `x19..x28 / w19..w28` | ***Callee**-saved* scratch registers |
| `x30` | Link register, holds return address |
| `sp` | Stack pointer register |
| `mov dst, src` | Copy `src` (register or immediate value) to `dst` |
| `add/sub/mul dst, src1, src2` | Add / subtract / multiply `src1` and `src2`, storing result in `dst` |
| `adds/subs/muls dst, src1, src2` | Same as above, but also set condition flags |
| `cmp src1, src2` | Set condition flags based on comparison of `src1` and `src2` |
| `beq / bne label` | Branch to `label` if equal / not equal |
| `blt / ble / bgt / bge label` | Branch to `label` if < / <= / > / >= (signed) |
| `blo / bls / bhi / bhs label` | Branch to `label` if < / <= / > / >= (unsigned) |
| `b label` | Branch to `label` unconditionally |
| `bl label` | Call function at `label` and save return address in `x30` |
| `ret` | Return to code at address in `x30` |
| `ldr dst, [Xn]` | Load from memory address in register `Xn` into register `dst` |
| `str src, [Xn]` | Store into memory address in register `Xn` from register `src` |
| `[Xn, offset]` | *Immediate offset* addressing mode: *addr* = reg[`Xn`] + `offset` |
| `[Xn, Xm]` | *Register offset* addressing mode: *addr* = reg[`Xn`] + reg[`Xm`] |
| `[Xn, Xm, LSL n]` | *Scaled register offset* addressing mode:<br>*addr* = reg[`Xn`] + (reg[`Xm`] << `n`), `n` = 3 for 8-byte, 2 for 4-byte |

## 5. Assembly Errors

For each of the following snippets of C code, analyze the corresponding AArch64 assembly and ***choose the single most appropriate option*** depending on whether the code: *(3 pts ea)*

**A**: Has no errors
**B**: Doesn't assemble
**C**: Causes a segmentation fault or other run-time crash
**D**: Doesn't segfault, but doesn't achieve the C code's intended purpose

(a) C code:

```
long volume = length * width * height;
```

Assembly:

```
// Assume that volume, length, width, and height are on the stack at
//   offsets 8, 16, 24, and 32, respectively.
ldr x0, [sp, 16]
ldr x1, [sp, 24]
ldr x2, [sp, 32]
mul x0, x1
mul x0, x2
str x0, [sp, 8]
```

Answer (A-D):

B

mul should have 3 operands, not 2.

(b) C code:

```
extern int addTwoNumbers(int, int);
int sum = addTwoNumbers(200, 17);
```

Assembly:

```
// Assume that sum is at offset 8 on the stack.
mov w1, 200
mov w2, 17
bl addTwoNumbers
str w0, [sp, 8]
```

Answer (A-D):

D

addTwoNumbers will look for its arguments in w0 and w1, not w1 and w2.

(c) C code:

```
struct List {
    long value;
    struct List *next;
};
// linkedList is a pointer to the first node of a linked list
// containing **at least two** nodes
long secondNodeValue = linkedList->next->value
```

Assembly:

```
// Assume that secondNodeValue is at offset 8 on the stack.
// Assume that linkedList is at offset 16 on the stack.
ldr x0, [sp, 16]
add x0, x0, 8
ldr x0, [x0]
ldr x0, [x0]
str x0, [sp, 8]
```

Answer (A-D):

A

1st ldr loads linkedList ptr from stack; add gets address of linkedList-next field; 2nd ldr gets value of field, which is address of the 2nd node in list; 3rd ldr gets value field of 2nd node; str copies into variable

(d) C code:

```
long arr[20];
long i = 0;
while (i < 20) {
    arr[i] = 0;
    i++;
}
```

Assembly:

```
// Assume that i is stored in callee-saved register x19.
// Assume that a pointer to arr[0] is stored in register x20.
mov x19, xzr
loop1:
str xzr, [x20, x19, lsl 3]
add x19, x19, 1
b loop1
endloop1:
```

Answer (A-D): infinite loop that writes unbounded off the end of the array

C

## 6. Bubble Sort Asm

Consider this correct C implementation of the naïve sorting algorithm BubbleSort:

```
void swap(int array[], size_t i, size_t j)
{
   int temp = array[i];
   array[i] = array[j];
   array[j] = temp;
}

void bubbleInner(int array[], size_t n, size_t pass)
{
    size_t k = 0;
    while (k < n - pass - 1) {
        if (array[k] > array[k + 1])
            swap(array, k, k + 1);
        k++;
    }
}

void bubbleSort(int array[], size_t n)
{
    size_t pass;
    for (pass = 0; pass < n - 1; pass++)
        bubbleInner(array, n, pass);
}
```

And here is a ***buggy*** implementation of `bubbleInner()` in AArch64 assembly language:

```
1   .equ ARRAY, 8
2   .equ N, 16
3   .equ PASS, 24
4   .equ K, 32
5
6   .global bubbleInner
7  bubbleInner:
8   sub sp, sp, 48
9   str x30, [sp]
```

```
10   str x0, [sp, ARRAY]
11   str x1, [sp, N]
12   str x2, [sp, PASS]
13   str xzr, [sp, K]
14
15 inner_loop:
16   ldr x0, [sp, K]
17   ldr x1, [sp, N]
18   ldr x2, [sp, PASS]
19   sub x3, x1, x2
20   sub x3, x3, 1
21   cmp x0, x3
22   bge afterInner
23
24   add x3, x0, 1
25   ldr x4, [sp, ARRAY]
26   ldr w5, [x4, x0, lsl 3]
27   ldr w6, [x4, x3, lsl 3]
28   cmp w5, w6
29   ble afterInner
30
31   mov x1, x0
32   mov x0, x4
33   mov x2, x3
34   bl swap
35
36 afterIf:
37   ldr x2, [sp, K]
38   add x2, x2, 1
39   str x2, [sp, K]
40   bl inner_loop
41
42 afterInner:
43   ldr x30, [sp]
44   add sp, sp, 48
45   ret
```

This program has a whopping ***four (4)*** mistakes, each with a different characteristic.

For each of the following bugs, ***write the line number(s) of the offending instruction(s).*** You need not identify the bug, just the line number.
*(2 pts ea)*

(a) 2 buggy instructions (with the same bug) that will cause the assembler to emit an error.

26, 27

These instructions should use lsl 2, since the array elements are ints.

b) 1 buggy instruction that will cause `bubbleSort` to fail to sort many simple array examples.

29

This should be ble afterIf to avoid calling swap, not break out of the loop entirely.

(c) 1 buggy instruction that could cause `bubbleSort` to fail a large stress test.

22

The conditional branch should be bhs, because k, n, and pass are all of unsigned type size_t.

(d) 1 buggy instruction that does not compromise `bubbleSort`'s behavior as-is, but would cause the program to do the wrong thing if we chose to optimize by inlining `swap` and not constructing a stackframe at all.

40

The branch should be a simple unconditional branch b, not a function call bl that sets x30.

(e) Now write an "optimized" AArch64 assembly language version of `bubbleSort()` using the same strategy that you used in Part 2e of A5 (`bigintaddopt.s`) – i.e., storing all local variables and parameters in ***callee-saved registers*** instead of on the stack. You should assume that the `.equ` and `.req` definitions at right have been included in your assembly source file. *(14 pts)*

```
array .req x19
n     .req x20
pass  .req x21
.equ oldX19, 8
.equ oldX20, 16
.equ oldX21, 24
```

```
    .global bubbleSort
bubbleSort:
    sub sp, sp, 32
    str x30, [sp]
    str x19, [sp, oldX19]
    str x20, [sp, oldX20]
    str x21, [sp, oldX21]

    mov array, x0
    mov n, x1
    // pass = 0;
    mov pass, xzr // many valid alternatives
loop:
    // if(pass >= n - 1) goto endLoop;
    sub x3, n, 1
    cmp pass, x3
    bhs endLoop

    // bubbleInner(array, n, pass);
    mov x0, array
    mov x1, n
    mov x2, pass
    bl bubbleInner

    // pass++;
    add pass, pass, 1
    // goto loop;
    b loop

endLoop:
    ldr x19, [sp, oldX19]
    ldr x20, [sp, oldX20]
    ldr x21, [sp, oldX21]
    ldr x30, [sp]
    add sp, sp, 32
    ret
```

(f) Finally, write the assembly code for the `swap()` function, using ***exactly 5 instructions***: 4 memory accesses using ***scaled register offset*** memory operands, plus a return instruction (which is provided for you). ***Do not*** write a prolog/epilog to manage a stackframe; thus, ***use only caller-saved scratch registers***. *(8 pts)*

```
    .global swap
swap:


    ldr w3, [x0, x1, lsl 2]
    ldr w4, [x0, x2, lsl 2]
    str w4, [x0, x1, lsl 2]
    str w3, [x0, x2, lsl 2]









    ret
```