This document is a mock midterm exam for the 2021 offering of COMP0019. This paper is unmarked and unassessed; it is provided for students to take so that they can gauge their understanding of material in the module thus far, and gain experience with the style and difficulty level of exam questions in a Computer Systems class. The instructors recommend that students prepare for the mock midterm by studying as they would for an assessed paper. When taking the mock midterm, students should allow themselves 1 hour 25 minutes of working time. The questions are intended to be answered under exam conditions, without any access to books, notes, or Internet references.

Computer Systems Mock Midterm Exam, COMP0019, 2021

Answer **ALL** questions. Type your answers to Parts One and Two in a single answer document. *Please be succinct.* In open-ended questions, we are looking for to-the-point answers, and will not award marks for off-topic responses.

Show your work. If your solution requires multiple steps of reasoning but you do not show the steps, we cannot award marks for the final answer. Please state any assumptions you make.

GOOD LUCK!

Marks for each part of each question are indicated in square brackets

Calculators are permitted

		Raw score	**Earned marks**	**Possible marks**
Part ONE	Question 1			20
	Question 2			20
Part TWO				40
GRAND TOTAL				80

Part ONE

1. **C Data Representation, Programming Constructs, and Undefined Behavior**

   a. Consider the following C code fragment, in which each "[arbitrary unsigned long constant value]" can be any constant unsigned long value.

```
unsigned long a = [arbitrary unsigned long constant value];
unsigned long b = [arbitrary unsigned long constant value];
unsigned long x = [arbitrary unsigned long constant value];
unsigned long r = 0;

r = (- (unsigned long) (a < b)) & x;
```

   Rewrite the final line of the above C code fragment to replace the right-hand side of the assignment (i.e., everything after the equals sign) with a single C `?` `:` conditional expression that assigns the same value to `r`. You may not use any artithmetic operators other than a single comparison operator in your expression.
   Hint: Note that the expression `(a < b)` *can only ever evaluate to 0 or 1.*

if a<b - 1 0xffff -f x

if a>=b - 0. 0&x = 0.

r = a < b ? x : 0

[3 marks]

b. Consider the expression below, in which `T` is some standard integer C type:

```
(T) -1 == (T) 255
```

On x86-64, the above expression is true when type `T` is either of two standard C integer types of some size and signedness (considering only types built into the compiler; not ones defined in include files, such as `intN_t`, etc.). What are those two types?

Hint: When one converts from a larger signed integer type (consisting of more bytes) to a smaller signed integer type (consisting of fewer bytes) in C on x86-64, the result simply consists of as many of the least significant bits in the larger signed integer value that fit in the smaller signed integer type.

unsigned char. / char.

[4 marks]

c. Suppose that C variable `x` has type `int` on an x86-64 machine. Can the expression:

```
x - 1 + 1
```

ever yield undefined behavior (UB)? If so, give at least one example of a value of `x` where UB results. Whether you answer yes or no, explain your reasoning.

Hint: Remember that in C, the addition and subtraction operators are left-associative.

x: INT_MIN. signed overflow is UB.

[3 marks]

d. Consider the following C program running on an x86-64 machine under Linux:

```
#include <stdio.h>

int *foo(int y)
{
  int x;

  x = y + 1;
  return &x;
}

int main(int argc, char **argv)
{
  int *px;

  px = foo(19);
  printf("*px is %d\n", *px);
}
```

i. The above code includes undefined behavior (UB). What exactly is the UB in the above code? Using your knowledge of how procedure calls behave in C on x86-64 and Linux, explain exactly why the C language standard classifies the relevant part of the code above as UB.

return &x is UB.
x is on stack, upon exit,
px → an invalid address

[4 marks]

ii. Below on the left is the x86-64 assembly code that results when compiling the above C code with the command `cc -Og -o ub ub.c`. And below on the right is the x86-64 assembly code that results when compiling the above C code with the command `cc -O2 -o ub ub.c`.

```
_foo:
        pushq   %rbp
        movq    %rsp, %rbp
        leaq    -4(%rbp), %rax
        popq    %rbp
        retq

_main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $19, %edi
        callq   _foo
        movl    (%rax), %esi
        movq    L_.str, %rdi
        xorl    %eax, %eax
        callq   _printf
        xorl    %eax, %eax
        popq    %rbp
        retq

L_.str:
        .asciz  "*px is %d\n"
```

```
_foo:
        pushq   %rbp
        movq    %rsp, %rbp
        leaq    -4(%rbp), %rax
        popq    %rbp
        retq

_main:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    L_.str, %rdi
        xorl    %eax, %eax
        callq   _printf
        xorl    %eax, %eax
        popq    %rbp
        retq

L_.str:
        .asciz  "*px is %d\n"
```

Notes: `L_.str` *is just a symbolic label for the first argument to* `printf()` *in the C source; it evaluates in assembly to the memory address holding the bytes of that string given as a constant in the C source. And* `callq _printf` *is just a procedure invocation to the* `printf()` *function in the standard C library.*

Question continues on next page!

The assembly code generated in the two compilations differs. Explain what is functionally different about the optimized compilation's assembly code on the right vs. the unoptimized compilation's assembly code on the left (don't just quote instructions—explain the consequence), what effect you would expect this difference to have on the output of the program when run, and why one compiler invocation produces code on the right that differs in this way from another compiler invocation that produces the code on the left.

unoptimised code calls foo(); and puts the result in (rax) to esi for print.
ptr
likely to print ≥0.

[6 marks]

for optimised version, foo()
is not even invoked due to UB.
it would just be a random value
in ptr of. esi. very likely
seg fault.

[Total 20 marks]

Anything can happen in UB.
for compiler, less code = faster.

2. **CPU Caches and `malloc()`**

Consider the design of a simple `malloc()` dynamic memory allocator that only supports allocation requests up to 112 bytes. This `malloc()` implementation organizes the heap in fixed-size 128-byte blocks, where each block can only be used for a single allocation, and these 128-byte blocks each begin at 128-byte-aligned memory addresses. The first 16 bytes of each 128-byte block are used for a metadata header describing the block's allocation status. There is no footer.

This limited `malloc()` implementation is running on a CPU with a direct-mapped L1 data cache and no further data caches. The L1 data cache has a block size of 16 bytes and 128 sets.

a. What is the capacity of the L1 data cache? Show your derivation.

16 × 128 = 2048 bytes

[2 marks]

b. Consider a variant of this L1 data cache with the same number of sets and same block size as above, but that is 4-way set associative. Does the capacity of this variant of the L1 data cache differ from that of the original L1 data cache above, and if so, what is its capacity? Show your derivation.

every set can now store 4 × 16 = 64 bytes

2048 × 4 = 8192 bytes

[2 marks]

c. How many different L1 cache sets (regardless of which of the two variants of the L1 cache described above is used) can the initial byte of all 128-byte blocks of the heap map to? Show your full derivation.

[6 marks]

d. Suppose a program using this limited `malloc()` implementation makes 128 allocations, each for the exact same `struct` type whose length is under the 112-byte maximum of the limited implementation. Suppose further that this program repeatedly accesses the *same* data field in turn of each of the `structs` located within the payloads of those 128 allocations' blocks.

For the direct-mapped variant of the L1 data cache described above, how many sets in the cache can these `struct` field accesses map to? Assume that the field accesses do not span cache blocks or sets.

[4 marks]

e. The `struct` field access pattern scenario in the previous part of this question will result in frequent L1 data cache misses. Which of the categories of cache misses will most of these cache misses fall under? Justify your answer by explaining how the field access pattern yields that category of cache miss.

[2 marks]

f. Suppose the design of the L1 data cache were changed to be fully associative, with 128 lines in total, and all other parameters as in the initial variant of the L1 data cache at the start of this question.

Suppose further that an application uses the same limited `malloc()` dynamic memory allocator described above to allocate 500 `structs` that fit within the 112-byte per-allocation limit. This application then accesses the same field repeatedly across all 500 of these `structs`.

This workload will exhibit frequent cache misses. What category of cache miss will these predominantly be? Justify your answer by explaining how the field access pattern and cache design yield that category of cache miss.

[2 marks]

g. How many set index bits does a fully associative cache use? Explain why.

[2 marks]

[Total 20 marks]

Part TWO

True/False/Don't Know

This part of the midterm consists of numbered topics, each of which is followed by five statements lettered A through E. For each such lettered statement, indicate whether that statement is True ("T"), False ("F"), or if you Don't Know ("D"). For example, for topic number 1, you should format your answers in the following way (assuming you chose to answer "Don't Know" for every statement; you will not want to do so in your actual answers document!):

1.
   A. D
   B. D
   C. D
   D. D
   E. D

DO NOT GUESS. You will be awarded one mark for each True or False statement marked as such, but you will lose one mark for each statement you incorrectly identify as True or False. You will neither gain nor lose a mark for each statement you mark "Don't know." Statements left blank, marked illegibly or ambiguously in the judgement of course staff, or having more than one choice marked will be counted as "Don't Know."

The total mark for this part of the exam will be scaled linearly into [0, 40].

1. **Dynamic Memory Allocation in C with `malloc()` and `free()`**

   Consider the behavior and underlying implementations of `malloc()` and `free()` in C in the Linux/UNIX programming environment on x86-64 CPUs.

   A. A debugging `malloc()` implementation that places reserved (known) values before and after each region of allocated memory cannot detect out-of-bounds memory writes at the time of an instruction that writes out of bounds—detection may occur only during subsequent invocations of a memory management routine, such as a `malloc()` or `free()`.

   B. Boundary tags for fast coalescing of blocks during calls to `free()` are fundamentally incompatible with an explicit free list implementation of dynamic memory allocation.

   C. When using boundary tags in a dynamic memory allocation implementation, free blocks need not include a footer.

   D. Consider a first-fit dynamic memory allocation system that employs block coalescing, in which there have been M total successful invocations of `malloc()` and F total successful invocations of `free()`, where $F = M/2$. All invocations are with valid arguments (e.g., all calls to `free()` give an argument that is the address of a currently allocated block). Assume there are no invocations of any other dynamic memory allocation functions. The number of free blocks after this sequence of operations must be greater than 2.

E. When there are many blocks already allocated in the dynamic memory allocation system, in the average case, making a new allocation using a first-fit, explicit free list implementation of `malloc()` will outperform doing so using a first-fit, implicit free list implementation of `malloc()`.

2. **Storage and The Memory Hierarchy**

Consider secondary storage and the memory hierarchy as discussed in COMP0019.

A. Suppose that a read from a spinning magnetic disk requires a seek. Average-case transfer time from the disk's surface dominates average-case seek time on modern 7200 rpm spinning magnetic disks.

B. Suppose that applications have previously written data to all pages within a single block on a solid state disk (SSD). Later an application requests a write to a single page within that block. The flash translation layer of the SSD can accomplish this application-level request by making just one write of a single page to the flash physical medium within the SSD.

C. Loops exhibit better spatial and temporal locality for instruction fetches than long sequences of instructions without loops.

D. Recall that one may view the CPU registers as a still faster, smaller cache with respect to the CPU's level 1 data cache. For a C program, the C compiler determines the replacement policy for the CPU registers.

E. On a modern SSD, writing repeatedly to the same logical block address will cause repeated erases of the same block on the SSD's physical medium, such that the SSD will wear out extremely rapidly (given that a single block can only be erased 100,000 times).

3. **C Data Structures on x86-64**

Consider the following C `structs`, and their representation in main memory on an x86-64 machine:

```
struct bar {
  char arr[5];
};

struct foo {
    struct foo *p;
    struct bar x[3];
    struct bar *q;
};
```

A. The C expression `sizeof(struct bar)` evaluates to 8.

B. The C expression `sizeof(struct bar *)` evaluates to 5.

C. The C expression `sizeof(struct foo)` evaluates to 32.

D. Given the above definitions, no other ordering of the same three elements within `struct foo` will arrive at a different value for the C expression `sizeof(struct foo)` than the value for the ordering in the definition of `struct foo` above.

E. Suppose a C program that includes the above declarations later includes a statement within a function declaration as follows:

```
struct foo *f = NULL;
```

Thereafter within that function, the C expression `sizeof(*f)` will yield the same result as the C expression `sizeof (struct foo)`.

4. **LZW Compression and CW3**

Consider 0019 CW3 and the LZW compressor and decompressor you implemented in CW3.

A. If the LZW compressor and decompressor are implemented correctly, when the LZW decompressor reads a code from the compressed input, it will always find the string in that code's table entry already defined.

B. A correct LZW compressor for CW3 may in at least one case produce an output compressed data file of equal length in bytes to the length of the non-compressed input data file.

C. A correct LZW compressor for CW3 may in at least one case produce an output compressed data file of greater length in bytes than the length of the non-compressed input data file.

D. When there are repeated patterns in its uncompressed input, the LZW compression algorithm is less effective at compressing data at the very start of its input than compressing data later in its input.

E. If one represents the four possible DNA bases in CW3 as `chars` with numerical values 0, 1, 2, and 3, one may build a correct LZW compressor implementation by storing strings of bases directly as standard C language strings (i.e., where a string is a series of `chars` in a standard C string, using the aforementioned encoding of bases into numerical values.

5. **CW2**

Consider the debugging memory allocator you built for 0019 CW2, running on x86-64 under Linux.

A. Suppose an application using a spec-compliant 0019 CW2 debugging `malloc()` implementation (with no further extensions beyond the spec) allocates two heap regions, then does bad pointer arithmetic on an address within the first region, which produces an address that falls within the second region. The application then uses this resulting address to write entirely within the bounds of the second region. The spec-compliant CW debugging `malloc()` implementation will detect this programming error in the application.

B. Because the size argument to `malloc()` is of type `size_t`, when the application program invokes your CW2 debugging `malloc()`, there is no risk to program correctness if there is an arithmetic overflow in the expression to compute the size argument in that application call to your debugging `malloc()`.

C. Calling `free()` on the address of a pointer pointing into the current procedure's stack frame will cause a correct 0019 CW debugging `malloc()` implementation to generate an error message for the programmer at run-time.

D. Calling `free()` with an argument of NULL is undefined behavior.

E. Without a debugging `malloc()` implementation like that in CW2, it's undefined behavior to invoke the system's standard `malloc()` with an enormous request size, such as in `malloc((size_t) -1)`.

[Total for Part TWO: 40 marks]

END OF PAPER