CS 449 Fall 2019 – Final Exam

Please read through the entire examination first!

- You have **110 minutes** for this exam. Pace your time wisely.
- The questions are not necessarily in order of difficulty. Skip around. Make sure you get to all the questions.
- Understand a question before you start writing. Write down thoughts and intermediate steps so you can get partial credit. But clearly indicate what your final answer is.
- The exam is CLOSED book and CLOSED notes (no summary sheets, no calculators, no mobile phones).
- **The exam is printed double-sided.**

The exam has **7 pages** with **5 problems** for a total of **73 points**. The point value of each problem is indicated in the table below. **Please write your answer neatly in the spaces provided**.

Please *do not ask or provide anything* to anyone else in the class during the exam. Make sure to ask clarification questions early so that both you and the others may benefit as much as possible from the answers.

Good Luck! And Relax. **You are here to learn!**

Last Name: Peiffer

First Name: Avery

PeopleSoft ID: 4042056 aep65

All work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS 449 who haven't taken it yet. Violation of these terms could result in a failing grade.

Avery Peiffer

(Please sign)

Problem	Topic	Max Score
1	C Programming	16
2	Processes	16
3	Caches	13
4	Virtual Memory	12
5	Memory Allocation	16
TOTAL		**73**

1. C Programming [16 pts]

For each of the following functions there are comments about what certain lines are meant to do. Mark any lines whose contents DO NOT accomplish what the comment asks it to do. **Important: Please justify your mark explaining (next to the code section) why there is an error.** If everything functions as described, only mark "no errors" at the end of the code section.

Note the comment ABOVE describes the line(s) BELOW. We have also provided a comment about what the whole function is meant to do, but that should not be necessary to complete this question (although you may find it helpful when trying to understand the code). For this section you may assume that the file contains all necessary includes, that all calls to malloc succeed, and that all input arguments to the functions are valid.

```
      1.
      /* Function that takes in an array of integers, mallocs space for a new array,
      * and copies the integers from the first array into the new array. It returns
      * the new array.*/
      int* copy_ints(int* arr) {
         /* Allocates space to store all integers in arr. */
(X)      int* new = malloc(sizeof(arr)); Not casting to (int *)
         /* Iterates over all the elements in arr. */
(X)      for (int i = 0; i < sizeof(arr); i++) { Stop term is size of arr, not num of elements
            /* Loads an element from arr and stores it in new. */
( )         *(new + i) = *(arr + i);
         }
         /* Returns a pointer that can be dereferenced in other functions. */
( )      return new;
       }
( )   no errors
```

```
      2.
      /* Function that takes in an integer, interprets it as a boolean value,
      * and returns a string that can be dereferenced outside the function
      * indicating if it was true or false.*/
      char* bool_to_string (int i) {
            /* Allocates space for a pointer. */
(X)         char* ret_val; No space allocated for actual string
            /* Evaluates to true on all false values and false on all true values. */
(X)         if (i == 0) {
                  ret_val = "false"; Not dereferencing
            } else {
                  ret_val = "true";
            }
            /* Returns a pointer that can be dereferenced in other functions. */
( )         return ret_val;
      }
( )   no errors
```

Avery Peiffer
4042056

3.

```
/* Function that takes in a non-null-terminated string and its length and
* returns a pointer to a malloc'd, null-terminated copy of the string.*/
char* null_term(char* str, unsigned int len) {
    /* Allocates space to store a null-terminated version of str. */
(X)     char* copy = (char*) malloc(sizeof(char) * len); Should be len + 1 to account for null-terminator character
    /* Iterates over all the elements of str. */
( )     for (int i = 0; i < len; i++) {
        /* Loads an element from str and stores it in copy. */
(X)         copy[i] = *(str++); should be *str++ - dereference, then increment pointer
    }
    /* Appends a null terminator to the end of copy. */
(X)     copy[len] = '\0'; Not part of allocated string, happens because of above
    /* Returns the string, that can be dereferenced in other functions. */
(X)     return &copy; return copy
}
( ) no errors
```

4.

```
/* Function that takes in the start of a linked list, mallocs space for a
* new element, appends that element to the front, and returns the new start
* of a linked list.*/
typedef struct int_node {
    int value;
    struct int_node* next;
} int_node_t;
int_node_t* append_front(int_node_t* current, int value) {
    /* Allocates space for a new int node. */
(X)     int_node_t* new = malloc(sizeof(int_node_t*)); Not casting to (int_node_t*)
    /* Assigns the value field to the value parameter. */
( )     new->value = value;
    /* Assigns the next field to the current front. */
( )     (*new).next = current;
    /* Returns a pointer that can be dereferenced in other functions. */
( )     return new;
}
(/) no errors
```

2. Processes [16 pts]

(A) The following function prints out *three numbers*. In the following blanks, **list three possible outcomes**: [6 pts]

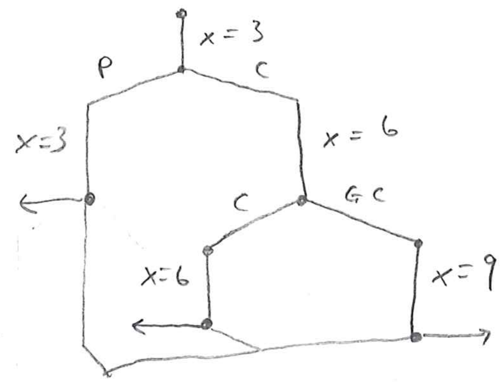
```
void concurrent(void) {
    int x = 3, status;
    if (fork() == 0) {
        x += 3;
        if (fork() == 0) {
            x += 3;
        } else {
            wait(&status);
        }
        printf("%d ",x);
        exit(0);
    }
    printf("%d ",x);
    wait(&status);
}
```

child waits for grandchild; 9 comes before 6

(1) 3 9 6

(2) 9 6 3

(3) 9 3 6



(B) In the code above, we will refer to the 3 processes as the "parent," "child," and "grandchild." Who cleans up the grandchild process? **Mark the true statement** below. [2 pts]

wait(& status)

Reaped by ~~parent~~	Reaped by child	Reaped by init/systemd	Must be manually killed
____	X	____	____

(C) In the following blanks, write "**Y**" for yes or "**N**" for no if the following need to be updated *during* a **context switch**. [4 pt]   Registers

Stack N   %rsp Y   PTBR Y   %rip Y

(D) A student wants to write a concurrent algorithm using `fork()` where all processes **write to different parts of the same array in the heap**. Will this work or not? Explain *briefly*. [4 pt]

Mark one: Yes X

No ____

Explanation: The processes can share the address space of the array and separately write to it.

Avery Peiffer
4042056

3. Caches [13 pts]

In a typical cache, the most significant bits of an address make up the tag, the next bits make up the index, and the least significant bits make up the offset. Call this the TIO scheme.

(A) Consider the execution of the following function on a direct mapped cache of total size 128 bytes with 16 sets and a block size of 8 bytes. Assume that `i` and `array` (the pointer) are stored in registers, array is 8-byte aligned, and no errors will occur during execution.

$\frac{128}{8} = \frac{2^7}{2^3} = 16$ blocks

```
void loopy(int array[32]) {
    int i;
    for (i = 0; i < 32; i++)
        array[i] += 3;
    for (i = 0; i < 32; i++)
        array[i] -= 3;
}
```

16 Blocks at 8 bytes each

Miss on every other, pull in 2 ints.

Go to 128

16 misses on first

00 ✗ 04 ✓

08 ✗ 0C ✓

⋮

The next 2 questions ask about a single execution of loopy, starting with an empty cache, under the TIO scheme. Phrase your answer as a number of misses followed by a description of the miss kind, in order of occurrence. Example: "4 compulsory misses, then 10 conflict misses." Hint: there are no capacity misses.

(2 points) How many misses did the first loop take? What kinds were they?

\# of misses = 16 Compulsory (accessing each block for the first time)

Everything in cache after this point

(2 points) How many misses did the second loop take? What kinds were they?

\# of misses = 0

(B) For each of the following sequences of memory accesses, determine the best cache parameters to reduce miss rate. Some parameters will be given, so fill in the remaining one. Assume that the cache is empty at the beginning of each sequence, the cache uses an LRU replacement policy, and that all accesses are valid.

(3 points) `0x00, 0x08, 0x10, 0x18, 0x20, 0x28, 0x30, 0x38`

Total cache size = 32 bytes Associativity = 1 Block size = 32 bytes

16 bytes, miss 4 times (50%)

8 bytes, miss 8 times

32 bytes, miss 2 times (25%)

(3 points) `0x00, 0x20, 0x40, 0x80, 0x01, 0x21, 0x41, 0x81`

Total cache size = 32 bytes Block size = 8 bytes Associativity = 4

All will map to same location, so want them to all be stored

(3 points) `0x00, 0x01, 0x38, 0x48, 0x00, 0x01`

Total cache size = 16 bytes Block size = 8 bytes Associativity = 1

o = 3

$\underbrace{00}\underbrace{111000}$

Associativity = 2 → 3 misses (50%)

4. Virtual Memory [12 pts]

Consider the following system:

- 16-bit virtual addresses, 10-bit physical addresses.
- A page size of 16 bytes. o = log₂ (PS) = log₂ (16) = 4
- 2-way set associative TLB with 8 total entries. 4 indices → 2 bits
- All page table entries NOT in the initial TLB start as invalid.

(4 points) Compute the following quantities:

Page offset bits: 4 VPN bits: 12

PPN bits: 6 TLB index bits: 2

(6 points) The TLB has the following state:

Set	Tag	PPN	Valid	Tag	PPN	Valid
0	0x1B2	–	0	0x283	0x3A	1
1	0x2FB	0x29	1	0x0E8	0x1D	1
2	0x004	–	0	0x346	–	0
3	0x3F4	0x1B	1	0x257	0x36	1

Fill in the associated information for the following accesses to virtual addresses. (enter N/A if the answer cannot be determined):

Virtual Address	TLB Hit?	Page Fault?	PPN
0 E 8 0011 1010 0001 0111 0x3A17	Yes	No	0x1D - 0001 1101
0 0 4 0000 0001 0010 0011 0x0123	No	N/A	N/A

(2 points) We've said in lecture that pages are analogous to cache blocks (i.e., they are the unit of data transfer between memory and disk). Why are pages significantly larger than cache blocks?

They have to index a much larger space than cache blocks.

Avery Peiffer
404 2056

5. Memory Allocation [16 pts]

(A) (4 points) Below is the current state of the heap after the following sequence of allocations and frees:

A allocated, **B** allocated, **C** allocated, **B** freed, **D** allocated

Start | End

A	**A**	**A**	(B)	(.B)	**C**	**D**	**D**		

Which allocation strategy was used?

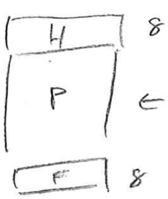
$\boxed{\text{Next Fit Placement}}$

(B) (6 points) We are designing a dynamic memory allocator for a **64-bit computer** with **8-byte boundary tags** and **alignment size of 16 bytes** using an **explicit free list**. Assume a footer is always used. Answer the following questions:

Maximum tags we can fit into the header (not counting `size`): 2 tags

4 bytes per tag

Minimum block size: 32 bytes



Payload must be aligned to 16 bytes

Minimal amount of *internal fragmentation* in block allocated by `malloc(7)` request: 25 bytes

(C) (6 points) Consider the C code shown here. Assume that the `malloc` call succeeds and that `foo` and `str` are stored in memory (not registers). In the following groups of expressions, **MARK the one** whose returned *value* (assume just before `return 0`) is the **lowest/smallest**. Hint: Recall how C program code and data are laid out in virtual memory segments.

```
#include <stdlib.h>
int ZERO = 0;

int main(int argc, char *argv[]) {
    char *str = "cs449";
    int *foo = malloc(8);
    return 0;
}
```

Group 1:	( ) &foo	( ) &str	(X) ZERO
Group 2:	( ) &foo	( ) &main	(X) &ZERO
Group 3:	( ) foo	( ) str	(X) &ZERO

Comparing against 0

Comparing to actual vals