Final Exam  Fall 2025

This exam consists of eight questions. You have 180 minutes – budget your time wisely. Assume the ArmLab/gcc217 environment unless otherwise stated in a problem.

Do all of your work on these pages. You may use the provided blank pages at the end for scratch space, however this exam is preprocessed by computer, so for your final answers to be scored you must write them inside the designated spaces and fill in selected circles and boxes completely (⚫ and ⬛, not ✔ or ✘). Please make text answers dark and neat.

Name: Sample Solutions NetID:

Precept:

- ◯ P01 - MW 1:20 Christopher Moretti
- ◯ P02 - MW 3:30 Amelia Dobis
- ◯ P03 - TTh 12:15 Kevin Alarcón Negy
- ◯ P04 - TTh 12:15 Andrew Johnson
- ◯ P05 - TTh 12:15 Nicholas Yap
- ◯ P07 - TTh 1:20 Lana Glisic
- ◯ P08 TTh 3:30 Kevin Alarcón Negy

This is a closed-book, closed-note exam, except you are allowed one two-sided study sheet. 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, smartwatches except to check the time, etc. may not be used during this exam.

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.

In the box below, copy **and** sign the Honor Code pledge before turning in your exam:
"I pledge my honor that I have not violated the Honor Code during this examination."

Exam statistics:
Min: 37/96
Mean: 73.7/96 Standard Deviation: 11.1/96
Median: 75/96
Max: 94/96

X_____

Question 1: Building and Running Programs 8 points

You are given the following five files for a simple vector library.

vector.h

```
typedef struct vector* vector_T;
```

vector.c

```
#include "vector.h"

struct vector {
    double *elements;
    int length;
};

/* some more code */
```

vector_math.h

```
#include “vector.h”

double max(vector_T v);
double min(vector_T v);
vector_T add(vector_T v1, vector_T
v2);
vector_T dot(vector_T v1, vector_T
v2);
```

vector_math.c

```
#include "vector_math.h"

/* some more code */
```

client.c

```
#include "vector.h"
#include "vector_math.h"

/* some more code */
```

a. (2 points) You are also given a `Makefile` with the following entry. Does this entry need fixing? If so, fix it in-place in the box below by adding, striking out, or keeping whatever you need. If not, write “OK” in the box. You can assume that tabs and spacing are correct.

```
client: client.o vector.o vector_math.o
     gcc217 client.o vector.o vector_math.o ~~vector.h vector_math.h~~ -o client
```

b. (2 points) Answer the same question as in part a., but for the following `Makefile` entry in the box below.

```
vector_math.o: vector_math.c vector_math.h vector.h
     gcc217 vector_math.c -c
```

c. (2 points) Running `make` to build the `client` executable results in the following message:

```
Warning: redefinition of typedef 'vector_T'.
```

Which file would you change to not have this message show up next time `client` is built, and what changes would you make to that file. Describe in English, not code, in the box below.

Filename: vector.h

Add a guard against multiple inclusion around the typedef .

d. (1 point) Describe, in a sentence, the meaning of using "partial builds," e.g., in `make`.

The term "partial builds" refers to:

building and re-building individual files as needed rather than the entire pipeline of builds every time.

e. (1 point) State, in a sentence, one important benefit of partial builds.

One important benefit of partial builds is:

Example correct responses:
- Decreases overhead when rebuilding only files that changed.
- Allows parallelization of building files.
- May help debugging by isolating specific problematic files.

Question 2: Number Systems

9 points

Let us consider numbers in base-32. We extend the notation used in hexadecimal (base 16). That is, after 0-9 and a-f, we use the letters g through v to represent the base-32 digits with decimal values 16 through 31.

a. (2 points) Convert the base-32 number $5g_{32}$ to hexadecimal. Show your work in the open space below on the left and write your hexadecimal answer in the box.

There were many ways to do this. Perhaps the simplest was identifying that a base-32 symbol can be represented in 5 bits, compared with 4 bits for a base-16 hexit, so one can convert to binary and then re-group by 4s instead of 5s: 00101 10000 = 1011 0000 = b0

b0

b. (2 points) Perform the base-32 subtraction of two base-32 numbers, 217_{32} minus 126_{32}, and convert the result to decimal. Show your work in the open space below on the left and write your decimal answer in the box.

The base-32 subtraction yields the result v1. This is 31*32 + 1*1.

993

c. (4 points) What is the maximum value of a C `signed char` (assuming 8-bit two's complement representation) and of a C `unsigned char`, both represented in base-32? Show your work in the open space below on the left and write your base-32 answers in the boxes.

The maximum 8-bit two's complement signed value is:
011 11111 = 3v.

The maximum 8-bit unsigned value is:
111 11111 = 7v.

`signed`	`unsigned`
3v	7v

d. (1 points) In one sentence, state one drawback of using one's complement instead of two's complement to represent signed integers in a computer.

Example correct responses:
- Ones' complement has two representations of zero
- Ones' complement has different arithmetic algorithms for signed and unsigned numbers.
- Ones' complement can represent (exactly 1) fewer distinct integers than 2's complement.

Question 3: Modularity and Scope 14 points

Consider the following incomplete scaffolding for an English language dictionary program:

```
int iMaxWords = 500;
static int iEditionYear = 1987;
extern char *pcNewDefinition;

struct dictionaryEntry {
    char *pcWord;
    char *pcDefinition;
    int iIsNoun;
};
struct dictionaryEntry sEntry1;

static void setToNoun(struct dictionaryEntry *psEntry){
    /* some code here */
}

int main(void){
    static char *pcMyWord;
    /* some code here */
    struct dictionaryEntry sEntry2 = {pcMyWord, "", 0};
    /* some code here */
}
```

Complete the table below to list the scope (“FILE” or “BLOCK”), linkage (“INTERNAL” or “EXTERNAL”), and duration (“PROCESS” or “TEMPORARY”) of each variable and the section of memory in which it resides (assume `gcc217` compilation). If the memory section of some variable cannot be determined without additional information, write “UNKNOWN.”

	Scope	Linkage	Duration	Memory Section
`iMaxWords`	FILE	EXTERNAL	PROCESS	DATA
`iEditionYear`	FILE	INTERNAL	PROCESS	DATA
`pcNewDefinition`	FILE	EXTERNAL	PROCESS	UNKNOWN
`sEntry1`	FILE	EXTERNAL	PROCESS	BSS
`pcMyWord`	BLOCK	INTERNAL	PROCESS	BSS
`sEntry2`	BLOCK	INTERNAL	TEMPORARY	STACK
`setToNoun()`		INTERNAL		TEXT
`main()`		EXTERNAL		TEXT

Question 4: Modularity

11 points

We have seen the ASCII representation for character encodings. In ASCII, the set of lowercase characters `a-z` and the set of uppercase characters `A-Z` are each contiguous. The set of uppercase letters comes before the set of lowercase letters, but there are other non-letter characters between these two sets.

a. (5 points) In the box below, implement the common functions `is_upper()` and `to_upper()` for characters, assuming the ASCII representation and the following interface.

```
/* Takes in a valid character and returns 1 if it is an uppercase letter,
and 0 if not. */
int is_upper(char c);

/* Takes in a valid character (lowercase or uppercase letter) and returns it
in uppercase form. */
char to_upper(char c);
```

```
int is_upper(char c) {

    if ( 'A' <= c && c <= 'Z' )
       return 1;
    return 0;



}

char to_upper(char c) {

    if (is_upper(c))
       return c;
    return c - ('a' - 'A'); /* equivalent to (c + 'A' - 'a') */



}
```

Now imagine a new character representation called LIR (Letter Interleaved Representation) in which the letters of the upper and lowercase alphabets are interleaved as follows: `{'A', 'a', 'B', 'b', 'C', 'c', 'D','d', 'E', 'e', ...}`.

b. (2 points) How would you change the interface to adapt to this character representation?

A correct answer will say “You wouldn’t!”
because an interface shouldn't change based
on a new representation or implementation.

c. (4 points) Implement `is_upper()` and `to_upper()` for the LIR implementation, assuming that `'A'` is at an odd index in the encoding.

```
int is_upper(char c) {

    if(c % 2 == 1)
      return 1;
    return 0;


}

char to_upper(char c) {

   if (is_upper(c))
     return c;
   return c - 1;


}
```

Question 5: Dynamic Memory, Defensive Programming 16 points

Consider the following `String.h` interface file for a new string data type.

```
typedef struct String {
  char *text;      /* String_T owns its string contents: defensive copy */
  size_t phys_len; /* number of bytes allocated for text */
} *String_T;

/* Returns a String_T representing the contents of C string user_string,
including the nullbyte. If unable to allocate memory, returns NULL. */
String_T new_String(char *user_string);

/* Returns the String_T object s after expanding the capacity of its
referenced character array to len. If expansion fails, returns unmodified
object s. */
String_T expand_String(String_T s, size_t len);

/* Frees all memory allocated for String_T object s */
void free_String(String_T s);

/* there may be additional interface functions declared but not shown */
```

Write the code that implements this interface in the boxes below. Include all appropriate parameter validation. Assume all necessary header files have been included, including the C standard library's `string.h`.

a. (6 points)

```
String_T new_String(char *user_string){
    char *defensive_cpy;
    String_T my_str;
    size_t len;

    assert(user_string != NULL);

    my_str = malloc(sizeof(struct String));
    if (my_str == NULL)
        return NULL;
    len = strlen(user_string) + 1;
    defensive_cpy = malloc(len /*optional: * sizeof(char) */);
    if (defensive_cpy == NULL){
        free(my_str);
        return NULL;
    }
    strcpy(defensive_cpy, user_string);
    my_str->len = len;
    my_str->text = defensive_cpy;
    return my_str;
}
```

b. (5 points)

```
String_T expand_String(String_T s, size_t len){

        /* realloc version */
        char * new_string;
        assert (s != NULL);
        /* optional: assert(len >= s->len); */
        new_string = realloc(s->text, len);
        if(new_string == NULL)
            return s;

        s->text = new_string;
        s->phys_len = len;
        return s;

        /* new string allocation version */
        char * new_string;
        assert (s != NULL);
        /* optional: assert(len >= s->len); */
        new_string = malloc(len);
        if(new_string == NULL)
            return s;
        strcpy(new_string, s->text);
        free(s->text);
        s->text = new_string;
        s->phys_len = len;
        return s;

}
```

c. (2 points)

```
void free_String(String_T s){

    assert (s != NULL);
    free(s->text);
    free(s);


}
```

d. (2 points) Does the `String.h` interface shown above represent an abstract data type (ADT)? If so, say what about the interface most makes it so. If not, say what main types of changes you would make to have it represent an ADT.

The String data type is NOT an ADT.

Example correct suggested changes:
- String struct definition should be in the .c file, not the interface where it is now.
- Function comment of expand_String should not reveal that object has a referenced character array.

Question 6: Extensibility in ADTs

11 points

We talked in class about how ADT interfaces expose a set of functions but can also allow a client to provide a function (or functions) that can be applied to the data type. Consider the following program that utilizes that technique:

```
#include <stdio.h>
typedef /*redacted*/;

int add(int a, int b) {
    return a + b;
}

int sub(int a, int b) {
    return a - b;
}

int mul(int a, int b) {
    return a * b;
}

int div(int a, int b) {
   return a / b;
}

int apply(mathOp f, int x, int y) {
    return f(x, y);
}

int compose(mathOp f, int x, int y, mathOp g, int z) {
    return g(f(x, y), z);
}

int main(void) {
    mathOp ops[4];
    int i;
    ops[0] = add;
    ops[1] = sub;
    ops[2] = mul;
    ops[3] = div;
    for (i = 0; i < 3; i++) {
        printf("%d\t", apply(ops[i], 2, 17));
        printf("%d\n", compose(ops[i], 2, 17, ops[i+1], 2));
    }
    return 0;
}
```

a. (2 points) In the box below, write out the `typedef` that was redacted from the code above (see the comment "`/* redacted */`" on the second line of the code) for the type `mathOp` that will make this program pass type checking.

```
typedef int (*mathOp)(int, int);
```

b. (3 points) Describe, concisely in English, the type of the variable `ops`. Don't refer to the type alias `mathOp`, but rather to underlying C types only.

ops is an array of (four) function pointers to functions that
each take two int parameters and return an int

c. (6 points) Carefully trace through the provided program and write, in the box below, its output as it would appear in `stdout`. (Reminder: `\t` is the horizontal tab character specifier.)

```
19    17
-15   -30
34    17
```

You may refer to this abbreviated ARM assembly language reference for Q7.

Instruction(s)	Description
`{add,sub,mul,sdiv,udiv} d, s1, s2`	`d = s1 {+, -, *, /, / } s2`
`{lsl, lsr} d, s1, imm`	`d = s1 {<<, >>} imm`
`{b, bgt, bl} label`	{Go to, Go to if greater , Call function at} `label`
`cmp first, second`	Compare `first` with `second`, setting bits in PSTATE
`ldr dst, [src]`	Load 4 or 8 bytes pointed to by `src` into `dst`
`str src, [dst]`	Store 4 or 8 bytes in `src` to memory pointed to by `dst`
`mov dst, src`	Copy contents of register `src` to register `dst`
`.equ NAME, value`	`NAME` is a symbolic constant for `value`
`NAME .req r`	`NAME` is a register alias for `r`

Question 7: Assembly Language 19 points

Consider a function that calculates the value of N choose R (in how many ways can R items be chosen out of N), written in assembly language. The method used is to calculate the number of permutations of R items from N and divide the result by the factorial of R. The assembly language code for this function, `long combinations(long lN, long lR)`, is shown below.

a. (8 points) For each blank comment box in the assembly code below, write a **single line of flattened C code** that corresponds to the assembly language instructions that follow that box (until the next box). Use the same variable names in your flattened C comment as are used in the assembly code.

```
    .equ COMBINATIONS_STACK_BYTECOUNT, 48
    .equ LN, 8
    .equ LR, 16
    .equ LPERM, 24
    .equ LRFACT, 32
    .equ I, 40

    .global combinations
combinations:
    // Prolog
    sub     sp, sp, COMBINATIONS_STACK_BYTECOUNT
    str     x30, [sp]
    str     x0, [sp, LN]
    str     x1, [sp, LR]

    // LPERM = permutations(LN, LR);
    ldr     x0, [sp, LN]
    ldr     x1, [sp, LR]
    bl      permutations
    str     x0, [sp, LPERM]

    // LRFACT = 1;
    mov     x0, #1
    str     x0, [sp, LRFACT]

    // I = 2;
    mov     x0, #2
    str     x0, [sp, I]

loop:
    // if (I > LR) goto endloop;
    ldr     x0, [sp, I]
    ldr     x1, [sp, LR]
    cmp     x0, x1
    bgt     endloop
```

```
// LRFACT *= I;
ldr     x0, [sp, LRFACT]
ldr     x1, [sp, I]
mul     x0, x0, x1
str     x0, [sp, LRFACT]

// I++;
ldr     x0, [sp, I]
add     x0, x0, #1
str     x0, [sp, I]

// goto loop;
b       loop

endloop:
// return LPERM / LRFACT;
ldr     x0, [sp, LPERM]
ldr     x1, [sp, LRFACT]
sdiv    x0, x0, x1
// Epilog
ldr     x30, [sp]
add     sp, sp, COMBINATIONS_STACK_BYTECOUNT
ret
```

b. (2 points) Observe in the code above that the loop control variable `I` is loaded from memory three separate times within the loop, as well as being stored in each iteration. How could you minimize these repeated memory references, without changing the correctness of the `combinations` function? Describe in one or two sentences (not code) in the box below.

A correct answer should mention using registers to store intermediate I values instead of storing and loading from stack repeatedly.

c. (1 point) Now suppose there is a function call made within the loop. How, if at all, would your answer from part b. change?

A correct answer should specify that callee-saved registers should be used.

d. (2 points) Which one ARMv8 register must a function always save if it, itself, is to make a function call? If that register were not saved, what would happen to the program's control flow?

A correct answer must mention x30 and indicate that
not storing x30 will cause an infinite self-recursion
(i.e., a function will return to the return point of its most recent function call).

e. (6 points) C provides the remainder operator (`%`). Using only a basic set of ARM assembly instructions, recreate the functionality of the remainder operator by filling in the following assembly language function. Acceptable instructions are the 3 parameter versions of:
`ADD`, `ADDS`, `SUB`, `SUBS`, `MUL`, `UDIV`, `LSL`, and `LSR`

You need only write code between the indicative comments shown. Maintain the dividend and divisor in caller-saved registers – indicate which registers you are using by completing the req directives, and then use these aliases in your assembly code.

```
// Takes in a positive integer dividend as the first argument and a
// positive integer divisor as the second argument and returns the
// remainder of dividing dividend/divisor.
   .global remainder
   .equ REMAINDER_STACK_BYTECOUNT, 16
remainder:
   sub sp, sp, REMAINDER_STACK_BYTECOUNT
   str x30, [sp]

// Write your code starting here
   DIVIDEND   .req w0
   DIVISOR    .req w1

   udiv w2, DIVIDEND, DIVISOR // w2 is an arbitrary choice among caller-saved registers

   mul w2, w2, DIVISOR

   sub w0, DIVIDEND, w2









// End your code before here
   ldr x30, [sp]
   add sp, sp, REMAINDER_STACK_BYTECOUNT
   ret
```

Question 8: The Process and Virtual Memory Abstractions 8 points

a. (2 points) We write programs using a core abstraction that allows us to reason algorithmically without having to worry too much about the details of the systems on which our program runs. This abstraction is called a process. In lecture, we discussed two key illusions that the process abstraction provides: list either in the box below.

The two intended answers were:
- The process always is in control of CPU.
- The process always owns all system resources (including memory).

We also accepted the abstraction of Virtual memory / address space.

b. (1 points) On Armlab, virtual addresses are 64 bits wide. What, if anything, does this tell us about how much physical memory each Armlab machine has?

The correct answer was to say “Nothing!”.
We accepted discussion that with today's physical memory sizes, we know that physical memory in armlab will be less than the virtual memory address space of 2^64 bytes.

c. (2 points) On Armlab, we also know a given virtual page contains 2^{16} bytes. Given this information, extract the virtual **page number** from the following hexadecimal address.

`0x0123456789ABCDEF`

Show your work in the open space below on the left and write your answer in the box on the right.

A virtual address is 64 bits, with the page number followed by the page offset. If a page is 2^16 bytes, then 16 bits are used for the page offset.
Thus, 64 - 16 = 48 bits are left for the page number.
This corresponds to the leftmost 12 hexits in the number above (the leading 0 is omitted in the answer in the box).

0x123456789AB

d. (3 points) Assuming the size of a physical page is the same as that of a virtual page, and that the Armlab processor our program is running on has 64GB of physical main memory, how many bits are used for the physical page number in Armlab? Show your work in the open space below on the left and write your answer, in bits, in the box on the right.

64 GB of physical memory = 2^36 bytes of memory, and thus 36 bits in a physical address. 16 are used for the page offset, so 36 - 16 = 20 bits are used for the physical page number.

20

Question 8 was the last question on the exam. This page is intentionally left blank.
You may use it for scratch work, but any answers given below will not be graded.