Final Exam  Fall 2022

This exam consists of 8 questions. You have 180 minutes – budget your time wisely. Assume the ArmLab/Linux/gcc217 environment unless otherwise stated in a problem.

Do all of your work on these pages. You may use the provided blank spaces 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 1:30 Xiaoyan Li	◯ P04 TTh 1:30 Qingchen Dang	◯ P07 TTh 3:30 David Xu
◯ P02 - MW 3:30 Xiaoyan Li	◯ P05 TTh 1:30 Donna Gabai	◯ P08 TTh 3:30 Donna Gabai
◯ P03 - TTh 12:30 Maxine Perroni-Scharf	◯ P06 TTh 2:30 Jihoon Chung	◯ P09 TTh 7:30 William Yang

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, music players, 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:
Mean: 63.3 / 85 (74.5%)
StdDev: 9.88 / 85
25th %ile: 58 / 85
50th %ile (Median): 65 / 85
75th %ile: 70 / 85
Max: 83 / 85

(The exam questions begin on page 3. This page may be used for scratch work, however any answers given on this page will not be graded.)

Question 1: Eh? Oh! Aw DraT! 9 points

For each statement below, identify whether it typically applies to an *abstract object* (**AO**), *abstract data type* (**ADT**), *both*, or *neither*. Fill in exactly one circle per line.

	AO	ADT	Both	Neither
a. Interface makes representation visible to the client:	○	○	○	●
b. Interface defines an opaque pointer type:	○	●	○	○
c. Interface methods do not take instance as argument:	●	○	○	○
d. Implementation uses file-scope variables:	●	○	○	○
e. Client may instantiate multiple instances:	○	●	○	○
f. Implemented in A1's `decomment.c`:	○	○	○	●
g. Defined by A2's `string.h` interface:	○	○	○	●
h. Implemented in A3's `symtablelist.c`:	○	●	○	○
i. Provided for you in A4's Part 2 (DT):	○	○	●	○

Question 2: Tally-ho! What fun! 4 points

Consider this function declaration: `void *funFun(int *, double*);`

We want to pass `funFun` as a parameter to a higher order function (a function that is parameterized by another function), `hoFun`, but `hoFun`'s parameter is a function pointer for a function that takes two generic pointers as parameters and returns a C string.

What is the cast required for `funFun` to be passed as an argument to `hoFun`?

```
(char *(*)(void *, void *)) funFun
```

Question 3: `Makefile` Go "Brrr!" 13 points

Consider this `Makefile`, in which the comments (starting with #) at the end of each of the six commands are line markers to identify the command in this problem.

```
prog: zero.o one.o two.o three.o four.o
     gcc217 -o prog {zero,one,two,three,four}.o   #A

zero.o: zero.c
     gcc217 -c zero.c                             #B

one.o: one.c
     gcc217 -c one.c                              #C

two.o: two.c two.h four.h
     gcc217 -c two.c                              #D

three.o: three.c three.h four.h
     gcc217 -c three.c                            #E

four.o: four.c four.h
     gcc217 -c four.c                             #F
```

Assume that the working directory initially contains all the referenced `.h` and `.c` files, the `Makefile`, and no other files. However, each item *continues* with the resulting contents from previous items. (For example, if item *m.* resulted in the creation of a file `foo.o`, then `foo.o` would still be there during item *n*.)

Recall these Linux details: `touch` changes the date/time stamp of the given file(s) to the current date/time (as if they had just been edited); `rm` removes a file; `t*.o` in item h. means "all `.o` files starting with `t`", and line `A` in the `Makefile` is equivalent to: `gcc217 -o prog zero.o one.o two.o three.o four.o`

For each item on the next page, select which `Makefile` rule command(s) – there may be more than one – `make` executes when it is invoked in this sequence, or mark the **None** box if `make` does not execute any of commands **A** through **F** for that item.

	A	B	C	D	E	F	None
a. `make two.h`	☐	☐	☐	☐	☐	☐	■
b. `make zero.o`	☐	■	☐	☐	☐	☐	☐
c. `make`	■	☐	■	■	■	■	☐
d. `touch zero.c` `make`	■	■	☐	☐	☐	☐	☐
e. `make`	☐	☐	☐	☐	☐	☐	■
f. `touch two.h` `make`	■	☐	☐	■	☐	☐	☐
g. `touch four.h` `make`	■	☐	☐	■	■	■	☐
h. `touch t*.o` `make`	■	☐	☐	☐	☐	☐	☐
i. `rm one.o` `make two.o`	☐	☐	☐	☐	☐	☐	■
j. `make one.o`	☐	☐	■	☐	☐	☐	☐
k. `make`	■	☐	☐	☐	☐	☐	☐

(The exam questions continue on page 7. This page may be used for scratch work for the next problems, however any answers given on this page will not be graded.)

Question 4: Eek! … or is that "Ek0e!"? 6 points

Consider the following code, which uses %hd as a format specifier for a decimal `short`:

```
unsigned char auc[2] = {1, 3};
short s = *(short *) auc;
printf("%hd", s);
```

a. What does this code print on armlab, where a `short` is 2 bytes and little-endian?

769
(0x0301:512+256+1)

b. What does it print on a *big-endian* machine where a `short` is still 2 bytes?

259
(0x0103:256+2+1)

c. What does it print on a *little-endian* machine where a `short` is only 1 byte?

1

Question 5: Encore! 8 points

In Precept 23 and Lecture 24 you saw the machine language representation of the ADR instruction. It was also given to you in `miniassembler.c` in Assignment 6. As a reminder, here are the details of that instruction's machine language representation:

For an instance of the instruction `ADR x1, label2` :

- assume the address of the label `label2` is: `0x217217`
- assume the address of this ADR instruction is: `0x214127`

What is the machine code for this instruction, represented in **hexadecimal**?

0x10018781

Question 6: ¡Ay Ay Ay! 18 points

Consider these three files which are built together into a single program:

file1.c

```
  #include <stdio.h>

  /* in lieu of file2.h */
1 void printI();

2 static int i;

  int main() {
3   printI();
4   printf("%d\n", i);
    /* could contain
       additional code */
    return 0;
  }
```

file2.c

```
  #include <stdio.h>
5 extern int i;

  void printI() {
6   printf("%d", i);
  }
```

file3.c

```
7 int i = 217;
8 extern int i;
```

a. For each underlined line number, indicate whether that line contains a definition, a declaration that is not a definition, or does not contain a declaration at all or results in an error. Fill in exactly one circle per line.

		Definition	Other Declaration	Neither or Error
i.	Line 1 in `file1.c`	○	●	○
ii.	Line 2 in `file1.c`	●	○	○
iii.	Line 3 in `file1.c`	○	○	● (This is a function call.)
iv.	Line 5 in `file2.c`	○	●	○
v.	Line 7 in `file3.c`	●	○	○
vi.	Line 8 in `file3.c`	○	●	○

b. What section of memory contains the `i` referenced on Line 2 in `file1.c`?

BSS
(Uninitialized FS/PD)

c. What section of memory contains the `i` referenced on Line 5 in `file2.c`?

Data
(The i on file3.c Line 7)

d. What section of memory contains the `i` referenced on Line 7 in `file3.c`?

Data
(Initialized FS/PD)

e. What integer is printed by Line 4 in `file1.c`?
If the value is uninitialized garbage, write "???".

0
(BSS starts out at 0)

f. Which `i` is printed by Line 6 in the `printI` function body in `file2.c`?

- ◯ always the one defined in `file1.c`
- ◯ always the one defined in `file2.c`
- ⚫ always the one defined in `file3.c`
- ◯ it depends: the one defined in the file where the function was called from.

g. For each statement, identify to which stage(s) – there may be more than one – of the build process (**P**reprocessor, **C**ompiler, **A**ssembler, **L**inker) the statement applies.

		P	C	A	L
i.	Line 1 in `file1.c` helps this stage do its job	☐	⬛	☐	☐
ii.	Defines labels for each `i` variable	☐	⬛	☐	☐
iii.	Defines offsets for each `i` variable	☐	☐	⬛	☐
iv.	Adds more declarations to the code in `file2.c`	⬛	☐	☐	☐
v.	Is the first to fail without Line 5 in `file2.c`	☐	⬛	☐	☐
vi.	May, in general, produce a finalized machine language branch instruction	☐	☐	⬛	⬛

Question 7: A-“Ha”! Presidents and PMs *are* Different. 12 points

The function `diffFiles` is a client of FT from Assignment 4. This function takes two strings representing file paths in the tree, prints the first difference, if any, in the files' contents and returns TRUE if the contents match exactly or FALSE if the contents do not match *or* if either path is not a file in the tree. The function does not attempt to handle empty files and assumes files' contents are always retrievable and match the size specified when they were inserted. Here is an excerpt of a test client for `diffFiles`:

```
   /* each file size includes the string's trailing '\0' */
   FT_insertFile("leaders/usa/#9","Harrison",9);
   FT_insertFile("leaders/usa/#19","Hayes",6);
   FT_insertFile("leaders/usa/#23","Harrison",9);
   FT_insertFile("leaders/usa/#29","Harding",8);
   FT_insertFile("leaders/usa/#47?","Harris",7);
   FT_insertFile("leaders/uk/#25","Hamilton-Gordon",16);
   FT_insertFile("leaders/can/#22","Harper",7);
   FT_insertDir("leaders/aus/#23"); /* Hawke */

   /* Harper matches Harper */
   assert(diffFiles("leaders/can/#22", "leaders/can/#22") == TRUE);
   /* Harrison matches Harrison */
   assert(diffFiles("leaders/usa/#9", "leaders/usa/#23") == TRUE);
   /* Harrison does not match Harris */
   assert(diffFiles("leaders/usa/#9", "leaders/usa/#47?") == FALSE);
   /* Hayes does not match Hamilton-Gordon */
   assert(diffFiles("leaders/usa/#19", "leaders/uk/#25") == FALSE);
   /* leaders/aus/#23 is a directory, not a file */
   assert(diffFiles("leaders/usa/#29", "leaders/aus/#23") == FALSE);
   /* leaders/ire/Haughey is not a path in the tree */
   assert(diffFiles("leaders/usa/#29", "leaders/ire/Haughey") == FALSE);
```

Unfortunately the lines from the core of this function's implementation have become jumbled, though the skeleton remains:

```
boolean diffFiles(char *pcPath1, char *pcPath2) {
   char *pcText1, *pcText2;
   size_t ulSize1, ulSize2;
   boolean bType1, bType2;
   size_t ulIndex = 0;
   assert(pcPath1 != NULL);
   assert(pcPath2 != NULL);

   /* missing lines here */
}
```

Here are the remaining lines in the function, which have lost their indentation and had any numbers in variables replaced by ?s (so each ? may represent either a 1 or a 2):

```
A while(ulIndex < ulSize? && ulIndex < ulSize?) {
B if(FT_stat(pcPath?, &bType?, &ulSize?) == SUCCESS && bType?) {
C if(pcText?[ulIndex] != pcText?[ulIndex]) {
D pcText? = FT_getFileContents(pcPath?);
E fprintf(stderr, "Files differ at byte %i: [%c,%c]\n",
          (int) ulIndex, pcText?[ulIndex], pcText?[ulIndex]);
F ulIndex++;
G } /* end if */
H } /* end while */
I else return FALSE;
J return FALSE;
K return TRUE;
```

As a reminder, here are abbreviated specifications for the two relevant FT API functions:

```
/* Returns SUCCESS if pcPath exists in the hierarchy.
   When returning SUCCESS, if path is a file:
   sets *pbIsFile to TRUE, and sets *pulSize to the contents' size */
int FT_stat(const char *pcPath, boolean *pbIsFile, size_t *pulSize);

/* Returns the contents of the file with absolute path pcPath.
   Returns NULL if unable to complete the request for any reason. */
void *FT_getFileContents(const char *pcPath);
```

There are 16 lines in total that are missing, so some lines appear more than once.

The first four missing lines set up the file contents from `pcPath1`, if possible. The second four missing lines set up the file contents from `pcPath2`, if possible, and thus will have the exact same set of line markers (letters **A**-**K**) from the list in the box above. In this box, enter the **four** line markers that represent this (repeated) four-line sequence:

BDGI
(BGID/DBGI also work)

The last eight missing lines use the file contents correctly set up in the first eight lines in order to implement diffFiles's specified functionality. Only 1 line from the four lines in the first box is reused in this portion of the code.
In this box, enter the **eight** line markers that represent this final eight-line sequence:

A C E J G F H K

Question 8: Oy! (Where's gdb when you need it?) 15 points

Once you finish this question, you will be done with COS 217 – something that merits a, to fit the exam's theme, Huzzah! Soon it could be time to start your LabTA career. Let's make sure you are ready for that endeavor.

a. Some students' task is to write an assembly function that returns $x \cdot 2^n$ for any `long x` and non-negative `int n`. To show you are up to the task of helping them, write the flattened C code corresponding to a *more general* version of this function that also accepts negative values for n, using the standard library function `abs` (which takes one `int` and returns its absolute value as an `int`):

C

```
long xTimes2ToN(long x, int n) {
   long result = x;

   if (n >= 0)
      result = result << n;
   else
      result = result >> abs(n);

   return result;
}
```

Flattened C

```
long xTime2ToN(long x, int x) {
     long result = x;

     if( n < 0 ) goto ElseClause;
     result = result << n;
     goto AfterElse;

ElseClause:
     result = result >> abs(n);

AfterElse:
     return result;
}
```

Now show you are ready to debug the numerous, creative, and diverse ways that assembly code can be subtly wrong. To help you do this, you can refer to this abbreviated ARM assembly language reference guide:

Instruction(s)	Description
`{add,sub,lsl} dst, src1, src2`	`dst = src1 {+, -, <<} src2`
`ldr dst, [src]`	Load 4 or 8 bytes pointed to by `src` into `dst`
`ldrsw dst, [src]`	Load 4 bytes pointed to by `src` into `dst` , then sign-extend that value to 8 bytes. Used because we cannot mix `w` and `x` registers in `lsl`
`str src, [dst]`	Store 4 or 8 bytes in `src` to memory pointed to by `dst`
`mov dst, src`	Copy `src` to `dst`
`ret`	Return to address pointed to by `x30`
`{x0-x7 , x0}`	Used for {arguments to , return value from} function

The simplified version of the function, called `powerShift`, has the following C code for students to translate into ARM assembly language:

```
long powerShift(long x, int n) {
    long result = x;
    result = result << n;
    return result;
}
```

On each of the next five pages, you are given an incorrect version of the assembly code for `powerShift` and the symptoms of the bug (e.g. returning the wrong answer or crashing). For each version, use the box below the assembly code to describe that version's bug in no more than two sentences.

b. Buggy Assembly 1
(Returns incorrect result.)

```
   .global powerShift
powerShift:
   // long powerShift(long x, int n) {
   .equ powerShift_BYTE_COUNT, 32
   // parameter stack offsets
   .equ x, 8
   .equ n, 16
   // local variable stack offset
   .equ result, 24

   //prolog
   sub sp, sp, powerShift_BYTE_COUNT
   str x30, [sp]

   //   long result = x;
   ldr x0, [sp, x]
   str x0, [sp, result]

   //   result = result << n;
   ldr x0, [sp, result]
   ldrsw x1, [sp, n]
   lsl x0, x0, x1
   str x0, [sp, result]

   // Epilog and return result;
   ldr x0, [sp, result]
   ldr x30, [sp]
   add sp, sp, powerShift_BYTE_COUNT
   ret
   .size powerShift, (. - powerShift)
```

Bug Description 1

x0 is not stored into [sp, x] and w1 is not stored into [sp, n], but the program subsequently loads from those memory locations

c. Buggy Assembly 2
(Returns incorrect result.)

```
   .global powerShift
powerShift:
   // long powerShift(long x, int n) {
   .equ powerShift_BYTE_COUNT, 32
   // parameter stack offsets
   .equ x, 24
   .equ n, 16
   // local variable stack offset
   .equ result, 8

   //prolog
   sub sp, sp, powerShift_BYTE_COUNT
   str x30, [sp]
   str w0, [sp, x]
   str w1, [sp, n]

   //   long result = x;
   ldr x0, [sp, x]
   str x0, [sp, result]

   //   result = result << n;
   ldr x0, [sp, result]
   ldrsw x1, [sp, n]
   lsl x0, x0, x1
   str x0, [sp, result]

   // Epilog and return result;
   ldr x0, [sp, result]
   ldr x30, [sp]
   add sp, sp, powerShift_BYTE_COUNT
   ret
   .size powerShift, (. - powerShift)
```

Bug Description 2

w0, rather than x0, is stored into [sp, x]

d. Buggy Assembly 3
(Does not assemble.)

```
   .global powerShift
powerShift:
   // long powerShift(long x, int n) {
   .equ powerShift_BYTE_COUNT, 32
   // parameter stack offsets
   .equ x, 8
   .equ n, 16
   // local variable stack offset
   .equ result, 24

   //prolog
   sub sp, sp, powerShift_BYTE_COUNT
   str x30, [sp]
   str x0, [sp, x]
   str w1, [sp, n]

   //   long result = x;
   mov x0, [sp, x]
   str x0, [sp, result]

   //   result = result << n;
   ldr x0, [sp, result]
   ldrsw x1, [sp, n]
   lsl x0, x0, x1
   str x0, [sp, result]

   // Epilog and return result;
   ldr x0, [sp, result]
   ldr x30, [sp]
   add sp, sp, powerShift_BYTE_COUNT
   ret
   .size powerShift, (. - powerShift)
```

Bug Description 3

`mov` tries to load from memory by dereferencing `[sp, x]`, which isn't a valid version of `mov`. `ldr` would have been the right instruction to load from memory.

e. Buggy Assembly 4
(Crashes with a segfault.)

```
   .global powerShift
powerShift:
   // long powerShift(long x, int n) {
   .equ powerShift_BYTE_COUNT, 32
   // parameter stack offsets
   .equ x, 8
   .equ n, 16
   // local variable stack offset
   .equ result, 0

   //prolog
   sub sp, sp, powerShift_BYTE_COUNT
   str x30, [sp]
   str x0, [sp, x]
   str w1, [sp, n]

   //   long result = x;
   ldr x0, [sp, x]
   str x0, [sp, result]

   //   result = result << n;
   ldr x0, [sp, result]
   ldrsw x1, [sp, n]
   lsl x0, x0, x1
   str x0, [sp, result]

   // Epilog and return result;
   ldr x0, [sp, result]
   ldr x30, [sp]
   add sp, sp, powerShift_BYTE_COUNT
   ret
   .size powerShift, (. - powerShift)
```

Bug Description 4

The local variable `result` is stored at offset `0` from `sp`, which Is also where `x30` is stored in the prolog. This corrupts the saved value of the address to which `ret` will return.

f. Buggy Assembly 5
(Crashes with a segfault.)

```
    .global powerShift
powerShift:
    // long powerShift(long x, int n) {
    .equ powerShift_BYTE_COUNT, 32
    // parameter stack offsets
    .equ x, 8
    .equ n, 16
    // local variable stack offset
    .equ result, 24

    //prolog
    sub sp, sp, powerShift_BYTE_COUNT
    str x30, [sp]
    str x0, [sp, x]
    str w1, [sp, n]

    //   long result = x;
    ldr x0, [sp, x]
    str x0, [sp, result]

    //   result = result << n;
    ldr x0, [sp, result]
    ldrsw x1, [sp, n]
    lsl x0, x0, x1
    str x0, [sp, result]

    // Epilog and return result;
    add sp, sp, powerShift_BYTE_COUNT
    ldr x30, [sp]
    ldr x0, [sp, result]
    ret
    .size powerShift, (. - powerShift)
```

Bug Description 5

The epilogue is in the wrong order: tearing down the stackframe before loading the saved return address and the return value from the stack. The values loaded into `x30` and `x0` will be nonsensical data from a different stackframe!