Top of Form

**Q1**

Nearby is source code along with a terminal session that compiles that source. Analyze the source code and following questions.

> cat incval\_main.c

#include <stdio.h>

void incval(int \*x);

int main(){

int a = 1;

printf("a: %d\n",a);

incval(&a);

printf("a: %d\n",a);

return 0;

}

> cat incval\_func.c

void incval(float \*x){

float y = \*x;

\*x = y+1.0;

}

> gcc incval\_main.c incval\_func.c

> ./a.out

a: 1

a: 1065353216

**A**

The line incval(&a); in the main() function uses a capability in C which is not present in most other high-level programming languages. Describe whether the Value for variable a is used on this line or something else instead.



**B**

The two C files that are compiled and linked contain definitions for incval. Describe what definition(s) for incval exist in the resulting a.out file and why one definition is chosen for the executable over the other(s). Mention which are Strong and Weak definitions in your answer.



**C**

Explain the extremely strange values that are printed when the executable runs. Demonstrate your understanding of the binary layout of the different number types we have studied in the class and how these come into play here.



**D**

What common C technique is used to ensure that all C files (compilation units) are aware and agree on the types for functions and variables? How would it be used in to prevent the errors that appear in this session?



**Q2**

Nearby is output of compiling/running the program myprog which prints some of its memory addresses. While it is run, pmap is used to print information about the running program. Answer the following questions about what is shown in the output.

// FILE: myprog.c

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int ack(int x){

printf("The number is %d\n",x);

}

int yikes=456;

int whoa[1024];

int zoinks[2048]={7,8,9};

int main(){

int blerg = 123;

int \*meh = malloc(8);

printf("ack: %016p\n",ack);

printf("main: %016p\n",main);

printf("yikes: %016p\n",&yikes);

printf("whoa: %016p\n",whoa);

printf("zoinks:%016p\n",&zoinks);

printf("meh: %016p\n",meh);

printf("blerg: %016p\n",&blerg);

int pid = getpid();

printf("my pid is %d\n",pid);

printf("press enter to continue\n");

getc(stdin); // wait for 'enter'

return 0;

}

## TERMINAL 1 ##

> gcc -o myprog myprog.c

> ./myprog

ack: 0x0055bac8dff299

main: 0x0055bac8dff2C2

yikes: 0x0055bac8e02080

whoa: 0x0055bac8e040c0

zoinks:0x0055bac8e020a0

meh: 0x0055baca7d82a0

blerg: 0x007ffc7a78dfd8

my pid is 820467

press enter to continue

## TERMINAL 2 ##

> pmap 820467

820467: ./myprog

0055bac8dfe000 4K r---- myprog # 1

0055bac8dff000 4K r-x-- myprog # 2

0055bac8e00000 4K r---- myprog # 3

0055bac8e01000 4K r---- myprog # 4

0055bac8e02000 12K rw--- myprog # 5

0055bac8e05000 4K rw--- [ anon ] # 6

0055baca7d8000 132K rw--- [ anon ] # 7

007f559ab31000 8K rw--- [ anon ] # 8

007f559ab33000 152K r---- libc-2.32.so # 9

007f559ab59000 1332K r-x-- libc-2.32.so #10

007f559aca6000 304K r---- libc-2.32.so #11

007f559acf2000 12K r---- libc-2.32.so #12

007f559acf5000 12K rw--- libc-2.32.so #13

007f559acf8000 24K rw--- [ anon ] #14

007f559ad2c000 8K r---- ld-2.32.so #15

007f559ad2e000 132K r-x-- ld-2.32.so #16

007f559ad4f000 36K r---- ld-2.32.so #17

007f559ad58000 4K r---- ld-2.32.so #18

007f559ad59000 8K rw--- ld-2.32.so #19

007ffc7a76f000 132K rw--- [ anon ] #20

007ffc7a7cb000 12K r---- [ anon ] #21

total 2348K

**A**

For each of the below C entities, identify which page line of pmap output accounts for its location. Output lines are numbered #1 #2 etc.. ALSO indicate which the 4 main Memory Areas the C entity exists within.

ack pmap Output Line# and Memory Area:



yikes pmap Output Line# and Memory Area:



whoa pmap Output Line# and Memory Area:



main pmap Output Line# and Memory Area:



blerg pmap Output Line# and Memory Area:



**B**

The C symbols for main and blerg are declared very close together in the source code for myprog.c. Are the Virtual addresses for these two entities close together or not? Explain why in a few sentences.



**C**

The entities yikes and whoa are declared very close together in the source code for myprog.c. Are the Virtual addresses for these two entities close together or not? What can be said about the Physical addresses for these: are they on the same Physical Page? Explain why in a few sentences.



**Q3**

Programmers are accustomed to a great deal of control in their programs but we have learned that at the lowest level, some parts of hardware have a mind of their own. Consider a simple declaration like int x = 5;. Describe the parts of the physical memory system at which the compiler and operating system may place this variable and why programmers don't have much control over this. Full credit answers will mention at least **3 physical locations** that a copy of 5 may appear associated with x.



**Q4**

We have seen that, when retrieving the value associated with a 64-bit memory address, the address is often divided into chunks by the memory system which then uses the chunks differently. Discuss how the bits in a Memory Address are broken into chunks and used in the following 2 settings. Describe briefly how the chunks of address bits are used in both cases.

**A 5pts**

To check for a value in the CPU Cache Memory:



**B**

To translate a Virtual Address to a Physical Address:



**Q5**

Nearby is an outline for code to print records in a simple binary file format using mmap(). The file is arranged into 3 sections:

1. The Header section which contains overall file offsets in a fhead\_t struct. This section **appears first in the file.**
2. The Names section which is a series of null-terminate strings.
3. The IDNM (ID/Names) section which contains an array of idnm\_t which correspond numeric IDs to names.

**Relevant C structures**

typedef struct { // IDS STRUCT

int id; // numeric of a person

size\_t nm\_off; // name offset in Names sec

} idnm\_t;

typedef struct { // FILE HEADER STRUCT

size\_t idnm\_sec\_off; // offset of IDNM sec

size\_t idnm\_sec\_count; // # IDNM array elems

size\_t nm\_sec\_off; // offset of Names sec

} fhead\_t;

**Expected Output**

Several runs of the program are shown below.

> gcc -o idnm\_lookup idnm\_lookup.c

> ./idnm\_lookup ids-names.dat 123

FOUND at index 0

123 : Klaus

> ./idnm\_lookup ids-names.dat 345

FOUND at index 2

345 : Sunny

> ./idnm\_lookup ids-names.dat 666

FOUND at index 3

666 : Olaf

> ./idnm\_lookup ids-names.dat 789

789 NOT FOUND

**OUTLINE / TEMPLATE**

Copy the template provided and WRITE CODE in teh sections marked A. B. etc. to complete the program. Each space should require 1-4 lines of C code.

#include "idnm.h" // contains PTR\_PLUS\_BYTES() macro

int main(int argc, char \*argv[]){

// (A) Exit if less than 3 command line arguments

char \*filename = argv[1];

// (B) Convert second command line arg to an integer

int search\_id =

// set up for memory mapped file

int fd = open(filename, O\_RDONLY);

struct stat stat\_buf;

fstat(fd, &stat\_buf);

int size = stat\_buf.st\_size;

// (C) Memory map file

void \*file\_bytes =

// (D) Set to fhead\_t struct at beginning of the file

fhead\_t \*fhead =

// (E) Set positions of IDNM Array / Names Sections

idnm\_t \*idnm\_sec =

char \*nm\_sec =

// loop to find entry with search\_id and print name

int found = 0;

for(int i=0; i<fhead->idnm\_sec\_count && !found; i++){

// (F) Calculate position of name for this entry

char \*name =

// (G) Check if search\_id matches; when found print

// FOUND at index 'i'

// 'search\_id' : 'name'

// filling in values for the above, set found=1

if(!found){

printf("%d NOT FOUND\n",search\_id);

}

munmap(file\_bytes, size);

close(fd);

return 0;

}

Enter your completed code here:



Bottom of Form