Name: SAMPLE ANSWERS CSCI 237
Practice Problems: Cache Memories

Structs and Memory

Consider the following C structure that contains multiple fields and is similar to one you may want to use in your Cache lab:

```
struct simulation {
   int s;
   int b;
   int E;
};
```

The `struct simulation` contains 3 fields, each corresponding to some parameter relevant to your cache simulation.

Declaring a local `struct simulation` variable allocates enough space for the structure on the stack frame of the current function. The local variable's fields can be accessed with dot notation, and the memory is reclaimed once the function returns.

Passing a local struct variable as an argument to a function creates a complete bitwise copy of the structure. This means that changes to the parameter's value inside the function do not affect the original struct.

Consider the following code:

```
void initialize_simulation(struct simulation sim) {
   sim.s = 3;
   sim.b = 3;
   sim.E = 4;
}

int main(int argc, char *argv[]) {
   struct simulation sim;
   initialize_simulation(sim);
   printf("s: %d, b: %d, E: %d\n", sim.s, sim.b, sim.E);
}
```

What is printed when the function `main()` is executed?
`s: 0, b: 0, E: 0`

The `printf` function displays 3 integer values that we unfortunately have no information about! In the example answer above, 0 is printed, but it could be any integer value---`printf` would interpret whatever values were in the memory where the main function's stack frame was created.

Instead, we would like this program to print the output:
`s: 3, b: 3, E: 4`

We have inserted whitespace where additions *may* be needed; in some cases we also deleted necessary notation that you will need to restore. Modify the code so that it correctly complies and prints the desired output.

```
void initialize_simulation(struct simulation * sim   ) {
   sim -> s = 3;

   sim -> b = 3;

   sim -> E = 4;
}

int main(int argc, char *argv[]) {
   struct simulation   sim;

   initialize_simulation(   &sim   );

   printf("s: %d, b: %d, E: %d\n",
          sim . s, sim . b, sim . E);
}
```

What changes were necessary and why?

It still makes sense to declare a local variable `struct simulation sim`; inside `main()`, because that avoids the need to manually allocate memory with `malloc()`/`free()`. We also only use `sim` inside this function, so we don't need the variable's lifetime to extend beyond main().

However, if we want `initialize_simulation()` to modify the values that exist inside `main()`'s stack frame, we need to pass it a pointer to the location of those values. This changes the type of the function's argument to a `struct simulation *`, and requires us to pass the address of `sim` (using `&`) when we invoke that function within `main()`.

Finally, the accesses to the struct fields `s`, `b`, and `E` inside `initialize_simulation()` are now accessing the fields of a `struct simulation *`, so we need to use an arrow (`->`) instead of dot (`.`) notation.

File Reading and Parsing

There are several ways to interact with files using C. When we simply want to read form a file, it may be helpful to compare it to steps we take when reading from a book.
The first step is to open the book. Once the book is open, we can begin reading from the beginning. Each word that we read moves us forward, and we use a finger to track our current position. If we pause our reading, we pick up again from the current word that our finger is pointing to. When we are done for good, we close the book.

A file can be opened using the function `fopen()`, passing the file's pathname and the permissions as arguments (e.g., `'r'` for read-only). What is returned is a pointer to a data structure that keeps track of our current position within the open file (like our finger). Each time we read data, our position (finger) advances, and subsequent reads pick up from where we left off. Once we are done reading, we must close the file (like we close a book).

```
FILE *fp;
char str[60];

/* opening file for reading */
fp = fopen("file.txt" , "r");
if(fp == NULL) {
  perror("Error opening file");
  return(-1);
}
if(fgets(  str    ,  60     , fp) != NULL) {
  printf("%s", str);
}
fclose(fp);
```

The above code is not yet complete, yet it is close enough that you should be able to reason about it.

What do you think is the purpose of the variable `char str[60]`?

It allocates a memory region that is 60 bytes large that we can use to store data that we read from the file.

Based on your understanding of the documentation for `fgets()` (found by typing `man fgets` at the command line, or by searching for the "linux fgets manual page" online), what should be the first two arguments to `fgets()`?

`str, 60`

We saw the function `sscans()` in the “bomb lab”. It lets us read formatted input from a string. Consider the incomplete C code below:

```
int main (int argc, char *argv[]) {
  int day, year;
  char weekday[20], month[20], dtm[100];
  strcpy(dtm, "Saturday March 25 1989");
  sscanf(dtm, "%s %s %d %d", weekday, month, &day, &year);
  printf("%s %d, %d = %s\n", month, day, year, weekday);
  return 0;
}
```

The function `strcpy()` copies a null-terminated string from the address pointed to by its second argument into the memory at the address pointed to by its first argument.

The function `sscanf()` uses a format string to convert components of the string passed as its first argument and store them at locations specified by its last arguments.

The output printed by this program should be:

```
March 25 1989 = Saturday
```

Complete the program so that it achieves this output.

The sscanf accepts a source string, a format string, and then a variable number of arguments. The number of arguments after the format string should match the number of components of the format string that need to be matched (for example, “%s %s %d %d” would require 4 pointer arguments, since it is trying to match 4 components: two strings and two integers). The pointer arguments should all correspond to appropriate types based on the format string. Sscanf will scan through the source string, and attempt to match its contents from left to right to the components of the format string. For each successful match, it updates the next pointer argument with the matched value.

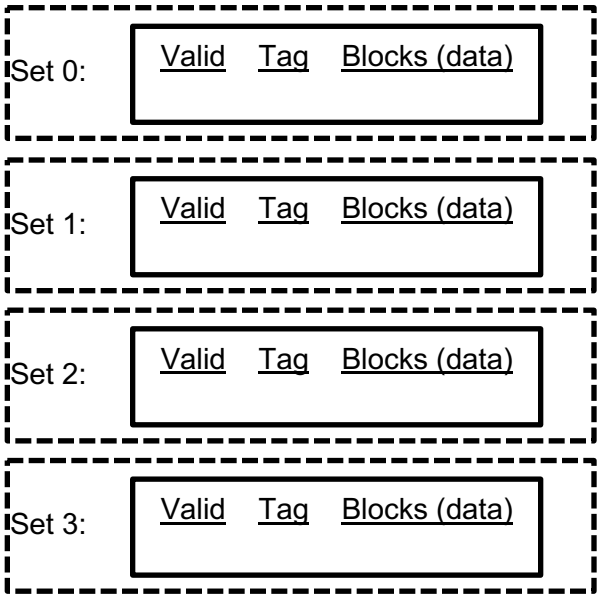
Weekday and months are already pointers (char *), but day and year are not. This is why we need to use the address-of operator (&) so that we can get a pointer to the memory where those int variables are stored.

Cache Memories

Consider a system with sixteen bytes of memory, and a direct mapped cache (E=1) with 4 sets (s=2) and a block size of 2 (b=1). For each address below, underline the set bits, and note which set index it maps to.

Address	Set Index
0000	0
0001	0
0010	1
0011	1
0100	2
0101	2
0110	3
0111	3
1000	0
1001	0
1010	1
1011	1
1100	2
1101	2
1110	3
1111	3

(we break apart the address as follows: TSSB)



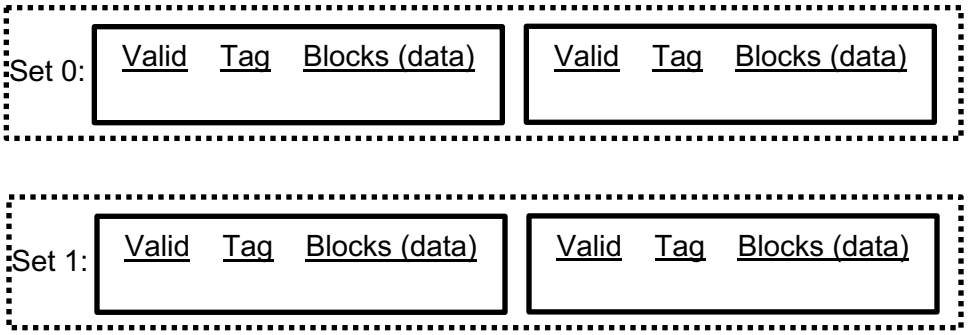
To the right of the table above, sketch a picture that corresponds to the dimensions of the cache described. Use your drawing and your understanding of cache memories to indicate whether reads to the (hex) addresses below correspond to hits or misses in the cache.

Address	Hit (H)/Miss (M)
C	M -> bring in block CD
3	M -> bring in block 23
9	M -> bring in block 89
0	M -> evict, bring in block 01
8	M -> evict, bring in block 89
3	H
B	M -> evict, bring in block AB
F	M -> bring in EF
4	M -> evict, bring in 45
5	H
7	M -> evict, bring in 67
D	M -> evict, bring in CD
C	H
6	H
1	M -> evict, bring in 01
A	H
E	M -> evict, bring in EF

Now consider a system with sixteen bytes of memory, and a two-way set-associative cache (E=2) with 2 sets (s=1) and a block size of 2 (b=1). For each address below, underline the set bits, and note which set index it maps to.

Address	Set Index
0000	___0___
0001	___0___
0010	___1___
0011	___1___
0100	___0___
0101	___0___
0110	___1___
0111	___1___
1000	___0___
1001	___0___
1010	___1___
1011	___1___
1100	___0___
1101	___0___
1110	___1___
1111	___1___

(we break apart the address as follows: TT**S**B)



Once again, sketch a picture that corresponds to the dimensions of the cache described. Use your drawing and your understanding of cache memories to indicate whether reads to the (hex) addresses below correspond to hits or misses in the cache. When evicting, use the LRU policy.

Address	Hit (H)/Miss (M)
C	**M -> bring in block CD**
3	**M -> bring in 23**
9	**M -> bring in 89**
0	**M -> evict CD, bring in 01**
8	**H**
3	**H**
B	**M -> bring in AB**
F	**M -> evict 23, bring in EF**
4	**M -> evict 01, bring in 45**
5	**H**
7	**M -> evict AB, bring in 67**
D	**M -> evict 89, bring in CD**
C	**H**
6	**H**
1	**M -> evict 45, bring in 01**
A	**M -> evict EF, bring in AB**
E	**M -> evict 67, bring in EF**