Instructions. Create an EECS instructional class account if you have not already. To do so, visit https://acropolis.cs.berkeley.edu/~account/webacct/, click “Login using your Berkeley CalNet ID,” then find the cs161 row and click “Get a new account.” Be sure to take note of the account login and password.

Log in to your instructional account, create an empty directory, and create files q1.txt, q2.txt, q3.txt, and (optionally) q4.txt in that directory following the instructions below. To submit the first part of your homework solution, type submit hw1 from that directory. If you realize you missed anything, you can always submit again up until the submission deadline.

For Q5, please write your answers on a separate sheet of paper. Put your name and student ID at the top, and submit it using Gradescope. You should have received an email with an invitation to Gradescope; use that account to submit Q5. Q5 is not optional—don’t forget to submit your answer to Q5 on Gradescope.

This homework is due Monday, February 1st, at 11:59pm. It must be submitted electronically. No late homeworks will be accepted.

Problem 1 Policy (10 points)
The aim of this exercise is to ensure that you read the course policies, as well as to make sure that you are registered in the class and have a working EECS instructional class account.

Open the course website http://www-inst.eecs.berkeley.edu/~cs161/sp16/. Append ?lastname=<name>&userid=cs161-xy to the URL in the address bar, where <name> is your last name (as in campus records) and cs161-xy is your class ID (with xy replaced with the two final characters of your class account). (If you have spaces or apostrophes in your last name, go ahead and type them in: they should not cause any problems.) Thus, the URL you will open is:

http://www-inst.eecs.berkeley.edu/~cs161/sp16/?lastname=<name>&userid=cs161-xy

Please read and check that you understand the course policies on that page. If you have any questions, please ask for clarification on Piazza.

To receive credit for having read the policies, create a text file called q1.txt and put in it the string “I understand the course policies.” (no quotes necessary)
Solution: The correct file contents are:

I understand the course policies.
:-)
Problem 4  Memory Layout  
Consider the following C code:

```c
int frog(int i, int j, int k) {
    int t = i + j;
    char *buf = malloc(2);
    t = t + k;
    return t;
}

void duck(char *buffer) {
    int i[2];
    i[0] = 1;
    i[1] = frog(i[0], 2, 3);
}

int main() {
    char foo[4];
    duck(foo);
    return 0;
}
```

The code is compiled and run on a 32-bit x86 architecture (i.e., IA-32). Assume the program is run until line 5, meaning everything before line 5 is executed (i.e., a breakpoint was set at line 5). We want you to sketch what the layout of the program’s stack looks like at this point. In particular, print the template provided on the next page and fill it in. Fill in each empty box with the value in memory at that location. Put down specific values in memory, like 1 or 0x00000000, instead of symbolic names, like foo. Also, on the bottom, fill in the values of `%ebp` and `%esp` when we hit line 5.

Assumptions you should make:
- memory is initially all zeros
• execution starts at the very first instruction in main(), and %esp and %ebp start at 0xc0000000 at that point
• the call to malloc returns the value 0x12345678
• the address of the code corresponding to line 17 is 0x01111110
• the address of the code corresponding to line 12 is 0x01111144
• a char is 1 byte, an int is 4 bytes
• no function uses general-purpose registers that need to be saved (other than %ebp)

See the next page for the template.

Solution: See next page for solution
Going through the code line by line and trying to figure out what the assembly instructions they correspond to will help us figure out what the stack looks like.

First let’s consider what happens at the very start of \texttt{main}. We know that all functions have a prologue that creates a new stack frame for the function, which looks like:

\begin{verbatim}
main:
    push %ebp
    mov %esp, %ebp
    sub $???, %esp
...
\end{verbatim}

The \texttt{???} will correspond to the total size needed for all of \texttt{main's} local variables, which is 4 (4 bytes for the local variable \texttt{foo}).

Initially:
\begin{verbatim}
%ebp=0xc0000000
%esp=0xc0000000
\end{verbatim}

After the prologue instructions have been executed, the value 0xc0000000 will have been pushed onto the stack and the new values will be:
\begin{verbatim}
%ebp=0xbfffffff
%esp=0xbfffffff8
\end{verbatim}

Next, we have the function call \texttt{duck(foo)}. Calling convention dictates that arguments be pushed on the stack in reverse order before making the function call. The assembly for this might look like:

\begin{verbatim}
main:
    ...
    push ???
\end{verbatim}
call duck
...

The argument to the function call is a pointer, so we will be pushing 0xbfffe8 onto the stack (which is the space on the stack for char foo[4]). After that, we will push the return address, which is given as 0x111100.

At this point the stack looks like:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbfffffff0c</td>
<td>0xc0000000</td>
</tr>
<tr>
<td>0xbfffffff8</td>
<td>0</td>
</tr>
<tr>
<td>0xbfffffff4</td>
<td>0xbfffffff8</td>
</tr>
<tr>
<td>0xbfffffff0</td>
<td>0x1111110</td>
</tr>
</tbody>
</table>

%ebp=0xbfffffffcc
%esp=0xbfffffff0

Now we move into duck, which, just like main, will have a function prologue that sets up a new stack frame.

duck:
push %ebp
mov %esp, %ebp
sub $8, %esp

The local variable int i[2] requires 8 bytes, so the stack is subtracted by 8.

The stack at this point:
Next up is the line \texttt{i[0] = 1}. We know that the 4-byte slots at \texttt{0xbfffffe4} and \texttt{0xbfffffe8} have been allocated for \texttt{int i[2]}, but which of these slots corresponds to \texttt{i[0]}? Higher array indexes always correspond to higher memory addresses, so \texttt{i[0]} would actually refer to the 4-byte slot at \texttt{0xbfffffe4}.

Afterwards is another assignment statement, but with a function call on the right instead of a constant. The function call \texttt{frog(i[0], 2, 3)} will have to be completed before the assignment can be executed. Just as before, we need to push the arguments on the stack in reverse order before the call. Although \texttt{i[0]} lives in the stack, the value of \texttt{i[0]} still needs to be pushed as an argument to the function call (due to C’s pass by value nature). After the arguments have been pushed, the return address \texttt{0x1111144} will be pushed.

The stack at this point:
Now we move into **frog**, which again will begin with setting up a new stack frame. **frog** has local variables `int t` and `char *buf`, meaning 8 bytes will be subtracted from the stack pointer to make room. The order of these two local variables isn’t guaranteed, but for now we’ll just assume the higher memory address slot is used for `int t` and the lower one is used for `char *buf`. These values will change according to the assignments.

The final stack:
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbffffffffc:</td>
<td>0xc0000000</td>
</tr>
<tr>
<td>0xbffffffff8:</td>
<td>0</td>
</tr>
<tr>
<td>0xbffffffff4:</td>
<td>0xbffffffff8</td>
</tr>
<tr>
<td>0xbffffffff0:</td>
<td>0x1111110</td>
</tr>
<tr>
<td>0xbffffffffec:</td>
<td>0xbffffffffc</td>
</tr>
<tr>
<td>0xbffffffffe8:</td>
<td>0</td>
</tr>
<tr>
<td>0xbffffffffe4:</td>
<td>1</td>
</tr>
<tr>
<td>0xbffffffffe0:</td>
<td>3</td>
</tr>
<tr>
<td>0xbffffffffdc:</td>
<td>2</td>
</tr>
<tr>
<td>0xbffffffffd8:</td>
<td>1</td>
</tr>
<tr>
<td>0xbffffffffd4:</td>
<td>0x11111144</td>
</tr>
<tr>
<td>0xbffffffffd0:</td>
<td>0xbffffffffec</td>
</tr>
<tr>
<td>0xbffffffffcc:</td>
<td>6</td>
</tr>
<tr>
<td>0xbffffffffc8:</td>
<td>0x12345678</td>
</tr>
</tbody>
</table>

%ebp = 0xbffffffffd0

%esp = 0xbffffffffc8