SID: _____

Question 1: Main [Memory] Stacks (16 pts)

Recall the idea of the Stack section in our memory hierarchy: when functions are called, their function frames are **pushed** onto the stack; when a function returns, it is **popped** off the stack. We will be implementing the Stack section of memory in software by representing each function frame on the stack as a struct, defined below. Assuming we are running on a **32-bit machine** with **32-bit integers**.

```
typedef struct stack {
      void *frame;          // Pointer to the space allocated for this function frame
      struct stack *prev;   // Pointer to previous stack node
      char *func_name;      // Name of the stack frame's function
      int threads;          // The number of threads below this stack frame
      int in_use;           // The number of threads currently pointing to this stack frame
} StackNode;

StackNode *sp = NULL;
```

1) What would `sizeof (StackNode)` return?

_____

Suppose that the staff has implemented a function `frame_size` which takes in a C function's name as a string and outputs how many **bytes** that the function's local variables would need in the function frame:

```
int frame_size (char *func_name) {
      \\ Assume this function has been correctly implemented for you.
}
int main_stacks (int argc, char *argv[]) {
      char arr[] = "61C rox";
      return 0;
}
```

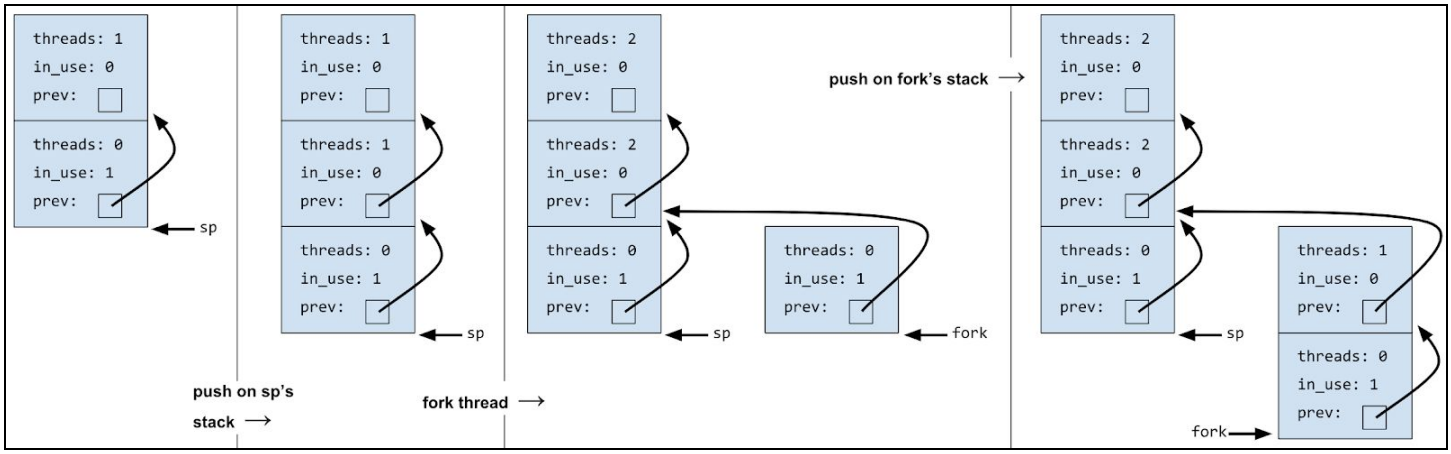
2) What would `frame_size ("main_stacks")` return?

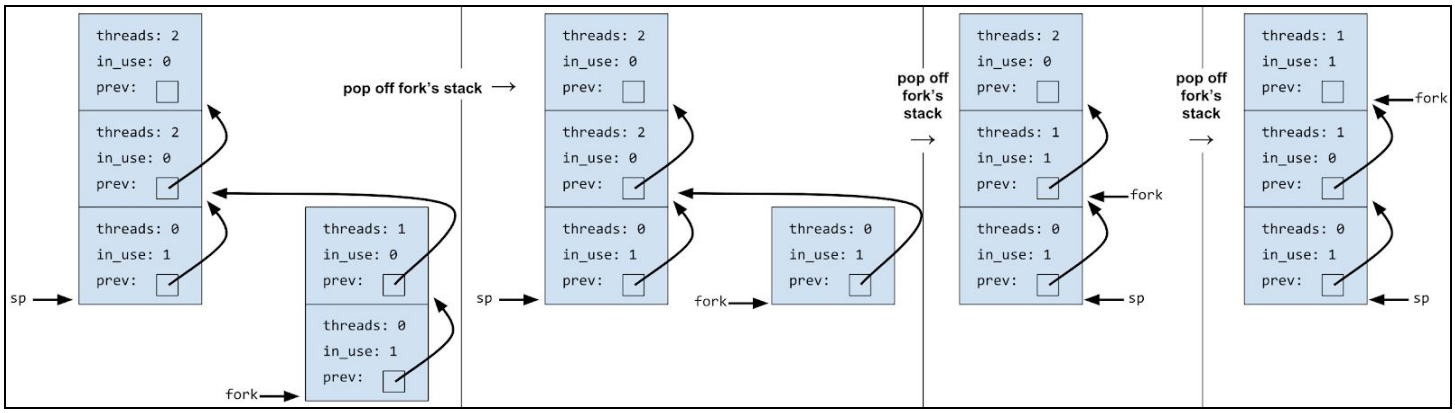
_____

Recall that for a single thread, its Stack section is essentially a linked list where the stack pointer **sp** points to the end of the stack. Also recall that when you **fork** a thread, the new thread will have their own copy of the stack. Instead of actually copying the function frames, we just point to the "shared" stack frames.

On the next page is a visual of the stack that we will be designing. The first set is an example of pushing function frames on the stack along with creating a new fork. The second set is continuously popping off from the fork's stack.

SID: _____





Fill in the following code sequence that will push the function frame onto the stack and modify the `sp` passed in. Assume `sp` is never `NULL` and `func_name` is always stored in static data—don't allocate space for it.

```
void push (char *func_name, StackNode **sp) {
      StackNode *temp = (StackNode *) malloc(sizeof(StackNode));
      temp->frame = malloc(_____________________________________);
      temp->prev = _______________;
      temp->func_name = _________________;
      temp->threads = _______________;
      temp->in_use = _______________;
      if (*sp == NULL) {
            _____________________________;
            return;
      }
      if ((*sp)->in_use > 0) {
            (*sp)->threads += 1;
            (*sp)->in_use -= 1;
      } else {    /* We are making a fork. */
            for (_____________________________________________________________) {
            ____________________________________________________________
            }
      }
      ____________________________________________________________________
}
```

SID: _____

Fill out the following code sequence that will pop the function frame off the stack, modifying the `sp` passed in. Also assume that `sp` is never `NULL`. Since our stack frame is possibly shared by multiple "threads" you will want to consider when we should free the memory for a frame we pop off.

```
void pop (StackNode **sp) {
          printf("%s has been returned\n", (*sp)->func_name);
          (*sp)->in_use--;
          bool free_frame = false;
          if (_____________________________________________) {
             free_frame = true;
          }
          StackNode *temp = (*sp)->prev;
          if (free_frame) {
              _____________________________;
              _____________________________;
          }
          _____________________________;
          if ((*sp) != NULL) {
              _____________________________;
              _____________________________;
          }
}
```

Question 2: Main [Memory] Stacks Management (8 pts)

In this question, we will continue the Stack implementation from Question 1, assuming it functions correctly. Consider the following program (and feel free to draw the stack in the space to the right :D):

```
StackNode *sp = NULL;
int main(int argc, char *argv[]) {
      push("main", &sp);
      push("foo", &sp);
      push("bar", &sp);
      StackNode *fork = sp;
      push("orig", &sp);
      push("split", &fork);
      return 0;
}
```

Each of the following values on the next page evaluate to an address in the C code above. Select the region of memory that these addresses point to right before the `main` function returns.

SID: _____

1. `main`	Ⓐ Stack	Ⓑ Heap	Ⓒ Static	Ⓓ Code
2. `&sp`	Ⓐ Stack	Ⓑ Heap	Ⓒ Static	Ⓓ Code
3. `sp`	Ⓐ Stack	Ⓑ Heap	Ⓒ Static	Ⓓ Code
4. `*sp`	Ⓐ Stack	Ⓑ Heap	Ⓒ Static	Ⓓ Code
5. `sp->func_name`	Ⓐ Stack	Ⓑ Heap	Ⓒ Static	Ⓓ Code
6. `&fork`	Ⓐ Stack	Ⓑ Heap	Ⓒ Static	Ⓓ Code
7. `argv`	Ⓐ Stack	Ⓑ Heap	Ⓒ Static	Ⓓ Code

Suppose we had a simple recursive function defined as follows:

```
long factorial(long n):
    if (n == 1):
        return 1;
    else:
        return n * factorial(n-1)
```

Assume that function frames only require space for the local variables (i.e. the return value of `frame_size("factorial")`). You are given the following specifications:

Stack and Heap: 16 KiB
Static: 12 KiB
Code: 4 KiB

```
frame_size("factorial") = 8
sizeof(StackNode) = 56
```

Suppose we call the `factorial` function on some number N **using our Stack data structure from Question 1** (note: we allocated data for our `StackNode` structs):

```
StackNode *sp = NULL;
int main () {
      push("factorial", &sp);
      push("factorial", &sp);
      push("factorial", &sp);
      ...

      // the N'th call to factorial
      push("factorial", &sp);

      // the first return from factorial
      pop(&sp);
      ...
}
```

What is the smallest value of N that will cause a maximum recursion depth error (meaning no more function frames can be created)? Ignore the stack space required for the `main` function. If convenient, put your answer as a power of 2.

N = _____