

# Stack Memory

I am sure a lot of you have used [stackoverflow.com](https://stackoverflow.com) right? Stack overflows is one of the hardest problems to catch and diagnose, and thus no wonder someone invented this creative website URL.

## Stack Memory consumers

Stack memory is required for the following:

1. Local Variables: Stored in the stack frame and automatically cleaned up when the function exits.
2. Function Calls: When a function is called, a stack frame is created and pushed onto the stack.
3. Function Returns: When a function completes, its stack frame is popped off the stack.

Fundamentally, a single core CPU contains a "Stack Pointer" which is a hardware register keeping track of memory. When a compiler generates code for a local variable, the stack pointer is typically decremented to make space for that variable.

```
void example_function() {
    int local_variable;
}
// Assembly:
example_function:
    push {lr}                // Save the link register (return address)
    sub sp, sp, #4          // Decrement the stack pointer by 4 bytes to allocate space for localV
    ...
    add sp, sp, #4          // Clean up the stack by incrementing the stack pointer
    pop {pc}                // Restore the link register and return from the function
```

## POSIX Thread

Let us experiment with a POSIX thread and visualize the stack usage.

```
#include <pthread.h>
#include <unistd.h> // sysconf()
```

```

#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
void* thread_entry(void *p) {
    //puts("Hello from thread");
    //uint8_t memory[280 * 1024] = { 0 };
    //uint8_t *memory = malloc(1 * 1024 * 1024);
    return NULL;
}
void *allocate_stack(const size_t stack_size_in_bytes) {
    const size_t align_to = sysconf(_SC_PAGESIZE);
    void *aligned_stack_ptr = aligned_alloc(align_to, stack_size_in_bytes);
    return aligned_stack_ptr;
}
void posix_thread_experiment() {
    void *thread_stack_mem_ptr = NULL;
    pthread_t posix_thread = {0};
    pthread_attr_t pthread_attributes = {0};
    const size_t stack_size = 256 * 1024;
    int status = pthread_attr_init(&pthread_attributes);
    if (0 == status) {
        //puts("Success: pthread_attr_init()");
        thread_stack_mem_ptr = allocate_stack(stack_size);
        status = pthread_attr_setstack(&pthread_attributes, thread_stack_mem_ptr, stack_size);
    }
    ////////////////////////////////////////////////////
    // TODO:
    ////////////////////////////////////////////////////
    // Write a pattern on all of the memory that will be used for thread stack
    // In other words, you are writing a "watermark pattern"
    if (0 == status) {
        //puts("Success: pthread_attr_setstack()");
        status = pthread_create(&posix_thread, &pthread_attributes, thread_entry, NULL);
    }
    if (0 != status) {
        puts("ERROR: Failed to create thread");
    }
}

```

```

}
// Wait for the thread to exit:
pthread_join(posix_thread, NULL);
// |----|  <- lower memory which is thread_stack_mem_ptr
// |----|
// |----|
////////////////////////////////////
// TODO:
////////////////////////////////////
// Count all the bytes with the watermark intact
uint32_t unused_stack_space = 0;
printf("Total bytes un-used: %u / %u\n", (unsigned)(unused_stack_space*sizeof(uint64_t)), (unsigned)
}
int main(int argc, char **argv) {
    //////////////////////////////////////
    // TODO
    // How much memory is the main() thread launched with?
    //uint8_t memory[8 * 1024 * 1024] = { 0 };
    //////////////////////////////////////
    posix_thread_experiment();
    return 0;
}

```

# Exercises

## Exercise 1:

1. Experiment and determine the maximum stack allocation size of the main() function
  - You can modify size of allocated memory inside main() and assess when the program crashes or experiences a segmentation fault
2. Complete the TODOs in the code to determine the stack usage of a thread
  - Paint the allocated stack memory with a watermark pattern
  - Count the number of stack memory bytes with un-altered pattern
  - Experiment with various stack memory usage of the thread to determine if your algorithm is working
  - Experiment with stack usage

- Note down how much stack space a printf() method takes

## Exercise 2:

- Create two threads that perform some operations and sleep periodically
  - Use about 128k stack space for each thread
- Create a thread monitoring thread (third thread)
  - This thread should monitor the stack usage of each thread
  - If stack free reaches < 4000 bytes, print out a warning message
  - Periodically, every 5 seconds, print out stack usage of each thread

---

Revision #4

Created 1 year ago by [Preet Kang](#)

Updated 1 year ago by [Preet Kang](#)