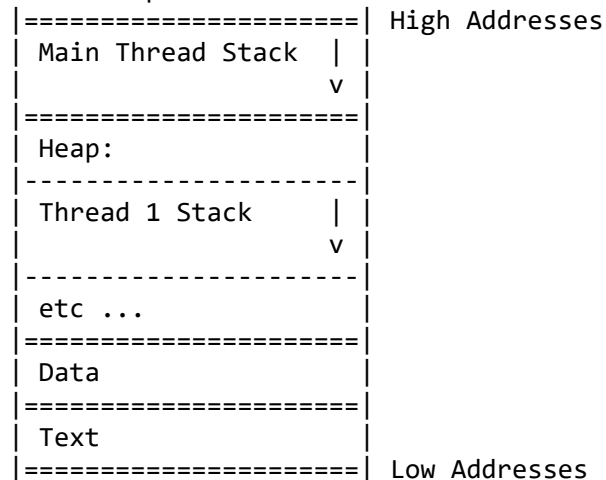


# Assignment 1: Context Switching

**Due Date: Friday, October 10th**

In this project, each thread's activation stack after the main thread will live in the heap:

Address space:



Context switching will take place by explicitly modifying the system stack pointer register (%rsp on x86-64 machines) using an assembly routine.

The goal of this assignment is to develop a C data structure representing a thread, and to develop and test the x86-64 assembly routines for starting threads and switching between them.

## Instructions

1. Write a C data structure, `struct thread_t`, to represent a thread table entry (the data pertaining to a thread). At first, it should have at least a stack pointer (perhaps of type `void*`), and a pointer to an initial function (perhaps of type `void(*initial_function)(void)`). For more on function pointer syntax, [see here](#).
2. Write an assembly routine to start a new thread, with the prototype:

```
void thread_start(struct thread_t * old, struct thread_t * new);
```

This routine should do the following:

1. Push all callee-save registers (%rbx, %rbp, %r12-15) onto the current stack.
2. Save the current stack pointer (%rsp) in old's thread table entry.
3. Load the stack pointer from new's thread table entry into %rsp.
4. Jump to the initial function of new.

For some tips on writing assembly, [see here](#).

3. Write an assembly routine to switch between two threads, with the prototype:

```
void thread_switch(struct thread_t * old, struct thread_t * new);
```

This routine is similar to `thread_start`, and should do the following:

1. Push all callee-save registers onto the current stack.
2. Save the current stack pointer in `old`'s thread table entry.
3. Load the stack pointer from `new`'s thread table entry.
4. Pop all callee-save registers from the new stack.
5. Return.

4. Test your code! Some suggestions:

- Create two global struct `thread_t` variables, `current_thread` and `stored_thread`.
- Add the following `yield` function:

```
void yield() {
    struct thread_t temp = current_thread;
    current_thread = stored_thread;
    stored_thread = temp;
    thread_switch(&stored_thread, &current_thread);
}
```

- Create a function that continuously prints a message and calls `yield()`. `current_thread` is soon to start with this function, so in `main`, set this to be the initial function of `current_thread`.
- Next, in `main`, `malloc` a suitable region for `current_thread`'s stack. Experiment with different stack sizes. Remember, stacks grow down, so the initial value of the stack pointer should be set to the `END` of the allocated range.
- Given the above, `stored_thread` represents the main thread, soon to be suspended and stashed. Its stack does not have to be allocated. Why not?
- In `main`, call `thread_start(&stored_thread, &current_thread)`.
- After `thread_start`, have `main` continuously print a message and call `yield()`.

5. Some things to think about:

- Instead of having `main` and your target function call `yield()` in an infinite loop, have them do so for a fixed number of iterations.
- Notice that if `main` finishes first, the other thread's function will not get to finish. Why is this? How could we prevent this?
- Notice that if the other thread's function finishes first, the program will crash with a Segmentation Fault. This is harder to understand. What happens when a function ends? Think about how we have constructed the thread's activation stack. What steps could we take to ensure that a thread terminates gracefully?
- Think about how to free the memory associated with a thread's stack. Simply free-ing the stack pointer after the thread has finished will not work. Why?

## What To Hand In

If you are working with a partner, please prepare just one submission for the both of you. Make sure your (and your partner's) names are included in each file you submit.

You should submit:

1. All code, including tests.
2. A brief written report, including:
  1. A description of what you did and how you tested it.
  2. Your thoughts on the questions posed in section 5, above.

Please submit your code files *as-is*; do not copy them into a Word document or PDF. Plain text is also preferred for your write-up.

Email your submission to the TA at [kstew2 at cs.pdx.edu](mailto:kstew2@cs.pdx.edu) on or before the due date. The subject line should be "CS533 Assignment 1".

## Need Help?

If you have any questions or concerns, or want clarification, feel free to [contact the TA](#) by coming to office hours or sending an email.

You may also send an email to the [class mailing list](#), but please pose discussion questions only; **do not** email any code to the class mailing list!