Operating Systems CS416, Project One: Stacks and Basics

Sankar Gollapudi (SAG341), Rohan Karamel (RAK218)

September 24, 2024

Part 1: Stack Operations Report

`stack.c` was completed by Sankar Gollapudi.

What are the contents in the stack?

The stack contains many elements, including, but not limited to, registers, variables, and arguments to pass into function calls. The stack grows downwards, and RSP (the stack pointer) points to the bottom of the stack (which is the front as the stack grows downward), while RBP (the base pointer) points to the top of the stack (or the back). This arrangement allows RSP to reset and allocate the appropriate amount of memory in the stack upon a function call.

Where is the program counter, and how did you use GDB to locate the PC?

The program counter is usually saved in the EIP/RIP, which is the instruction pointer (EIP for 32-bit and RIP for 64-bit). In this case, EIP was pushed to the stack during the 5-length instruction CALL. Using this assumption, we employed GDB's `x/10gx` around `$rbp` and `$rsp` to search for a piece of memory holding the return pointer (0x56556265).

What were the changes to get the desired result?

We first identified the location of the `stackno` on the stack (&stackno). Then, we created a pointer to this pointer so that we could offset it appropriately. Without creating a pointer to the pointer, the memory offset might affect the location of the pointer itself rather than shifting the stack location. After creating the pointer to the pointer, we offset it, dereferenced it, and modified the program counter's memory address by adding 2 to skip to the next instruction in assembly.

Part 2: Bit Operations Writeup

Both `bitops.c` and `threads.c` were completed by Rohan Karamel.

1. Extracting the Top Bits: `get_top_bits`

```
static unsigned int get_top_bits(unsigned int value, int num_bits) {
    return (num_bits > 0 && num_bits <= 32) ? value >> (32 - num_bits) : 0;
}
```

Purpose: This function extracts the most significant (top) bits from a 32-bit value.

Input:

- `value`: The 32-bit unsigned integer from which we want to extract the top bits.
- `num_bits`: The number of top bits to extract.

Operation: The function uses a right shift operation (`>>`) to shift the value by (32 - `num_bits`) positions. This effectively discards the lower bits and keeps only the top `num_bits`. For example, to extract the top 4 bits, we would shift the value by $32 - 4 = 28$ bits to the right, leaving the top 4 bits intact.

Conditions: The function ensures that `num_bits` is within a valid range (greater than 0 and less than or equal to 32). If not, the function returns 0.

2. Setting a Bit at a Specific Index: `set_bit_at_index`

```
static void set_bit_at_index(char *bitmap, int index) {
    bitmap[index / 8] = bitmap[index / 8] |= (1 << index % 8);
}
```

Purpose: This function sets a specific bit in a bitmap to 1 at a given index.

Input:

- `bitmap`: A pointer to the bitmap, which is an array of bytes (`char` array) representing bits.
- `index`: The index of the bit to set.

Operation: The function calculates which byte in the `bitmap` array contains the bit at the given `index` by performing integer division (`index / 8`). Then, it uses a bitwise OR operation (`|=`) to set the appropriate bit in that byte. The bit to set is determined by `1 << (index % 8)`, which shifts the value 1 to the correct position within the byte.

3. Retrieving a Bit at a Specific Index: `get_bit_at_index`

```
static int get_bit_at_index(char *bitmap, int index) {
    return (bitmap[index / 8] & (1 << index % 8)) != 0;
}
```

Purpose: This function retrieves the value of a specific bit in the bitmap at the given index (either 0 or 1).

Input:

- `bitmap`: A pointer to the bitmap, which is an array of bytes (`char` array) representing bits.
- `index`: The index of the bit to retrieve.

Operation: Similar to `set_bit_at_index`, the function first calculates which byte in the `bitmap` contains the bit at the given `index`. It then uses a bitwise AND operation (`&`) to isolate the specific bit by shifting 1 to the correct position (`1 << (index % 8)`). The result is checked to see if the bit is set (`!= 0`).