Session 4 Pointers & Heap

4.1 Exercises

Exercise 1 Consider the following C function, where `in1`, `in2` and `out` are all pointers to arrays of `n` integers. Translate this function to RISC-V.

```
void sum(int *in1, int *in2, int *out, int n) {
    for (int i = 0; i < n; i++) {
        *out = *in1 + *in2;
        out++;
        in1++;
        in2++;
    }
}
```

Exercise 2 The following C function compares two strings. It returns 1 if they are equal and 0 if they are not. Translate to RISC-V.

```
int streq(const char *p1, const char *p2) {
    while (*p1 == *p2) {
        if (*p1 == '\0') {
            return 1;
        }
        p1++;
        p2++;
    }
    return 0;
}
```

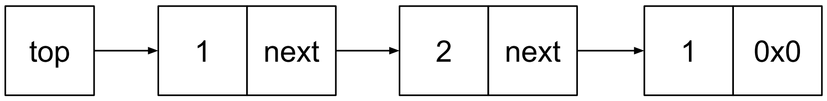

Figure 1: Representation of a stack with three elements 1, 2, 1. Every square corresponds to a 32-bit region on the heap.

Exercise 3 Create a dynamic stack data structure on the heap. Use the simple allocator below and write the following functions in RISC-V.

```
.data
heap: .space 1000000

.text
.globl main
allocate_space:
    mv t0, a0
    mv a0, gp
    add gp, gp, t0
    ret

main:
    la gp, heap
```

stack_create Creates a new, empty stack. This function allocates enough heap memory to store a pointer to the top of the stack. Since the stack is empty, have the top pointer point to the address 0. Return the address of this top pointer in $a0$. This can be considered the address of the stack.

stack_push Adds a new element to the top of the stack. Provide the address of a stack in $a0$. Provide the value to be pushed on the stack in $a1$. Allocate enough heap memory to store the new value. You will also need to allocate heap memory to store a reference to the previous top (why?). Make sure to modify the top pointer to point to your newly pushed element.

stack_pop Removes and returns the top element from a stack. Provide a stack address in $a0$. Return the popped element in $a0$. Make sure to correctly update the top pointer of the stack.

Exercise 4 Would it be possible to allocate growing data structures, like a binary tree or a linked list, on the call stack? The following code provides a suggestion for a simple allocator that tries to do exactly this. Can you see any problems with this approach?

```
allocate_stack:
    mv t0, a0
    mv a0, sp
    addi sp, sp, t0
    ret
```

Exercise 5 *Hard exercise* Can you come up with an allocator function that allows you to free previously allocated memory? The allocator should re-use previously freed memory. The following approaches might help:

1. Store the address of the first empty (free) heap region in a global variable.
2. Allocate space for metadata when creating chunks.
3. Store the size of a chunk in the chunk metadata
4. For free chunks, also store the address of the next free chunk in the chunk metadata