

Computer Systems Lab 4

Getting Started

Download the file `lab4.tgz` and unpack it. This will create a directory called `lab4` with subdirectories called `lists` and `vm24`.

Important: The two parts of this lab were made available at slightly different times. If you started Part 1 early by downloading `lab4.1.tgz`, just download and unpack `lab4.2.tgz` to do Part 2.

Part 1: VM24 Function Calls

In this part of the lab you will implement two of the three function-related instructions of the VM24 machine: `icall` and `ret`. (The direct `call` instruction will be implemented in Lab 6, after we talk about linking.) You will do this in three stages, exploring two different strategies for implementing function calls. **It is very important that you read the online slides of Lecture 7 before starting work on this part of the lab.**

As you know by now, every activation of every VM24 bytecode function has its own set of argument variables, its own set of local variables, and its own operand stack. This per-activation storage is roughly analogous to a stack frame on the x86. In the VM24 interpreter we are developing, the state of a function is captured by a structure of the type `localstate`, defined in `vmdata.h`. The components of a `localstate` are as follows:

<code>bytefunction *bf;</code>	the function this state belongs to
<code>unsigned char nargs;</code>	number of argument variables
<code>wordval *args;</code>	array of argument variables
<code>unsigned char nlocal;</code>	number of local variables
<code>wordval *locals;</code>	array of local variables
<code>unsigned char nstack;</code>	capacity of operand stack
<code>wordval *stack;</code>	operand stack
<code>int sp;</code>	operand stack pointer
<code>unsigned pc;</code>	program counter
<code>localstate *callerstate;</code>	(used for explicit interpretation of calls)

The first component of a `localstate` is a `bytefunction` (defined in the same file), which is the internal representation of a bytecode function; it has the following fields:

<code>char *name;</code>	name of the function (to use in error messages)
<code>memval *modsymbols;</code>	(leave this alone until Lab 6)
<code>unsigned char *bytes;</code>	the bytecode of this function as read from the m24 file

There are actually two different kinds of functions our VM24 implementation must be able to handle: bytecode functions, which are interpreted using `interp_bytecode`, and native functions,

which are library routines like `puts` that are written in C. When your interpreter encounters an `icall` instruction, the value on top of the operand stack will be a reference to a function, but you will have to examine the function to determine which kind it is.

Recall that operands on the stack have the type `wordval`, a union that may be either an integer or a `memval_payload`. In turn, `memval_payload` is a union that may be, among other things, a pointer to a `vmfunction`. A `vmfunction` is a structure that can represent a bytecode function or a native function, and it has a `type` field that indicates which. So, when interpreting an `icall` instruction, if

```
stack[sp].refval.function.type
```

is equal to `FUNC_BYTE` then

```
stack[sp].refval.function->func.bytefunc
```

is a valid `bytefunction`. Otherwise, the callee is a native function. For your convenience, I have provided a function called `call_vm_function` which does the right thing for native functions and calls back to your code for bytecode functions.

Getting Started

Copy your code from Lab 3 into the new `machine.c` from the Lab 4 distribution. Note that a new function, `call_bytecode_function`, has been added to this file.

Phase 1: Meta-Circular Implementation

The provided code for this lab uses the meta-circular implementation of `call` by default. (The metacircular implementation of `ret` has always been in plain view in `machine.c`.) When the interpreter encounters a `call` to a bytecode function, it calls `call_bytecode_function` with a pointer to the `bytefunction` being called, the number of arguments given, and an array containing the arguments themselves.

You must do two things in order to get the interpreter working with the metacircular implementation of calls:

1. Complete the implementation of `call_bytecode_function` so that it creates a `localstate` for the new function activation and calls `interp_bytecode` with that new state. For reasons that will become clear in Lab 5, **you should not use `malloc` or `free` to create or destroy the local state**. Instead, use the `localstate_alloc` and `localstate_free` functions I have provided. These functions are declared in `lab-machine.h`. Furthermore, you should not use `malloc` to create a new `args` array for the local state; it is okay to use the array passed to `call_bytecode_function`.
2. Implement the `icall` instruction. The encoding of this instruction consists of the opcode byte followed by one additional byte that gives the number of arguments to be passed to the function. As you have seen in the lecture notes (read them now if you have not yet!), the value

on top of the stack will be a reference to the function being called. Below that will be the arguments, if any, pushed in their natural order. All you have to do is decode the instruction, pop the function reference and arguments from the stack, and call `call_vm_function`; for bytecode functions, this will result in a call to `call_bytecode_function`.

Phase 2: Explicit `call` and `ret`

I have provided an explicit, non-meta-circular implementation of the `call` instruction. Switch to using it by uncommenting the commented-out lines in the case for `CALL` in `machine.c`. When you recompile after this change, programs that contain `call` instructions will crash when one of those functions tries to return.

The implementation of `ret` must now deal with the fact that there are two different ways the current function might have been called. It might have been called through `call_bytecode_function` (`main` is called this way, as is anything called with an `icall` that you interpret), or it might have been called with a `call` instruction. In the former case, the `callerstate` field of the current local state will be `NULL`; in the latter case, it will be a pointer to the `localstate` that was in effect when the `call` was executed. Add code to the implementation of `ret` to handle this new possibility.

The `pc` field of the caller state will point to the instruction following the call, and the `sp` field will reflect the arguments having been popped.

Phase 3: Explicit `icall`

Now that you have an implementation of `ret` that is consistent with the explicit implementation of `call` I have provided, replace your meta-circular implementation of `icall` with an explicit one, using the sketch in the lecture notes as a guide. You will have to look at the `type` field of the function to determine whether it is native or bytecode; you should still call `call_vm_function` for native functions, but you should **not** do so for bytecode functions.

Testing

The only test program I have provided so far that uses the `icall` instruction is `spoints`. To test your implementation thoroughly, you should probably write some test code of your own.

In addition, take a look at the file `irecurese.a24` in the Lab 4 distribution. This simple program will behave rather differently under meta-circular and non-meta-circular implementations of `icall`. Do you understand why?

Part 2: Linked Lists in Assembly

In this part of the lab, you will write two functions in IA-32 assembly that work with linked lists of the type defined in the file `list.h`.

Reversing a List

Translate the following C function into assembly; put your code in `reverse.s`.

```
list reverse(list L) {
    list tail = NULL;
    while (L != NULL) {
        list nextL = L->next;
        L->next = tail;
        tail = L;
        L = nextL;
    }
    return tail;
}
```

This function reverses a linked list in place by reversing all the links, and returns a pointer to what used to be the last node and is now the first.

Copying a List

Translate the following C function into assembly; put your code in `copylist.s`.

```
list copylist(list L) {
    if (L == NULL) return NULL;
    list result = (list) malloc(sizeof(struct listnode));
    list M = result;
    while (1) {
        M->data = L->data;
        if (L->next == NULL) break;
        M->next = (list) malloc(sizeof(struct listnode));
        M = M->next;
        L = L->next;
    }
    M->next = NULL;
    return result;
}
```

This function makes a copy of a list. Do not worry about running out of memory.

Compiling and Testing

The file `test-lists.c` contains a testing driver for the `reverse` and `copylist` functions; type `make` to compile it. Then the command

```
./test-lists n
```

will test both of your functions on a list of length n . Make sure your handles the case where $n = 0$.