NTNU | Norwegian University of Science and Technology

Operating Systems

Q&A for PE5 – 09.04.2021

Michael Engel

PE5: Clean up task

- Mismatch regarding what function we should implement in the clean up task.
- In the exercise document the task says:
  - Implement the function destroy_symtab to remove dynamically allocated symbol table data at the end of the compilation.
- While in the guideline presentation it is
  - Destroy the whole structure that you created when it is no longer needed
  - implement the destroy_subtree function to do this
  - The exact implementation depends on your implementation
- Is there a typo in the guidelines presentation, or are we supposed both modify the destroy_subtree function and implement the destroy_symtab function? Alternatively, is the intended meaning something else?
- That was a typo, just implement destroy_subtree

PE5 - Initialization of global symbol table

- The guidelines (task 1) say that
  - The skeleton already initializes a global symbol table (global_names).
- However, the skeleton file does not seem to contain any calls to the function tlhash_init (where I would assume this initialization would take place).
- How/where does the initialization happen?

- in find_globals (called from create_symbol_table)
- Whenever a new scope is generated
  - e.g. in a function push_scope to build the scope stack

PE5 - tlhash_init

- How many buckets are we supposed to use in the tlhash_init function?

- Pick a number that's in the ballpark of the number of entries you expect.
- A power of two might be nice :)

PE5 - Multiple Questions

1. What exactly are sequence numbers used for? Apparently, they are not needed for global variables: Should we set them to 0 then? Are they needed for all of the following: functions, parameters and local variables?

They are e.g. used to calculate memory addresses (or offsets on the stack)

2. Regarding nested scopes and the proposed solution to avoid name clashes: Should we implement our own stacks?

Yes. You can e.g. implement the stack as a linked list with each level of the stack separately malloc'ed on the heap.

3. Regarding the "linking of names to symbols": I guess that's what "struct s *entry" in "node_t" is for?

Yes, struct s is the data structure for a (general) symbol table entry.

PE5 - Multiple Questions

4. In the hints, a "correct order to count" is mentioned, but not explained. This order is probably obvious once the exercise is solved, but not as obvious upfront. Could another hint in regards to these numbering schemes be given, especially since we are ask to not invent alternatives.

I would expect the order to derive more or less automatically due to the implementation of your parser and the way syntax tree nodes are traversed.

So just avoid anything unusual :).

PE5 - What function should look at the function parameters?

It is a bit unclear to me whether we are supposed to look at the function parameters in find_globals() or in bind_names().

Based on the task sheet, I would think that we should do it in bind_names(), as task 5.2 says:

Implement the function bind_names in ir.c to populate the local symbol tables with
• Parameters
• Local variables

However, from watching the guidelines video, I got the impression that it was recommended to fill the function name table with the parameter names already in find_globals(). I think the reasoning was that it would be convenient because this would save us a second pass.

TL;DR: In what function should we fill the function name tables with parameters?

- I opted to detect functions in find_globals(), and process functions in bind_names(). In my current working version of the PE5 delivery I call bind_names() when I have detected a function node in find_globals() and pass the symbol_t for that function node and the node_t of the function node itself to bind_names(). This gives me correct printout when using the skeleton code's provided symbol table print function.

PE5 - n_string_list

What is the size_t n_string_list meant to be used for?

- It is used for the capacity of string_list. It is set in vslc.c:
- size_t n_string_list = 8; // Initial string list capacity (grow on demand)
- When stringc equals n_string_list you have to resize string_list.

PE5 - String indices

Our tree dump of the file easy.vsl looks similar to the example tree dump, except that the strings in the string table have different indices. In particular, our string table starts to list the strings from the second function, then it lists the strings from the first function, while the example tree dump does the opposite.

Is this ok, and something that is due to the hashing, or is this one of the things that had to be equal to the examples?

Example string table:

```
String table:
0: "Testing plain call/return and expression evaluation"
1: "The deftion returned y:="
2: "My parameters are a:="
3: "and b:="
4: "Their sum is c:="
5: "Their difference is c:="
6: "Their product is c:="
7: "Their ratio is c:="
8: "(-c):="
9: "The sum of their squares is "
```

Our string table:

```
String table:
0: "My parameters are a:="
1: "and b:="
2: "Their sum is c:="
3: "Their difference is c:="
4: "Their product is c:="
5: "Their ratio is c:="
6: "(-c):="
7: "The sum of their squares is "
8: "Testing plain call/return and expression evaluation"
9: "The deftion returned y:="
```

- Differing string table indexes should not be a problem.

PE5 - Function declarations and header files

We are having trouble understanding when we should declare functions in header files and when we should declare them just in the top of the c file.

First we thought that the externally used functions should be in the header file. However, this does not seem to match with the skeleton file, as there is a part of the vslc.c with something that seems like declarations that should be used externally.

```
/* External interface */
void create_symbol_table(void);
void print_symbol_table(void);
void print_symbols(void);
void print_bindings(node_t *root);
void destroy_symbol_table(void);
void find_globals(void);
void bind_names(symbol_t *function, node_t *root);
void destroy_symtab(void);
```

- There are many different approaches to header file organization. Internally you have to decide on an approach that works well for your use case, for example a header file per C source file with it's related type definitions and non-internal functions, or maybe a single header file with all your functions and type definitions. When making a library you should in most cases present a single header file to the consumer, like is done with tlhash in this project. The same approach can be taken to modularize large projects.

PE5 - destroy_symtab()

Is there any way to check that our destroy_symtab function works correctly?

- You can use vallgrind to check that the memory is no longer in use once the process terminates valgrind --tool=memcheck --leak-check=full --show-leak-kinds=all --track-origins=yes <path_to_.vsl>
- It should be noted that if you have not taken any means to remedy it, yacc will have some memory still referenced upon termination, it appears as though it will consistently have 16,458 bytes in 3 blocks, so the output from vallgrind should be

  ==301033== LEAK SUMMARY:
  ==301033== definitely lost: 0 bytes in 0 blocks
  ==301033== indirectly lost: 0 bytes in 0 blocks
  ==301033== possibly lost: 0 bytes in 0 blocks
  ==301033== still reachable: 16,458 bytes in 3 blocks
  ==301033== suppressed: 0 bytes in 0 blocks

- Edit: This also assumes you free the static strings, I'm not 100% sure if we are supposed to do that.

PE5 - Hashtable returns broken keys?

While destroying the symbol table, we ask for all the hashtable's keys like so:

```
size_t symtable_size = tlhash_size(symtab);
char **st_keys = (char **) calloc(symtable_size, sizeof(char *));
tlhash_keys(symtab, (void **)st_keys);
```

We would then like to loop over every key of the hashtable to retrieve every symbol in the table and free what needs freeing. But this fails pretty fast, because it turns out that the hashtable seems to somehow screw up the retrieved keys. For example, the locals.vsl program produces the following keys:

```
p st_keys[0]
$3 = 0x5560adf985f0 "j\205\371\255`U"

p st_keys[1]
$4 = 0x5560adf988f0 "i\210\371\255`U"

p st_keys[2]
$5 = 0x5560adf99690 "nesting_scopes"

p st_keys[3]
$6 = 0x5560adf98490 "k"
```

PE5 - Hashtable returns broken keys?

Which causes a segfault because we try to perform a hashtable lookup with these keys, which of course fails and returns NULL. Everything works fine with tlhash_values, but it is not useful because it doesn't allow us to remove the entries from the hashtable, which has allocated memory for the keys...

To clarify, this only happens when you ask for keys with tlhash_keys. Everything still works if we attempt a lookup with something like:
tlhash_lookup(symtab, "j", 1, symbol_out);
So the keys are still "intact" internally.

- You can get every value directly with tlhash_values, free those, and then finalize the symtab with tlhash_finalize.
- As for finding out the error in your current approach, I would need to see the code for the failing lookup.