G1 - report

Ask Neve Gamby & Maya Saietz

February 13, 2014

1 Doubly-linked list

We have designed the doubly linked list based on using a single pointer as a seed for bijective mapping by exclusive or. We exploit the following form to find the targeted seed for the exclusive or:

$$left \oplus seed = right \wedge right \oplus seed = left \tag{1a}$$

$$left \oplus left \oplus seed = left \oplus right \tag{1b}$$

$$\bar{0} \oplus seed = seed = left \oplus right \tag{1c}$$

That is, for a node `x`, we find `x->ptr` by xor'ing the pointers to the elements on each side. This works as long as there is an element on each side, but we have yet to decide what to do with the ends of the list, and with the empty list.

For the empty list, we decided that the `head` and `tail` pointers in the `dlist` struct would both be `NULL`. We also considered adding an integer member to the struct, which would denote the size of the list, but decided that that wouldn't be a very elegant solution[1].

For the ends of a non-empty list, we had the option of making it cyclic (by xor'ing the ends together) or noncyclic (by xor'ing with 0). We decided on making it noncyclic, as making it cyclic was not a part of the assignment (and it works nicely with how xor behaves).

Moving through the list is done as follows: If we're at the beginning of the list, then the node's `ptr` simply points to the next node. If not, then we xor the current node's `ptr` with the pointer to the previous node in order to get the pointer to the next node.

`init`

We have added an `init` function to the header file – it takes a newly declared `dlist`, and initialises both members to `NULL`.

[1] It would also use an entire four bytes more! Not space-effecient at all.

`insert` and `extract`

`insert` and `extract` are both $O(1)$. For lists where there is more than one item (or in the case of `insert`, at least one item), they use `atTail` to get a pointer to the "first" node in the list. They both have a special case for the empty list – here, `extract` returns `NULL`, while `insert` sets the `head` and `tail` pointers to point to the new node.

`extract` also has a special case for the list of one item, where it returns that item, frees the `node` and sets the `head` and `tail` pointers to `NULL`.

`reverse`

`reverse` is $O(1)$. All it does is switch the `head` and `tail` pointers in the `dlist` – it doesn't touch the `node`s. This works because all we need to know in order to get to the next node, is the address of the previous node. It doesn't matter if "previous" means up or down in memory.

`search`

`search` uses a `while`-loop on "the next node we want to look at", with the condition that this node is not `NULL`. This means that as soon as we reach an end node, and xor's it with the pointer to the previous node, we get `NULL`, and the loop ends.

For each node, we simply run the given function. If it returns `true`, we return the item. If no item is found, we return `NULL`.

2 Basic I/O

For the basic read and write functionality, we simply inserted more cases in `proc/syscall.c` for different syscalls, which calls the appropriate functions in `kernel/io.c`[2]. These cases simply sends the call onwards to the the generic character device driver, assuming that we can find the driver and that more than 0 characters are to be written or read.

For testing, we found it interesting to write a small library for handling strings and standard I/O (found in `tests/libio.c` and `tests/libio.h`). Since our library allows for us to read a line (while showing it), create a prompt, write until string termination, and compare strings, it was easy for us to make a program that would keep prompting the user and echoing the input back, until he quit the program with a quit command.

[2]Declared in `kernel/io.h`