# BUS Exercise 2

Group 23


Thilo Metzlaff
406247

Mats Frenk
393702

René van Emelen
406008


May 11, 2020

# Contents

# Introduction

Welcome to this special endeavour, where I write every single thing in LaTeX How is any of this gonna work?
Simple: With patience and a format definition. Therefore, please let me define the format of this.

## THE FORMAT

- Every file will be named similar to the sections in here, so `2.1-stack_exercise.c` is Exercise 2, section 1.
- Every Solution **WILL** be in this pdf, but not necessarily anything predefined by the exercise.
- Any explanation will be both in this PDF as well as in each code file, provided by comments.
- This explanation will be in each PDF, in case someone who doesn't know the format tries to correct the exercises.

# 1 Stack Exercise

To create a stack, one needs three things:

- A root node, which is a pointer to the last element added to the stack or **NULL**, if the stack is empty.
- A `StackNode`, which has content and a pointer to the previously added node, or **NULL**, if it's the last element
- Data for the `StackNode` to point to.

Since the Exercise already predefined the Structure and creation of the `StackNode`, we didn't have to make that, but we *DO* have to define the behavior of the `push` and `pop` functions

## 1.1 a - The `push()` Function

The `push` function takes an element and pushes it onto the stack. Pushing to a stack has 2 cases we have to handle:

- The stack is empty, making the `root pointer` point to **NULL**
- There is at least one item on the stack, making the `root pointer` point to the last item.

Therefore we first have to test wether `root` points to **NULL**. In the first case, we only have to dereference the `root pointer` and point it to a newly created `StackNode`, due to the way `StackNode` is defined.
The second case has some more actions, but it's not a lot more complicated. Here are the steps:

1. create a new `StackNode` and call it `temp`
2. point the `next_node` pointer of the `temp StackNode` to the last element of the stack
3. dereference `root` and point it to `temp`

```
void push(StackNode **pointer2root, char *command)
{
  // YOUR SOLUTION GOES HERE ...
  if (*pointer2root == NULL)
  {
    // just set a new node, there is nothing on the stack
    *pointer2root = newNode(command);
  }
  else
  {
    StackNode *temp = newNode(command); // get a new node
    temp->next_node = *pointer2root;      // move node pointer
    *pointer2root = temp;                 // move root pointer
  }
}

```

Listing 1: The `push()` Function

## 1.2 b -The `pop()` Function

The `pop` function pops an element from the top of the stack and makes its data (in our case a `string`) available to the program. When trying to pop off an element we can encounter two cases:

- The stack is empty
- The stack has at least one element

In the first case, we can return **false**. If we do have an element, we have to do the following, before returning **true**:

1. get the last node from the stack
   `StackNode *temp = *pointer2root;`
2. get the next node and point to it in `root`
   `*pointer2command = temp->command;`
3. give a pointer to the data to the user
   `*pointer2root = temp->next_node;`
4. free the memory space that is now unused
   `free(temp);`

```
bool pop(StackNode **pointer2root, char **pointer2command)
{
  // YOUR SOLUTION GOES HERE ...
  if (*pointer2root == NULL)
  {
    return false; // nothing on the stack
  }
  else
  {
    StackNode *temp = *pointer2root; // get current stack element
    *pointer2command = temp->command; // set command pointer
    *pointer2root = temp->next_node; // move root pointer
    free(temp); // free memory

    return true; // something was indeed popped off
  }
}

```

Listing 2: The `pop()` Function

# 2 Bash Scripting

## 2.1 a - Replacing the first Occurence of a Character

To replace things, we can simply use `sed`. Here we need to replace the first `1` with a `2`. the Regex for this is 's_1_2_1'.
To do this easily, we can create a script file, which takes a string as its argument and pipes it into `sed`, like this:

```bash
#!/bin/bash

echo "$1" | sed 's_1_2_1'

```

## 2.2 b - Watching a Process

To list all processes, we can use `ps`. To get the process with the specified PID, we pipe its output into `grep`. we then use a `while` loop, checking wether our listing has vanished. While in that loop, we echo `"process ${PID} running"`. When the loop exits, we echo `"Process not running."`. And most importantly, we test if the user has specified **BOTH** arguments. The following piece of code accomplishes all of that.

```bash
#!/bin/bash

# test if argumaents were given
if [[ -z $1 ]] | [[ -z $2 ]]; then
  echo "The syntax is: ${PWD} {PID} {TIME}"
  exit
fi

PROCESS="$(ps | grep ${PID})"

# while process is running, print "Process {PID} running"
while [[ -n $PROCESS ]]; do
  echo "process ${1} running"
  sleep $2
  PROCESS="$(ps | grep ${1})"
done

echo "Process not running"

```

Listing 3: The Bash code that keeps on giving

## 2.3 c - Bash Code Analysis

We were given the following code (it has been prettyfied for readability):

```
1 S=0
2 for f in $(find . -name "*.c"); do
3   S=$( ( $S + $( wc -l $f | awk '{ print $1 }' ) ) )
4 done
5 echo $S
6
```

I will now give a line-by line explanation of each command and action that happens.

1 - Set variable `S` to `0`

2 - Get all names of C source files and loop through them

- `$( find . -name "*.c" )` → get all filenames of C source files from the current directory
- `for f in $(expr); do` → loop through all filenames

3 - Add the number of lines of the current file to S

- `$(wc -l | awk '{ print \$1 }' )` → get the number of lines in `f`
- `$(( S + $(expr) ))` → add the values of S and expr
- `S=$(expr)` → set S to the value of expr

4 - The end of the `for-loop`

5 - echo `S`, the total combined length of all C source files.

## 2.4 d - Recursive File Listing

To list Files recursively we only need a function that lists files recursively and an input filter. Filtering the input is quite easy, we test if `$1` is set. If it isn't, we take the current directory and feed it to the function, otherwise we take the user's input and feed that to our function.
This function needs to keeep track of where it was executed, so we can use `$PWD`. It also needs to keep track of all files in the current directory, as well as all directories, for which we can use `find` with the type argument.
The last thing we need is loops and recursive calls, and indent modification. But to make all things easier, we first set `IFS` to only use newline, or this will not work with directories that have spaces in them. Now that we have defined all the things needed, here's the code:

```bash
#!/bin/bash
function list_dirs() {
DIRLIST=$(find $1 -maxdepth 1 -type d) # get all subdirectories.
FLIST=$(find $1 -maxdepth 1 -type f)   # get all files in directory


IFS=$'\n'
# open all directories
for i in $DIRLIST; do
  # are we trying to open ourselves?
  if [[ $1 != $i ]]; then

    # there's some weirdness going on
    printf "%$(($2 + 2))s\e[1m\e[4m\e[38;5;6mDIRECTORY:\e[0m $(echo $i) \n"

    # recursive open
    list_dirs "${i}" $(($2 + 2))
  else

    # if we are in the root directory, we can print that, otherwise we don't care
    if [[ $i == $PWD ]] | [[ $i == "./" ]]; then
      printf "\n%${2}s\e[1m\e[4m\e[38;5;6mDIRECTORY:\e[0m $(echo $PWD | sed 's_^.^/\b__') \n"
    fi
    # list all files in directory
    for j in $FLIST; do
      # if it's not empty
      if [[ $j != -z ]]; then
      # print the current file
        printf "%$(($2 + 2))s\e[1mFILE:\e[0m \e[93m$(echo $j)\e[0m \n"
      fi
    done
  fi
done
}

if [[ -z $1 ]]; then
  list_dirs $PWD 0
else
  list_dirs $1 0
fi
```

# 3 Syscalls and Strace

## 3.1 a - The Nature of Syscalls

A system call is an interface between a given application and the Linux kernel. Using a system call, one can issue commands to the kernel and communicate with it. One can also use these calls to access system resources.

## 3.2 b - Describing specific Syscalls

`accept` - Extracts the first pending connection request from the queue of pending connections for the listening socket.

`brk` - Changes the location of the program break, letting a program allocate or deallocate memory, depending on wether the break is increased (allocation) or decreased (deallocation).

`mmap` - Creates a new mapping for a file or directory in the virtual address space.

`open` - Opens or creates a file specified by pathname, depending on its existence and the `O_CREAT` flag.

`write` - Writes up to `count` bytes from the buffer, starting at `buf` to the file descriptor `fd`

## 3.3 c - The Nature of Strace

`strace` is a command to trace syscalls and signals from a specified command. It runs a command until it exits, intercepting and recording all syscalls and signals of the process.

## 3.4 d - Stracing the ls command

`ls /etc` lists all subdirectories of `/etc`, needing two different syscalls, namely `fstat`, which gets information about an open file and `open` or `openat`, which open files at a specified path.
`ls -la` also prints the access permissions, link count, owner, group, file size, mlast modify date and filename. For this it needs **A LOT** more syscalls, because it also follows symlinks and reads more data. This makes it take slightly longer, but still uses the same syscalls.

## 4 Bash Text Processing

There's really not that much to describe here, so descriptions will be quite short.

### 4.1 a - Listing Files and Directories with Size

There actually already is a command for this, which is `ls`, it just adds a total size, which we can remove with grep. The full command then looks like this:

```
ls -sh | grep -ve "^[total]"
```

### 4.2 b - Switching Size and Name of a's Output

To do this, we only have to pipe to the prevous' command's output into awk and print the columns switched, like this:

```
ls -sh | grep -ve "^[total]" | awk '{ print $2, $1; }'
```

### 4.3 c - Processing Text File and Groupings

As I didn't read the exercise properly, I assumed that I could just list all things in a nice format and be done. I then *quickly* realized that this was actually the next exercise... So I just modified the command from the next subsection. Getting to the actual command, I used `sort teamnamen.txt` to first get a sorted list, because I then piped it into `uniq -c`, which gives a counted list of unique elements. After that, it is piped into `awk {print $1}'`, to strip the group names, then numerically sorted with `sort -n`, only to be piped into `uniq -c` again, so we can get a nicer output. The last part then gets piped into `head -n 1` and then to `tail -n 1` so it only gives us the line we want. Changing the input to `head -n 1` can change that output to be either a group of 1-3 people, or the nullgroup by using a value of 1-4. Now getting back to showing the actual monstrosity:

```
sort teamnamen.txt | uniq -c | awk '{print $1}' | sort -n | uniq -c | head -n 1 | tail -n 1
```

### 4.4 d - More Text Processing and grouping

Since i already explained everything up to the last `awk` command in the previous, i will not explain it again. The Last command, without `echo`, `head` and `tail` is piped into awk to pretty print the output, making it into the following monstrosity:

```
sort teamnamen.txt | uniq -c | awk '{print $1}' | sort -n | uniq -c | awk '{if ($2!="1" && $2!="2" && $2!="3")
 print "\n"$2, "Nullgroup"; else print $1, $2}'
```