

## Problem Sheet #1

### Problem 1.1: *strsplit* crash

(2 points)

A freshmen is learning the C programming language. He wrote the following program but it keeps crashing or producing unexpected outputs.

```
1  #define _DEFAULT_SOURCE
2
3  #include <string.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  /*
8   * Count the number of characters in string that are an element of the
9   * character set delim. Returns 0 if none of the characters in string
10  * is in the character set delim or the string is empty.
11  */
12
13  size_t strcnt(const char *string, const char *delim)
14  {
15      size_t cnt = 0;
16
17      for (const char *s = string; *s; s++) {
18          for (const char *d = delim; *d; d++) {
19              if (*s == *d) {
20                  cnt++;
21                  break;
22              }
23          }
24      }
25      return cnt;
26  }
27
28  /*
29   * Split the string whenever a character appears that is in the
30   * character set delim. Return a NULL terminated vector of pointers to
31   * the sub-strings.
32   */
33
34  char ** strsplit(char *string, const char *delim)
35  {
36      char *token;
37      size_t cnt = strcnt(string, delim);
38
39      char **splitv = calloc(cnt + 1, sizeof(char));
40      if (splitv) {
41          for (int i = 0; (token = strtok(&string, delim)); i++) {
42              splitv[i] = token;
43          }
44      }
45      return splitv;
46  }
47
48  int main(int argc, char *argv[])
49  {
50      for (int i = 1; i < argc; i++) {
```

```

51     char **splitv = strsplit(argv[i], " ");
52     if (splitv) {
53         for (int j = 0; splitv[j]; j++) {
54             (void) puts(splitv[j]);
55         }
56         (void) free(splitv);
57     }
58 }
59
60 return EXIT_SUCCESS;
61 }

```

- a) Explain why the program crashes or produces unexpected outputs.
- b) How can the program be fixed?

**Problem 1.2:** *memory segments (strndup)*

(2 points)

Look at the following program and write down what is stored in the text segment, the data segment, the heap segment, and the stack segment.

```

1  #include <stdlib.h>
2  #include <string.h>
3  #include <stdio.h>
4
5  char *strndup(const char *s, size_t n)
6  {
7      char *p = NULL;
8
9      if (s) {
10         size_t len = strlen(s);
11         if (n < len) {
12             len = n;
13         }
14         p = (char *) malloc(len+1);
15         if (p) {
16             strncpy(p, s, len);
17         }
18     }
19     return p;
20 }
21
22 int main(void)
23 {
24     static char m[] = "Hello World!";
25     size_t len = strlen(m);
26     for (size_t n = 1; n <= len; n++) {
27         char *p = strndup(m, n);
28         if (! p) {
29             perror("strndup");
30             return EXIT_FAILURE;
31         }
32         if (puts(p) == EOF) {
33             perror("puts");
34             return EXIT_FAILURE;
35         }
36         free(p);
37     }
38     if (fflush(stdout) == EOF) {
39         perror("fflush");
40         return EXIT_FAILURE;

```

```

41     }
42     return EXIT_SUCCESS;
43 }

```

**Problem 1.3:** *execute a command in a modified environment or print the environment* (6 points)

On Unix systems, processes have access to environment variables that can influence the behavior of programs. The global variable `environ`, declared as

```
extern char **environ;
```

points to an array of pointers to strings. The last pointer has the value `NULL`. By convention, the strings have the form “name=value” and the names are often written using uppercase characters. Examples of environment variables are `USER` (the name of the current user), `HOME` (the current user’s home directory), or `PATH` (the colon-separated list of directories where the system searches for executables).

Write a program `env` that implements some of the functionality of the standard `env` program. The syntax of the command line arguments is the following:

```
env [OPTION]... [NAME=VALUE]... [COMMAND [ARG]...]
```

- a) If called without any arguments, `env` prints the current environment to the standard output.
- b) If called with a sequence of “name=value” pairs and no further arguments, the program adds the “name=value” pairs to the environment and then prints the environment to the standard output.
- c) If called with a command and optional arguments, `env` executes the command with the given arguments.
- d) If called with a sequence of “name=value” pairs followed by a command and optional arguments, the program adds the “name=value” pairs to the environment and executes the command with the given arguments in the modified environment.
- e) If called with the option `-v`, the program writes a trace of what it is doing to the standard error.
- f) If called with the option `-u name`, the program removes the variable `name` from the environment.

Here are some example invocations:

```

$ env # print the current environment
$ env foo=bar # add foo=bar and print the environment
$ env -u foo # remove foo and print the environment
$ env date # execute the program date
$ env TZ=GMT date # add TZ=GMT and execute the program date
$ env -u TZ date # remove TZ and execute the program date
$ env -u x a=b b=c date # remove x, add a and b, execute date

```

Hand in the source code of your `env` program. Make sure that your program handles *all* error situations appropriately. Use the `getopt()` function of the C library for parsing command line options. Furthermore, use one of the `exec` system calls like `execvp()` to execute a command. (Using `system()` can be made to work but it is somewhat difficult to get right since concatenating strings using space characters may lead to surprises if the strings themselves contain space characters; to do this correctly, you have to quote the strings such that the shell called by the `system()` library function tokenizes the string properly again. Naive concatenation usually leads to a security weakness, it is often better to avoid the `system()` library function. See also the Caveats section in the Linux manual page describing the `system()` library function.)