Operating Systems
Constructor University
Dr. Jürgen Schönwälder

Module: CO-562
Date: 2023-09-08
Due: 2023-09-15

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.

```c
#define _DEFAULT_SOURCE

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

/*
 * Count the number of characters in string that are an element of the
 * character set delim. Returns 0 if none of the characters in string
 * is in the character set delim or the string is empty.
 */

size_t strcnt(const char *string, const char *delim)
{
    size_t cnt = 0;

    for (const char *s = string; *s; s++) {
        for (const char *d = delim; *d; d++) {
            if (*s == *d) {
                cnt++;
                break;
            }
        }
    }
    return cnt;
}

/*
 * Split the string whenever a character appears that is in the
 * character set delim. Return a NULL terminated vector of pointers to
 * the sub-strings.
 */

char ** strsplit(char **string, const char *delim)
{
    char *token;
    size_t cnt = strcnt(*string, delim);

    char **splitv = calloc(cnt + 1, sizeof(char));
    if (splitv) {
        for (int i = 0; (token = strsep(string, delim)); i++) {
            splitv[i] = token;
        }
    }
    return splitv;
}

int main(int argc, char *argv[])
{
    for (int i = 1; i < argc; i++) {
```

```c
        char **splitv = strsplit(&argv[i], " ");
        if (splitv) {
            for (int j = 0; splitv[j]; j++) {
                (void) puts(splitv[j]);
            }
            (void) free(splitv);
        }
    }

    return EXIT_SUCCESS;
}
```

a) Explain why the program crashes or produces unexpected outputs.

b) How can the pogram 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.

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

char *strndup(const char *s, size_t n)
{
    char *p = NULL;

    if (s) {
        size_t len = strlen(s);
        if (n < len) {
            len = n;
        }
        p = (char *) malloc(len+1);
        if (p) {
            strncpy(p, s, len);
        }
    }
    return p;
}

int main(void)
{
    static char m[] = "Hello World!";
    size_t len = strlen(m);
    for (size_t n = 1; n <= len; n++) {
        char *p = strndup(m, n);
        if (! p) {
            perror("strndup");
            return EXIT_FAILURE;
        }
        if (puts(p) == EOF) {
            perror("puts");
            return EXIT_FAILURE;
        }
        free(p);
    }
    if (fflush(stdout) == EOF) {
        perror("fflush");
        return EXIT_FAILURE;
```

```
    }
    return EXIT_SUCCESS;
}
```

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.)