Operating Systems
Constructor University
Dr. Jürgen Schönwälder

Module: CO-562
Date: 2024-09-05
Due: 2024-09-13

Problem sheet #01
Erza Dauti

Problem 1.1: *library and system calls* (1+1 = 2 points)

Answer the following questions by using strace and ltrace on a Linux system. Provide enough context information to make it clear how the results were obtained.

a) How many system calls and how many library calls does executing /bin/date produce?

- **-System calls (using `strace`) :**

To capture the number of system calls, we ran the following command:

```
~$ strace /bin/date
```

This output showed that **118 system calls** were made during the execution of `/bin/date`.

- **Library calls (using `ltrace`) :**

To trace the number of library calls, we used:

```
~$ ltrace /bin/date
```

This showed that **47 library calls** were made during the execution.

b) What are the most frequent (top three) library and system calls and what do these calls do?

Top Three System Calls (via `strace`):

1. `openat()`: **31 calls**
2. `fstat()`: **21 calls**
3. `close()`: **21 calls**

Top Three Library Calls (via `ltrace`):

1. `fwrite()`: **8 calls**
2. `fputc()`: **7 calls**
3. `fread()`: **4 calls**

Problem 1.2: *system call errors* (1+1 = 2 points)

System call errors are usually indicated by returning a special value (usually -1 for system calls that return an int) and by indicating the details in the global variable int errno, declared in errno.h.

a) For each of the following system calls, describe a condition that causes it to fail (i.e., a condition that causes -1 to be returned and that sets errno to a distinct value).

- int open(const char *path, int oflag, ...)

- [ENXIO]

The named file is a character special or block special file, and the device associated with this special file does not exist.

- int close(int fildes)

- [EINTR]

The *close*() function was interrupted by a signal.

b) What is the value of errno after a system call is completed without an error?

- When a system call completes without an error, the value of errno remains unchanged.
By default, the value of errno is 0, and it will stay at 0 if no error occurs during a system call.

`errno` is only updated when a system call fails, in which case it is assigned a non-zero value that represents a specific error code

Therefore, if a system call completes without errors, `errno` remains at 0 or retains its previous value if there were no prior errors.

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 argu- ments, the program adds the “name=value” pairs to the environment and executes the com- mand 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 weak- ness, 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.)

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


extern char **environ;

// Print environment variables
void show_environment() {
    char **env_ptr = environ;
    while (*env_ptr) {
        printf("%s\n", *env_ptr);
        env_ptr++;
    }
}

int main(int arg_total, char *arg_list[]) {
    int flag;
    int remove_flag = 0, verbose_flag = 0;

    opterr = 0;
    while ((flag = getopt(arg_total, arg_list, "vu:")) != -1) {
        switch (flag) {
        case 'v':
            verbose_flag = 1;
            break;
        case 'u':
            if (optarg[0] == '-') {
                fprintf(stderr, "Invalid -u option argument: '%s'!\n", optarg);
                return EXIT_FAILURE;
            }
            remove_flag = 1;
            if (unsetenv(optarg)) {
                perror("unsetenv");
                return EXIT_FAILURE;
            }
            break;
        default:
            fprintf(stderr, "Usage: %s [-v] [-u name] [name=value]... [command [arg]...]!\n", arg_list[0]);
            exit(EXIT_FAILURE);
            break;
        }
    }

  //Call without any arguments
    if (arg_total == 1) {
        show_environment();
    }

    if ((verbose_flag == 1) && (remove_flag == 1)) {
        for (int i = 1; i < optind; i++) {
            if (!strcmp(arg_list[i], "-u")) {
                fprintf(stderr, "Removed %s!\n", arg_list[i+1]);
            }
        }
    }
// Traverse the arguments that aren't options0
    for (int i = optind; i < arg_total; i++) {
        if (strchr(arg_list[i], '=')) {
            char *pair_copy = arg_list[i];
            char *key = strtok(pair_copy, "=");
            char *val = strtok(NULL, "=");

            if ((key == NULL) || (val == NULL)) {
                fprintf(stderr, "Invalid name=value pair!\n");
                return EXIT_FAILURE;
            }

            if (setenv(key, val, 1)) {
                perror("setenv");
                return EXIT_FAILURE;
            }

            if (verbose_flag == 1) {
                fprintf(stderr, "Added %s=%s pair!\n", key, val);
            }
        }
        else {
            // Print the name of the program to be executed
            if (verbose_flag == 1) {
                fprintf(stderr, "Executing %s!\n", arg_list[i]);
            }
            // Run
            execvp(arg_list[i], &arg_list[i]);
            perror("execvp");
            return EXIT_FAILURE;
        }
    }

    if ((remove_flag == 0) && (verbose_flag == 0)) {
        show_environment();
    }
  return EXIT_SUCCESS;
}
```