Dekker's mutual exclusion procedure

With the help of Dekker's algorithm, it is possible to synchronize two actions or two processes that, for example, look like this:

```
process proc(and){                    /*and[0.1] */
    repeat
        enter the critical section
        WHO;
        exit the critical section
        NKO;
    }while satisfied(condition);
}
```

Dekker's algorithm:

```
common variables : RIGHT, FLAG[0..1]

function enter_the_critical_section(i,j) {

    FLAG[i] = 1
    while FLAG[j]<>0does {
        if it is RIGHT==jthen {
            FLAG[i] = 0
            while RIGHT==jdoes {
                nothing
            }
            FLAG[i] = 1
        }
    }
}

function    exit_critical_section(i,j)
{
    LAW    = j
    FLAG[i]          = 0
}
```

Additional instructions:

If the program deals with processes, then the shared variables should be organized in such a way that the space for them is occupied at once and shared among them. This is necessary due to the limited number of segments and the large number of users.

Depending on the load on the computer and the number of running processes, and in order to see the differences during program execution, program execution can be slowed down with:

sleep(1);

at the end of the critical section (KO).

Lamport's mutual exclusion procedure

With the help of Lamport's algorithm, it is possible to synchronize two or more processes, that is, two or more processes, which, for example, can look like this:

```
process proc(and){                    /*and[0..n] */
    repeat
        enter the critical section;
        WHO;
        exit the critical section;
        NKO;
    }while satisfied(condition);
}
```

Lamport's algorithm:

```
common variables : INPUT[0..n-1], NUMBER[0..n-1] function
enter_critical_section(s) {

    ENTRANCE[and] = 1;
    NUMBER[and] =max(NUMBER[0],...,NUMBER[n-1]) + 1;
    ENTRANCE[and] = 0;

    for j=0to n-1does
        while ENTRANCE[j] <> 0does
            nothing;
        while NUMBER[j] <> 0 && (NUMBER[j] < NUMBER[i] || (NUMBER[j] == NUMBER[i] && j < does
and))
            nothing;
}

function exit_critical_section(s) {

    NUMBER[and] = 0;
}
```

Additional instructions:

If the program deals with processes, then the shared variables should be organized in such a way that the space for them is occupied at once and shared among them. This is necessary due to the limited number of segments and the large number of users.

Depending on the load on the computer and the number of running processes, and in order to see the differences during program execution, program execution can be slowed down with:

sleep(1);

at the end of the critical section (KO).

Problems due to execution of instructions "over the line"

For the Dekker, Peterson and Lamport algorithm to work correctly, it is assumed that the program instructions are executed in the given order. However, some processors, in order to speed up program execution, allow instructions to be executed "over the line" (eng.*out-of-order*). Such behavior can cause errors in the mutual exclusion algorithm.

For example, when performing Lampot's algorithm, it may happen that the next set of instructions is not executed in the given order.

```
1:      ENTRY[s]   =  1;
2:      largest    =  max(NUMBER[0],...,NUMBER[n-1]);
3:      NUMBER[i] = largest + 1;
4:      INPUT[i] = 0;
```

A processor that allows execution of instructions "over the line" might find that the instructions in the third and fourth lines are mutually independent and might execute them in an arbitrary order, e.g. line 4 before line 3 (the reason for this could be that the elementENTRY[s]located in a hand container, aNUMBER[s]not). In that case, it is no longer about Lamport's algorithm.

In order to avoid arbitrary execution of instructions, it should be indicated to the language processor that the access to disputed variables is precisely determined by the program code.

One of the waysto do this is to declare the variable as a data typeatomic:

```
# include <stdatomic.h> atomic_int
INPUT[N], NUMBER[N];
```

Another wayis to use atomic operations. Atomic operations, in addition to making the specified operation atomic (indivisible), also ensure the correct order of operations (instructions before must be finished first, instructions after must not start before this operation is finished). An example using them to set and read variables in the Lamport algorithm is below.

```
//instead of: INPUT[i] = 1;
__atomic_store_n (&INPUT[i], 1, __ATOMIC_RELEASE);

//instead: INPUT[i] = 0;
__atomic_store_n (&INPUT[i], 0, __ATOMIC_RELEASE);

//instead: NUMBER[i] = largest + 1;
__atomic_store_n (&NUMBER[i], largest + 1, __ATOMIC_RELEASE);
```

```
//instead: while (INPUT[j] == 1) ;
do __atomic_load (&INPUT[j], &inputJ, __ATOMIC_ACQUIRE); while (inputJ
== 1);

//instead: while (NUMBER[j] != 0 && etc) ;
do __atomic_load (&NUMBER[j], &numberJ, __ATOMIC_ACQUIRE); while
(numberJ != 0 && etc);

//instead: NUMBER[i] = 0;
__atomic_store_n (&NUMBER[i], 0, __ATOMIC_RELEASE);
```

More information about these operations atAtomic GCC youAtomic GCC: Memory model
.

Operating systems
Second laboratory exercise

1. Motivation

The goal of this laboratory exercise is a practical study and familiarization with the core calls related to the creation and management of processes. As part of the exercise, it is necessary to create a simple shell `fsh` *(Fer SHell)*.

2. Task

The task is divided into three main parts, and the list of functionalities that need to be realized is given below:

1. Basic launch of the program -**2 points**
2. Built-in commands `iCD` `exit` and signal propagation `SIGINT` - **1 point**
3. Extended program launch using environment variables -**1 point**

A shell is needed**be realized in the programming language C or C++**(any standard) and in any**UNIX environment**(eg GNU/Linux, FreeBSD, etc.). Additionally, the use of any functions it provides is allowed standard C library (eg `glibc` , `)musl` **except functions** **`system, execvp, execlp and execvpe`** .

2.1. Accessing documentation of core calls and library functions

During this lab exercise, you will encounter various complex kernel calls whose behavior needs to be studied in detail in order for the shell to work properly. The main source of information when working with commands or functions provided by the operating system is the command. The `Man` command receives `Man` as an argument the name of the program or function whose documentation is to be accessed. It is important to note that the documentation of the entire system is divided into separate sections grouped by the type of content they describe. For example, the third section describes library functions, the second section describes core calls, and the first section describes programs or built-in shell commands. Calling the command in `Man` the form `Man <program_name>` it searches the sections in sequence and returns the first documentation that corresponds to the name of the program. Unfortunately, some functions and commands share names, and it's possible that the command will `Man` return you with incorrect documentation, such as shown in the following example.

Accessing the corresponding documentation of the sleep library function.

```
$ man sleep
SLEEP(1)        User Commands
SLEEP(1)

NAME
          sleep - delay for a
specified amount of time
SYNOPSIS
          sleep NUMBER[SUFFIX]... sleep
          OPTION
. . .
```

```
$ man 3 sleep
sleep(3)        Library Functions
Manual       sleep(3)

NAME
          sleep - sleep for a specified
number of seconds

LIBRARY
          Standard C library (libc,
- lc)

SYNOPSIS
          # include <unistd.h>
          unsigned int
sleep(unsigned int seconds); . . .
```

We solve this problem by ordering `Man` we explicitly assign a section which should be searched. Thus, the documentation of any core call is accessed with the command `man 2 <call_name>`. and documentation of standard C functions libraries are accessed with the command `man 3 <function_name>`.

2.2. Basic program launch

Pseudocode1 roughly describes the operation of the shell. Before waiting for a command to be typed, your shell should print "fsh>". Each command consists of a command name and multiple arguments separated by an arbitrary number of spaces. It is not necessary to "delete" characters when entering the command.

Example 1: Pseudocode of basic shell operation.

```
while(1){
        prints"fsh> ";

        hang oncommand;
        read the character stringandprocess;

        command = find_command(); if(
        commandit is notbuilt in){
              createnewprocess;
              in the child start the program withenteredarguments; wait for the
              endof a child;
        }
}
```

Example 2: An example of a shell working properly.

```
fsh> /usr/bin/pwd
/home/student
fsh> /usr/bin/echo"greeting" greeting

fsh> /usr/bin/ls -lh
# print directory contents fsh> /bin/
asdf
fsh: Unknown command: /bin/asdf
```

2.2.1. Recommended course of action

1. Write the basic shell skeleton,
2. Create a function that breaks the input character string into command name and arguments (if any),
3. Study the core calls in detail fork, wait, i exec,
4. Create a function that recognizes "built-in" commands (required for another subtask),
5. To start the program using the core calls specified in the third point.

2.3. Basic built-in commands and signal propagation

2.3.1. Command CD

Built-in command CD should provide the user with basic navigation by file system, for which we recommend using the function chdir() (at least 3 chdir). An example of a correctly executed command is given in the Example3 . Special attention should be paid to the following cases:

- For all errors (e.g. non-existent directories) it is necessary to print a corresponding message at stderr

Example 3: Example of printout when using a correctly executed command CD

```
fsh> /usr/bin/pwd
/home/student
fsh> cd Documents
fsh> /usr/bin/pwd
/home/student/Documents
fsh> cd /does not exist
CD: The directory'/non-existent'does not exist
```

2.3.2. Command exit

Built-in command exit should allow the user to exit the shell when it the user types or when the user sends an empty string as a command.

2.3.3. Sending a signal

Your shell must allow the user to manually stop execution using a key combination CTRL-C , or by sending a signal SIGINT . At the same time, it is important yes sent signal**it does not stop the shell, but the currently running program**. If the user sends SIGINT while no program is started, it is only necessary to print a new line. To realize this functionality, we recommend a family of functions sigaction (at least 3 sigaction). Your shell should "catch" the signal and decide what to do with it to make.

2.3.4. Important notes

Concept*process groups*1and standard*POSIX*stipulate that all signals sent via the keyboard must be propagated to all processes that are members of the so-called *foreground process*groups. As our shell is a member of that group, in our case it leads to incorrect signal propagation SIGINT because it is propagated to all processes started from the shell, not just the shell. The solution to this problem is to place the newly created child process in a separate group**by invitation functions setpgid(0,0) (man 3 setpgid) in the child before the call execute()** .

By pressing the key combination CTRL+C Your shell**does not get any string characters to standard input**, but the core of the operating system recognizes that combination and sends a signalSIGINT . Therefore, it is not necessary to check whether the shell is received signs "J" as an entrance.

2.3.5. Recommended course of action

1. Add i CD exit in the list of built-in commands,
2. Add functionality for the previously mentioned commands,
3. Change the launch of the child so that the child is placed in a separate process group,
4. Create a signal processing function SIGINT.

2.4. Extended program startup using environment variables

In the previous examples, it was necessary to specify their full path for all commands. As this whole process is clumsy and time-consuming, shells usually provide the option to enter only the name of the program. After the command name is entered, the shell looks for the command program in previously defined parts of the file system that are accessible via an environment variable PATH [2].

Variable PATH contains a string of paths separated by "**:**", and an example of content variables is given in the Example4 .

Example 4: Example of variable content PATH

```
> echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/sbin
```

Your task is to extend the basic functionality of the shell to allow programs to be launched by their name. Your shell needs to retrieve the contents of the variable PATH using a library call getenv (man 3 getenv), process it and search everything every time the program is started listed directories to find the appropriate program.

Example 5: Using calls getenv.

```
process_variable_path() {
        . . .
        char*content = getenv("PATH"); . . .

}
```

2.4.1. Important notes

- To find the program in the directory, we recommend using the call access (and core call and library function are allowed)
- The search process must not be started if the user types a path (ie if the command name starts with'**/**'or'**.**')
- An error message should be printed if the program does not exist

2.4.2. Recommended course of action

1. Study the call `getenv()`,
2. Realize retrieval and processing of environment variable content `PATH` at startup shells,
3. Add search program command before starting.