OPERATING SYSTEMS
– Seminar 1 –

THE ATTENDANCE REQUIREMENTS

- minimum 5 seminar attendances (minimum 4 attendances for repeating students)

1. UNIX COMMANDS

- the structure of UNIX commands:

```
command [options] [values]
```

  - `command`: the first word in the command line (lowercase letters and/or digits)
  - `options`:
    - short option - a single letter preceded by *a single hyphen* (`-`)
    - long option - a word preceded by *a double hyphen* (`--`)
  - `values`: may be mandatory, optional or it may not exist
  - the command line arguments are separated by *SPACES*
  - the command line interpreter (the shell) is *case-sensitive*
- examples:
  - the command only: `pwd, ls`
  - command + option: `ls -l, ls -a (ls --all)`
  - command + value: `mkdir new_dir, touch new_file`
  - command + option + value: `ls -l /etc, cat -n hello.c`

2. REGULAR EXPRESSIONS

- a regular expression (*regexp*) = *a finite character sequence defining a search pattern*
- a match = *a single character, a sequence of characters, a sequence of bytes, a piece of text*
- the special characters (meta-characters):

. period (dot)	\ backslash	^ caret	$ dollar sign
| vertical bar	? question mark	* asterix (star)	+ plus sign
(opening parenthesis	) closing parenthesis	[square bracket	{ curly brace

  - these special characters *have a different meaning* in regular expressions
  - you need to escape them (using \ backslash) in order to restore their own regular meaning
- the meaning of special characters in regular expressions:

Expression	Matches
`.`	any single char
`\.`	the . (dot) char
`[abc]`	a single char inside square brackets (`a`, `b` or `c`)

`[^abc]`	a single char EXCEPT those inside square brackets (`d`, `e`, ..., `z`)
`[a-z]`	a single lowercase letter from `a` to `z` (any lowercase letter)
`[A-Z]`	a single uppercase letter from `A` to `Z` (any uppercase letter)
`[a-zA-Z]`	a single lowercase or uppercase letter
`[0-9]`	a single digit from `0` to `9`
`[^0-9]`	a single char which IS NOT digit
`\d`	a single digit from `0` to `9` (equivalent with `[0-9]`)
`\s`	a single whitespace char (including `SPACE`, `TAB`, `CR`, `LF`)
`\w`	a single alfanumeric char or `_` (underscore)
`\(\)`	capture a group

- example:

  1. Given the following text lines:

     ```
     abc
     bdf
     ceg
     ```

  you can write some regular expressions to match:

  - only the first line: `'abc'`
  - only the second line: `'bdf'`
  - only the third line: `'ceg'`
  - all the lines above: `'...'` or, much better, `'[abc][bde][cfg]'`

- anchors:

Symbol	Matches
`^`	the start of line
`$`	the end of line
`\<`	the empty string at the beginning of a word
`\>`	the empty string at the end of a word
`\b \b`	equivalent with `\< \>`

- repetition operators:

Operator	Meaning
`?`	either zero or one time
`*`	zero or more times
`+`	one or more times
`{n}`	exactly `n` times
`{n,}`	`n` times or more
`{,m}`	at most `m` times
`{n,m}`	at least `n` times, but at most `m` times

- example:

  2. Given the following text lines:

     ```
     aaabc
     ```

```
aaadf
aaace
```

you can write a regular expression to match:

- all the lines above: `'aaa[bdc][cfe]'` or `'a{3}[bdc][cfe]'`

3. grep

- searches the input file and prints all the lines which contain the given pattern
- its name is derived from "*global regular expression print*"
- command syntax:

```
grep [OPTIONS] PATTERN [FILE...]
grep [OPTIONS] [-e PATTERN] [-f FILE...] [FILE...]
```

- OPTIONS:

`-c (--count)`	print a count of matching lines
`-i (--ignore-case)`	ignore case distinctions
`-v (--invert-match)`	invert the sense of matching
`-A NUM (--after-context=NUM)`	print `NUM` lines after matching lines
`-B NUM (--before-context=NUM)`	print `NUM` lines before matching lines
`-C NUM (-NUM --context=NUM)`	print `NUM` lines from all matching lines

- *PATTERN* is usually provided in the command line using a regular expression
- to specify multiple search patterns, or to protect a pattern beginning with a *hyphen* (`-`):

```
-e PATTERN (--regexp=PATTERN)
```

- to obtain patterns from *FILE* (one pattern per line):

```
-f FILE (--file=FILE)
```

4. sed (Stream EDitor)

- is a non-interactive text editor used to perform basic text transformations on an input stream
- reads and process all lines of the input stream one by one, and prints the result on the screen
- command syntax:

```
sed [-n] [-e] '[/pattern/]command' [input-file]
sed [-n] -f script-file [input-file]
```

-n	suppress automatic printing of internal buffer *(pattern space)*
-e *script*	add *script* to the commands to be executed
-f *script-file*	add the contents of *script-file* to the commands to be executed

– the input stream may be: the standard input stream (keyboard), a file denoted by `input-file` or the result of another command(s) execution

– if not specified a pattern, a certain line, or multiple lines, `command` will be executed on all the lines of input stream

- selecting lines (line addressing):

`N`	just line `N`

`$`	just last line
`M, N`	from line `M` to line `N`
`M~step`	from line `M`, lines from `step` to `step`
`/regexp/`	just the lines containing the pattern given by `regexp`
`0, /regexp/`	just the first line containing the pattern given by `regexp`
`M, +N`	from line `M`, `N` lines after
`M, ~N`	from line `M`, all the lines which are multiple of `N`

- commands:
  - p (print)

```
sed angajati.txt
sed 'p' angajati.txt
sed –n 'p' angajati.txt
sed –n '2p' angajati.txt
sed –n '/Tudor/p' angajati.txt
sed –n '2,5p' angajati.txt
sed –n '/Ion/,/Victor/p' angajati.txt
sed –e '2p' –e '5p' angajati.txt
```

  - d (delete)

```
sed 'd' angajati.txt
sed '4d' angajati.txt
sed '/Tudor/d' angajati.txt
sed '2,5d' angajati.txt
sed '/Tudor/,$d' angajati.txt
sed –e '2d' –e '5d' angajati.txt
```

  - s (substitute)

```
sed 's/Tudor/Tudorel/' angajati.txt
sed –n 's/Tudor/Tudorel/' angajati.txt
sed –n 's/19/18/g' angajati.txt
sed –n 's/1931/1932/p' angajati.txt
sed –n 's/\(Ion\)el/\1ut/p' angajati.txt
sed -n 's/\[0-9\]\[0-9\]$/&\.5/' angajati.txt
sed –n '/Olga/,/Toma/s/$/**CONCEDIU**/' angajati.txt
```

  - a (append)

```
sed '3a Linie adaugata' angajati.txt
sed '$a TERMINAT' angajati.txt
sed '/Adrian/a Linie adaugata' angajati.txt
```

  - c (change)

```
sed '2c SALARIAT PENSIONAT' angajati.txt
```

  - i (insert)

```
sed '1i \t\t\tDATE DESPRE PERSONAL' angajati.txt
```

- q (quit)
  `sed '5q' angajati.txt`
- r (read content from file)
  `sed '3r text.txt' angajati.txt`
- w (write content to file)
  `sed -n 'w angajati.bak' angajati.txt`
- = (print line number)
- l (display control characters)
  `sed -n 'l' test.txt`
- n (next)
- y (transform)
- h (holding)
- g (getting)
- x (exchange)

5. awk

- is not only a text processing utility, but also an interpreted programming language with a C-like syntax
- its name is derived from its creators: Alfred Aho, Peter Weinberger, Brian Kernighan
- command syntax:

```
awk [OPTIONS] '/pattern/' [input-file]
awk [OPTIONS] '{action}' [input-file]
awk [OPTIONS] '/pattern/{action}' [input-file]
```

-F *fs*	to change the default input field separator with *fs*
-f *script-file*	to obtain the commands from *script-file*

- awk reads and process all lines of the input file one by one
- each line represents an input record
- default input record separator: CR (Carriage Return)
- the current input record is stored in the internal variable `$0`
- each input record is parsed and separated into chunks called fields
- default input field separators: SPACE or TAB
- built-in variables:

`$0`	the current input record
`$1, $2, ...`	the fields of the current input record
`NR`	the total number of input records seen so far
`NF`	the number of fields in the current input record
`RS`	the input record separator
`ORS`	the output record separator

`FS`	the input field separator
`OFS`	the output field separator
`OFMT`	the format for converting numbers to strings for printing with `print`
`ARGC`	the number of command line arguments
`ARGV`	the array of command line arguments
`FILENAME`	the name of the current input file
`FNR`	the current record number in the current file
`ENVIRON`	the array of environment variables

- examples:
  - print all lines of the input file:
    ```
    awk '{print}' angajati.txt
    awk '{print $0}' angajati.txt
    ```
  - print all lines which contain the given pattern:
    ```
    awk '/Tudor/' angajati.txt
    awk '/Tudor/{print}' angajati.txt
    awk '/Tudor/{print $0}' angajati.txt
    ```
  - change the default input field separator:
    ```
    awk -F: '{print $1}' /etc/passwd
    awk -F: '{print NR, $1}' /etc/passwd
    awk -F'[ :\t]' '{print $1, $2, $3}' angajati.txt
    ```
- *relational operators:*

Operator	Name	Example
`<`	less than	`x < y`
`<=`	less than or equal to	`x <= y`
`==`	equal	`x == y`
`!=`	not equal	`x != y`
`>`	greather than	`x > y`
`>=`	greather than or equal to	`x >= y`
`~`	matches the regular expression	`x ~ /regexp/`
`!~`	does not match the regular expression	`x !~ /regexp/`

- examples:
  - using relational operators:
    ```
    awk '$5 < 2000' angajati.txt
    awk '$5 < 2000 {print}' angajati.txt
    awk '$5 == 1942 {print NR, $1}' angajati.txt
    ```
  - using relational operators and regular expressions:
    ```
    awk '$1 ~ /Tudor/ {print}' angajati.txt
    awk '$1 !~ /Tudor/ {print}' angajati.txt
    ```
- *logical operators:* `&&` `||` `!`

- *arithmetic operators:* `+ - * / % ^`
- *assignment operators:* `= += -= *= /= %= ^=`
- *conditional expressions:*

  `condition ? expresion1 : expresion2`

  is equivalent with:

```
if (condition)
  expresion1
else
  expresion2
```

- scripts:
  - BEGIN: commands are executed once only, BEFORE the first input record is read
  - END: commands are executed once only, AFTER all the input is read
  - {} between BEGIN și END: commands are executed for each input record
  - examples:

```
awk 'BEGIN{FS = ":"}' /etc/passwd
awk 'BEGIN{FS = ":"; OFS="\t"} {print $1, S2}' /etc/passwd
awk '/Ion/{cnt++}END{print "Ion apare de " cnt " ori."}' angajati.txt
awk 'END{print "Nr. angajati: " NR}' angajati.txt
awk 'BEGIN{total=0} {total++} END{print "Total: " total}' angajati.txt
```

- instructions:

  `http://www.grymoire.com/Unix/AwkRef.html`
- built-in functions:

  `http://www.grymoire.com/Unix/AwkRef.html`

REFERENCES:

- Course page: `http://www.cs.ubbcluj.ro/~rares/course/os/`
- Regular expressions: `https://www.regular-expressions.info/quickstart.html`
- awk manual: `https://linux.die.net/man/1/awk`
- awk tutorial: `http://www.grymoire.com/Unix/Awk.html`
- grep manual: `https://linux.die.net/man/1/grep`
- sed manual: `https://linux.die.net/man/1/sed`
- sed tutorial: `http://www.grymoire.com/Unix/Sed.html#uh-41`

OPERATING SYSTEMS
– Seminar 2 –

FILES AND DIRECTORIES PERMISSIONS. SHELL PROGRAMMING

1. FILES AND DIRECTORIES PERMISSIONS

- *access rights (permissions)* are specified for each file or directory
- show files/directories permissions: `ls -l`

```
- rwx  rwx  rwx
  u    g    o
```

- symbolic representation of roles:
  - **u** (owner) - the owner of the file/directory
  - **g** (group) - the group to which the file/directory owner belongs
  - **o** (others) - other users (that are not part of the group to which the owner belongs)
- symbolic representation of files permissions:
  - **r** (read) - grants the capability to examine (view) the content of the file
  - **w** (write) - grants the capability to modify or remove the content of the file
  - **x** (execute) - allow users to run the file as a program
- symbolic representation of directories permissions:
  - **r** (read) - grants the capability to look at the files/subdirectories inside the directory
  - **w** (write) - grants the capability to add/delete files/subdirectories from the directory
  - **x** (execute) - grants the capability to enter (traverse) the directory
- numeric representation of access rights:

  r (read) = 4 **w** (write) = 2 **x** (execute) = 1
- changing access rights:

```
chmod 755 file_name
chmod 644 file_name
chmod +x file_name
chmod g+r file_name
chmod u+rw, g+r-w, g+r file_name
chmod u=rwx, g=rw, o=r file_name
```

2. SHELL PROGRAMMING

- a shell = a program that provides an interface between a user and an operating system (OS) kernel
- shells: *sh (Bourne shell)*, *csh (C shell)*, *ksh (Korn shell)*, *bash (GNU Bourne-again shell)*
- a script = a text file which contains commands (built-in and/or extern commands)
- a simple Bash script example:

```
#!/bin/bash

pwd
ls
```

Remarks:

– the sequence on the first line `#!` IS NOT a comment (it is called *shebang*)

– this sequence is followed by the absolute path to the command-line interpreter (to the Bash in this case)

- running a Bash script:
  `chmod +x script_1.sh`
  `./script_1.sh`
- comments: start with the `#` (hash)
- variabiles:
  - the name of a variabile may contain letters, digits and „ _” (underscore)
  - the first character must be a letter
  - the keywords cannot be used as variable names
  - the shell is case-sensitive
  - examples:
    ```
    n=45
    name=Ana
    msg="Enter a number:"
    ```
- keywords:
  `if then else elif fi`
  `for while until do done`
  `case in esac`
- built-in commands:
  - list the Bash built-in commands: `help`
  - display information about a specific command: `help` ***command***
  - examples: `echo` `read` `printf` `test`
- Bash script examples:
  - read and display a number: `script_2.sh`
  - read and display a string: `script_3.sh`
- built-in variables:

`$0`	the current file name
`$1,..., $9`	the arguments provided in the command-line
`$#`	the number of arguments provided in the command-line
`$*`	the list of arguments
`$@`	the list of arguments
`$?`	the exit status of the last command executed

`$$`	the PID of the current process
`$!`	the PID of the last command which was started in the background

- example: `script_4.sh`

2.1. ARITHMETIC EXPRESSIONS USING INTEGERS

- the shell variables are treated, by default, as strings (sequences of characters)
- example: `script_5.sh`

2.1.a. `expr` extern command

`expr` *`expression`*

- evaluates and displays the value of an arithmetic expression to the standard output
- operators:

`+ - \* / %`	Sum, difference, product, quotient or modulo
`= !=`	Numerical comparisons:
`\> \>=`	– returns 1 if the condition is true
`\< \<=`	– returns 0 if not
`\( \)`	defines a subexpression using round brackets
`S | D`	returns `S` if `S` is not NULL or 0, `D` otherwise
`S \& D`	returns `S` if `S` and `D` are not NULL or 0, 0 otherwise
`length S`	returns the length of the string `S`
`index S CHARS`	returns the position of `S` in `CHARS` or 0 (the first position is 1)
`substr S P L`	returns a substring of `S` which start at `P` and has the length `L`

2.1.b. `let` built-in command

- evaluates and displays the value of an arithmetic expression to the standard output
- operators: `++ -- ! ~ ** * / % + - << >> <= >= < > == != & ^ | && ||`

2.1.c. Compound command

- example: `script_6.sh`

2.2. `test` extern command

- sintax:

`test` *`expression`* or **`[`***`expression`*** `]`**

- evaluates *`expression`* and returns 0 if *`expression`* is true or a non-zero value if not
- is used for comparison of integers and character strings and to perform various file tests

a. Comparison of integers

- operators: `-lt -le -eq -ne -ge -gt`

b. Comparison of character strings

`-z s`	Check if the string s is 0 in length
`-n s1`	Check if the string s is different in length from 0
`s1 = s2`	Check if the two strings are equal
`s1 != s2`	Check if the two strings are not equal

c. File tests

`-e file`	Check if file exists
`-s file`	Check if file exists and is not empty
`-r file`	Check if file exists and can be read
`-w file`	Check if file exists and can be written
`-x file`	Check if file exists and can be executed
`-f file`	Check if file exists and is a regular file
`-d file`	Check if file exists and is a directory
`-L file`	Check if file exists and is a symbolic link
`-p file`	Check if file exists and is a pipe
`-c file`	Check if file exists and is a character special file
`-b file`	Check if file exists and is a block special file

2.3. IF/THEN/ELIF/ELSE/FI

- sintax:

```
if condition
then
  statement(s) to be executed
elif condition
then
  statement(s) to be executed
elif condition; then
  statement(s) to be executed
else
  statement(s) to be executed
fi
```

- examples: `if_1.sh, if_2.sh`

2.4. FOR/DO/DONE

- sintax:

```
for var in list
do
  statement(s) to be executed
done
```

- examples: `for_1.sh, for_2.sh, for_3.sh, for_4.sh, for_5.sh`
- *filename wildcards*:

*	match any number of characters
?	match only one character
`[abc]`	match any character in the square brackets
`[!abc]`	match any character which is not in the square brackets

- example:
  - list the files whose names begin with a letter followed by a dot and exactly 2 chars:
    `ls [a-zA-Z]*.??`

2.5. WHILE/DO/DONE, UNTIL/DO/DONE

- sintax:

```
while condition                    until condition
do                                 do
  statement(s) to be executed        statement(s) to be executed
done                               done
```

- examples: `while_1.sh`, `while_2.sh`, `while_3.sh`, `while_4.sh`, `while_5.sh`

2.6. CASE/ESAC

- sintax:

```
case var in
  pattern_1)
    statement(s) to be executed if patern_1 is matched;;
  pattern_2)
    statement(s) to be executed if patern_2 is matched;;
  ...
  *)
    default condition to be executed;;
esac
```

- example: `case_1.sh`

2.7. USEFUL COMMANDS

a. `cut` command

```
cut -d: -f 1 /etc/passwd
cut -d ":" -f 1 /etc/passwd
who | cut -d " " -f 1
```

b. `find` command

```
find . -type f -name "*.sh"
find /tmp -type d -empty
```

c. `shift` built-in command

`shift [n]` shifts to the left by `n` positions of the arguments provided in the command-line

d. `sleep` command

`sleep [n]` suspends the execution of current process for `n` seconds

e. `exit` built-in command

`exit [n]` finishes the execution and return to the process from which it was launched

REFERENCES:

- Course: `http://www.cs.ubbcluj.ro/~rares/course/os/`
- Seminar: `http://www.cs.ubbcluj.ro/~dbota/SO/sem_02`
- Shell programming: `https://ryanstutorials.net/bash-scripting-tutorial/`

OPERATING SYSTEMS
– Seminar 3 –

UNIX PROCESSES

1. UNIX PROCESSES

- *a proces* refers to a *program in execution (a running instance of a program)*
- anytime you run a program, you have created a process
- each process is assigned an unique process identification number (PID)
- the OS kernel permanently maintains the process list (process table)
- examining the process list (getting a snapshot of active processes):

```
ps -ef
ps -f -p 1
ps -f -u dbota
```

2. `fork()`

- function prototype:

```
#include <unistd.h>

pid_t fork(void);
```

- the **`fork()`** function shall create a new process
- the new process (child process) shall be an almost exact copy of the calling process (parent process)
- both processes shall continue to execute from the **`fork()`**function
- upon succesfull completion, **`fork()`** shall return:
  - `0` to the child process
  - process ID of the child process (child PID) to the parent process
- if **`fork()`** has failed:
  - -1 shall be returned to the parent process
  - no child process shall be created
  - **`errno`** shall be set to indicate the error
- the **`fork()`** function shall fail if:
  - insufficient storage space is available
  - the system lacked the necessary resources to create another process, or the system-imposed limit on the total number of processes under execution system-wide or by a single user would be exceeded
- examples: `fork_1.c, fork_2.c, fork_3.c`

3. wait() and waitpid()

- prototypes:

```
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- wait for a child process to stop or terminate
- the call **wait(&status)** is equivalent to **waitpid(-1, &status, 0)**
- the **wait()** function shall suspend execution of the calling thread until status information for one of the terminated child processes of the calling proces is available
- the **waitpid()** function shall suspend execution of the calling thread until one of the following events occurs:
  - the child process specified by **pid** has completed its execution
  - the child process specified by **pid** was stopped by a signal
  - the child process specified by **pid** was restarted by a signal
- choosing the right value for pid:

pid	Meaning
< -1	wait for any child process whose process group ID (GID) is equal to the absolute value of pid
-1	wait for any child procces
0	wait for any child process whose process group ID (GID) is equal to that of the calling process
> 0	wait for child process whose process ID (PID) is equal with pid

- examples: fork_4.c, fork_5.c

4. signal()

- function prototype:

```
#include <signal.h>

sighandler_t signal(int signum, sighandler_t handler);
```

- sets the disposition of signal *signum* to *handler*
- the *handler* is either **SIG_IGN**, **SIG_DFL** or the address of a programmer-defined function (a *„signal handler”*)
- if the signal *signum* is delivered to the process, then one of the following happens:
  - if the disposition is set to **SIG_IGN**, then the signal is ignored
  - if the disposition is set to **SIG_DFL**, then the default action associated with the signal occurs

– if the disposition is set to a function, then first either the disposition is reset to **SIG_DFL**, or the signal is blocked, and then *handler* is called with argument *signum*. If invocation of the handler caused the signal to be blocked, then the signal is unblocked upon return from handler.

- the signals SIGKILL and SIGSTOP cannot be caught or ignored
- to prevent the occurence of *„zombie"* processes, the parent process should call:
  `signal(SIGCHLD, SIG_IGN)`
- example: `fork_7.c`

5. `kill()`

- prototype:

```
#include <signal.h>

int kill(pid_t pid, int sig);
```

- send a signal to a process or a group of processes
- choosing the right value for `pid`:

pid	Meaning
`< -1`	The signal shall be sent to all processes whose process group ID (GID) is equal to the absolute value of `pid`, and for which the process has permission to send a signal
`-1`	The signal shall be sent to all processes for which the process has permission to send that signal
`0`	The signal shall be sent to all processes whose process group ID (GID) is equal to the process group ID of the sender, and for which the process has permission to send a signal
`> 0`	The signal shall be sent to the process whose process ID (PID) is equal with `pid`

- the list of signals:
  `kill -l`

6. `exec()` family of functions

- prototypes:

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char *const envp[]);

int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *file, char *const argv[], char *const envp[]);
```

- the current process image will be replaced with the new process image
- examples: `exec_1.c, exec_2.c, exec_3.c`

REFERENCES:

- Course: http://www.cs.ubbcluj.ro/~rares/course/os/
- Seminar: http://www.cs.ubbcluj.ro/~dbota/SO/sem_03

3. Procese Unix (în C): fork, exec, exit, wait system, signals

Contents

3.1. Standardul POSIX de gestiune a erorilor în apeluri sistem: errno

Marea majoritate a funcţiilor C şi practic toate apelurile sistem Unix întorc un rezultat care "spune" dacă funcţia/apelul s-a derulat normal sau dacă a apărut o situaţie deosebită. In caz de eşec funcţia / apelul sistem întoarce fie un întreg nenul (valoarea 0 este rezervată pentru succes), fie un pointer NULL etc. Metodologic, **se recomandă SA SE APELEZE:**

```
if (functie( - - - ) == SUCCES) { tratare normala }
else      { tratare situatie de derulare anormala }
```

NU (aşa cum din comoditate apelează mulţi programatori) :

```
functie( - - -); tratare normala
```

Evident forma a doua este mai scurtă, dar nu sunt tratate situaţiile de excepţie.

Pentru o abordare unitară a acestor tratamente, standardul POSIX oferă prin **`#include <errno.h>`** o variabilă întreagă **`errno`**, (care nu trebuie declarată) a carei valoare este setata de sistem *nunai în caz de derulare anormală a apelului*! **La o situaţie de derulare anormală sistemul fixează o valoare nenulă ce indică cauza erorii**. Pentru detalii vezi `man errno`, precum şi lista completă a cazurilor de erori, aflată de exemplu la **`http://www.virtsync.com/c-error-codes-include-errno`**
La apelul cu succes al unei funcţii sistem *errno nu se seteaza la 0!* Pentru a se vedea şi în clar eroarea depistată se pot folosi funcţiile `strerror` şi `perror` dau detalii pentru fiecare valoare a lui `errno`:

```
#include <errno.h>
 - - -
if (functie( - - - ) == SUCCES) { tratare normala }
else      { perror("Eroarea depistata este:");
            // sau printf("Eroarea depistata este:%s", strerror(errno); }
```

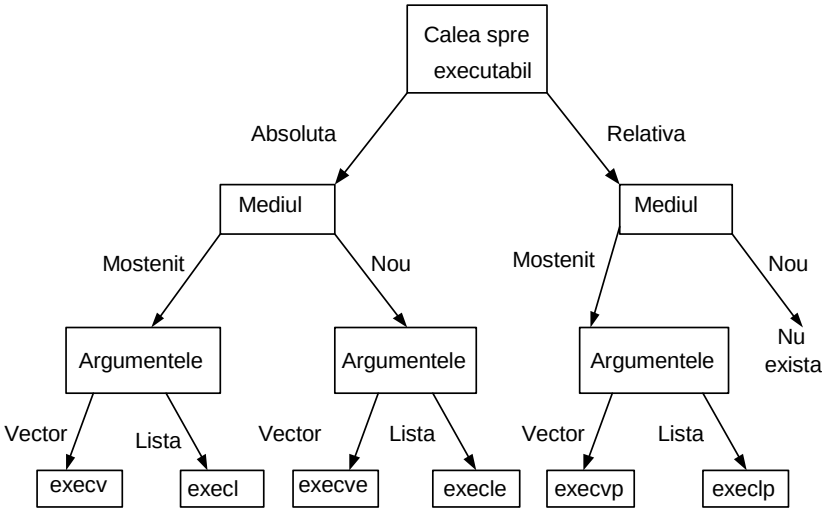
3.2. Principalele apeluri sistem Unix care operează cu procese

Tabelul următor prezintă sintaxele principalelor apeluri sistem Unix care operează cu procese:

Funcţii specifice proceselor

`fork()`
`exit(n)`
`wait(p)`
`exec*(c, lc)`
`system(c)`

Tipurile de exec:



Prototipurile lor sunt descrise, de regula, in <`unistd.h`> Parametrii sunt:

- `n` este intreg – codul de retur cu care se termină procesul;
- `p` este un pointer la un întreg unde fiul întoarce codul de retur (extras cu funcţia `WEXITSTATUS);`
- `c` este o comandă Unix;
- `lc` este linia de comandă (comanda c urmată de argumentele liniei de comandă);
- `f` este un tablou de doi intregi – descriptori de citire / scriere din / in pipe;
- `nume` este numele (de pe disc) al fişierului FIFO, iar `drepturi` sunt drepturile de acces la acesta;
- `fo` si `fn` descriptori de fisiere: `fo` deschis in program cu `open`, `fn` poziţia în care e duplicat `fo`.

In caz de eşec, functiile întorc -1 (NULL la `popen`) si poziţionează `errno` se depisteaza ce eroare a aparut.

Funcţiile `system` si `popen` au comanda completă (un string), interpretabilă de shell: aceste funcţii lansează mai întâi un shell, apoi în acesta lansează comanda `c`. Această lansare se face simplu folosind un apel sistem `execl`: **`execl("/bin/sh", "sh", "-c", c, NULL);`**
Comanda `c` din apelurile `exec*` ***NU permite specificări folosite uzual în liniile shell***. Astfel, în `c` NU trebuie să apară specificări generice de fişiere, redirectări de intrări / ieşiri standard, variabile shell, captări de ieşiri prin construcţii ` - - comanda - - ` etc. Dacă totuşi se doreşte acest lucru, trebuie să se lanseze, ca mai sus, interpretorul sh cu opţiunea -c şi apoi să se specifice comanda.

3.3. Exemple de lucrul cu procese

3.3.1. Utilizări simple fork exit, wait

Vom prezenta şi discuta două exemple de programe care utilizează apelurile sistem `fork, exit, wait`. Să considerăm **programul `f1.c`** căruia i-am numerotat liniile sursă:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <sys/wait.h>
5   int main() {
6       int p, i;
7       p=fork();
8       if (p == -1) {perror("fork imposibil!"); exit(1);}
9       if (p == 0) {
10          for (i = 0; i < 10; i++)
11              printf("Fiu: i=%d pid=%d, ppid=%d\n", i, getpid(), getppid());
12          exit(0);
13      } else {
14          for (i = 0; i < 10; i++)
15              printf("Parinte: i=%d pid=%d ppid=%d\n", i, getpid(), getppid());
16          wait(0);
17      }
18      printf("Terminat; pid=%d ppid=%d\n", getpid(), getppid());
19  }
```

Vom analiza comportamentul acestui program în diverse situaţii, făcând o serie de modificări în această sursă.

Rularea în forma iniţială: Sunt afişate 21 linii: 10 ale fiului de la linia 11 cu pidul lui şi al părintelui. 11 ale fiului, 10 de la linia 15 şi ultima de la linia 18. Părintele părintelui este pidul shell. Este posibil ca ordinea primelor 20 de linii să apară amesteecate, linii ale fiului şi liniile ale părintelui. Dacă la linia 10 şi la linia 14 se înlocuieşte 10 cu 1000, se vor afişa 2001 linii iar amestecarea între liniile fiului şi ale părintelui va fi mai evidentă.

Comentarea liniei 12: Procesul fiu se termină la linia 18, ca şi părintele. Se vor tipări 22 linii, linia 18 se va tipări de două ori: odată de părinte şi odată de fiu.
Comentarea liniei 16: Părintele nu mai aşteaptă terminarea fiului şi acesta din urmă rămâne în starea zombie. Se tipăresc cele 21 de linii ca în primul caz. O observaţie intereesantă: dacă ieşirea programului se redirectează într-un fişier pe disc, apar cele 21 linii. In schimb, dacă ieşirea se face direct pe terminal, apar doar liniile fiului. De ce oare? Rămâne un TO DO pentru studenţi.

Comentarea liniilor 12 şi 16: Se tipăresc 22 linii, cu aceeaşi observaţie de mai sus, de la comentarea liniei 16. Aici recomandăm modificări ale numărului liniilor tipărite de fiu (linia 10) şi a celor tipărite de părinte (linia 14). Se vor vedea efecte interesante.

Să considerăm **programul `f2.c`:**

```
main() { fork(); if (fork()) {fork();} printf("Salut\n");}
```

Care este efectul execuţiei acestui program? (Acoladele nu sunt necesare, dar le-am pus pentru a evidenţia mai bine corpul lui `if`). Să facem o primă analiză:

- Primul `fork` naşte un proces fiu. Ambele procese au de executat secvenţa: `if (fork()) fork(); printf("Salut\n");`
- Condiţia `fork` din `if` mai naşte câte un proces fiu cărora le rămâne de făcut doar `printf("Salut\n");` In acelaşi timp, cele două procese care evaluează if mai au de făcut `{fork();} printf("Salut\n");` Până aici avem patru procese.
- Fiecare fork dintre acolade mai naşte câte un proces fiu căruia îi mai rămâne de făcut `printf("Salut\n");` Avem încă două procese în plus.
- In concluzie, avem şase (6) procese care au de executat `printf("Salut\n");` In consecinţă, se va tipări de 6 ori `Salut`.

Merită să studiem mai atent acest exemplu. Principala carenţă a lui este aceea că nici un părinte care naşte un fiu nu aşteaptă terminarea lui prin `wait`. Consecinţa, vor rămâne câteva procese în starea zombie.

Pentru a aprofunda analiza, să rescriem puţin programul `f2.c`:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
    printf("START: pid=%d ppid=%d\n", getpid(), getppid());
    int i=-2, j=-2, k=-2;
    i=fork();
    if (j=fork())
        {k=fork();}
    printf("Salut pid=%d ppid=%d i=%d j=%d k=%d\n",getpid(),getppid(),i,j,k);
}
```

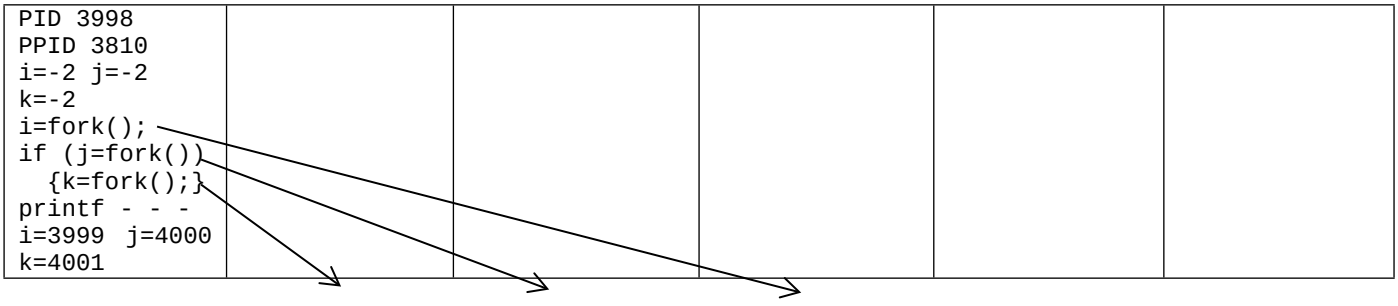
In fapt, am reţinut în variabilele `i, j, k` valorile PID-urilor create pe parcursul execuţiei. Rezultatul execuţiei este:

```
START: pid=3998 ppid=3810
Salut pid=3998 ppid=3810 i=3999 j=4000 k=4001
florin@ubuntu:~/c$ Salut pid=4001 ppid=1700 i=3999 j=4000 k=0
Salut pid=4000 ppid=1700 i=3999 j=0 k=-2
Salut pid=3999 ppid=1700 i=0 j=4002 k=4003
Salut pid=4003 ppid=1700 i=0 j=4002 k=0
Salut pid=4002 ppid=1700 i=0 j=0 k=-2
```

Să analizăm ordinea în care se execută aceste instrucţiuni:

- 3810 este PID-ul shell care afişează prompterul, iar 3998 este PID-ul programului iniţial.
- Procesul 3998 crează fiul i cu PID-ul 3999, fiul j cu PID-ul 4000 şi fiul k cu PID-ul 4001. Apoi îşi face tipărirea şi se termină - se vede tipărirea prompterului.
- Cele trei procese 3999, 4000 şi 4001 rămân active dar sunt în starea zombie (PPID-ul lor este 1700).
- Procesul 4001 preia controlul procesorului, valorile i şi j sunt moştenite de la 3998, iar k = 0 fiind vorba de fork în fiu, face tipărirea şi se termină.
- Procesul 4000 preia controlul procesorului, valorile i şi k sunt moştenite de la 3998 - i creat, k încă necreat, iar j = 0 fiind vorba de fork în fiu, face tipărirea şi se termină.
- Procesul 3999 preia controlul procesorului, i = 0 fiind vorba de fork în fiu, crează fiul j cu PID-ul 4002 şi fiul k cu PID-ul 4003. Apoi îşi face tipărirea şi se termină.
- Procesul 4003 preia controlul procesorului, valorile i şi j sunt moştenite de la 3999, iar k = 0 fiind vorba de fork în fiu. Apoi îşi face tipărirea şi se termină.
- Procesul 4002 preia controlul procesorului, valorile i şi k sunt moştenite de la 3999, iar j = 0 fiind vorba de fork în fiu. Apoi îşi face tipărirea şi se termină.

Tabelul următor prezintă cele 6 procese: ce valori moştenesc de la părinte, ce cod mai au de executat şi ce valori finale au (ce tipăresc).



	PID 4001 PPID 3998 i=3999 j=4000 k=0 printf - - - i=3999 j=4000 k=0	PID 4000 PPID 3998 i=3999 j=0 k=-2 printf - - - i=3999 j=0 k=-2	PID 3999 PPID 3998 i=0 j=-2 k=-2 if (j=fork()) {k=fork();} printf - - - i=0 j=4002 k=4003		
				PID 4003 PPID 3999 i=0 j=4002 k=0 printf - - - i=0 j=4002 k=0	PID 4002 PPID 3999 i=0 j=0 k=-2 printf - - - i=0 j=0 k=-2

In acest tabel, valoarea PPID este cea reală a părintelui creator, deşi la momentul terminării fiului părintele nu mai există, aşa că procesul intră în starea zombie.

Pentru a evita starea acumularea de procese în starea zombie, se poate folosi, spre exemplu, secvenţa:

```
 - - -
# include <signal.h>
int main() {
signal(SIGCHLD, SIG_IGN);
 - - -
```

In acest fel se cere ignorarea trimiterii de către fiu a semnalului SIGCHLD, pe care părintele ar trebui să îl primească (să fie în viaţă), să îl trateze cu un wait. Prin această ignorare, procesul fiu fiu este sters din sistem imediat după terminarea lui.

3.3.2. Utilizări simple execl, execlp. execv, system

Urmatoarele două programe, desi diferite, au acelasi efect. Toate trei folosesc o comanda de tip `exec`, spre a lansa din ea comanda shell:

```
ls -l
```

Programul 1:

```
#include <stdio.h>
#include <unistd.h>
int main() {
    char* argv[3];
    argv[0] = "/bin/ls";
    argv[1] = "-l";
    argv[2] = NULL;
    execv("/bin/ls", argv);
}
```

Aici se pregateste linia de comandă în vectorul `argv` spre a o lansa cu `execv`.

Programul 2:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h> // trebuie pentru system
int main() {
    //execl("/bin/ls", "/bin/ls", "-l", NULL);
    // execlp("ls", "ls", "-l", NULL);
    // execl("/bin/ls","/bin/ls","-l","p1.c","execl.c", "fork1.c", "xx", NULL);
    // execl("/bin/ls","/bin/ls","-l","*.c", NULL);
    system("ls -l *.c");
```

```
}
```

Aici se executa, pe rand, numai una dintre cele 5 linii, comentând pe celelalte 4. Ce se va intampla?

- Primul `execl` lanseaza `ls` prin cale absolută şi are acelaşi efect ca şi programul 1.
- Al doilea lansează `ls` prin directoarele din PATH, efectul este acelaşi.
- Al treilea cere `ls` pentru o listă de fişiere. Pentru cele care nu există, se dă mesajul: `/bin/ls: cannot access 'xx': No such file or directory` (in loc de xx apar numele fişierelor inexistente);
- Al patrulea exec va da mesajul: `/bin/ls: cannot access *.c: No such file or directory` **Nu este interpretat asa cum ne-am astepta!** De ce? Din cauza faptului ca specificarea *.c reprezinta o specificare generica de fisier, dar numai shell "stie" acest lucru si el (shell) inlocuieste aceasta specificare, in cadrul uneia dintre etapele de tratare a liniei de comanda. La fel stau lucrurile cu evaluarea variabilelor de mediu, ${---}, inlocuirea dintre apostroafele inverse ` --- `, redirectarea I/O standard etc.
- Apelul `system` are efectul aşteptat, făcând rezumatul tuturor fişierelor de tip c din directorul curent.

Să ne oprim puţin asupra funcţiei system. Ea crează prin fork un proces fiu, în care lansează comanda `execl("/bin/sh", "sh", "-c", c, NULL)` aşa cum am arătat mai sus şi întoarce codul de retur cu care s-a terminat execuţia comenzii. Doritorii pot să vadă sursa `system.c`, care este o funcţie simplă, de maximum 100 linii în care se includ comentariile, tratările cu `errno` ale posibilelor erori şi manevrarea unor semnale specifice. Sursa poate fi găsită la: `http://man7.org/tlpi/code/online/dist/procexec/system.c`

3.3.3. Un program care compileaza şi rulează alt program

Exemplul care urmeaza are acelasi efect ca si scriptul sh:

```
#!/bin/sh
if gcc -o ceva $1
then ./ceva $*
else echo "Erori de compilare"
fi
```

Noi nu il vom implementa în `sh`, ci vom folosi programul `compilerun.c`.

```
// Similar cu scriptul shell:
// #!/bin/sh
// if gcc -o ceva $1; then ./ceva $*
//     else echo "Erori de compilare"
// fi
#include<stdio.h>
#include<unistd.h>
#include <stdlib.h>
#include<string.h>
#include <sys/wait.h>
int main(int argc, char* argv[]) {
    char comp[200];
    char* run[100];
    int i;
    strcpy(comp, "gcc -o ceva ");
    strcat(comp, argv[1]); // Fabricat comanda de compilare
    if (WEXITSTATUS(system(comp)) == 0) {
        run[0] = "./ceva";
        for (i = 1; argv[i]; i++) run[i] = argv[i];
        run[i] = NULL; // Fabricat comanda pentru execv
        execv("./ceva", run);
    }
    printf("Erori de compilare\n");
}
```

Compilarea lui se face

```
gcc -o compilerun compilerun.c
```

Executia se face, de exemplu, prin

```
./compilerun argvenvp.c a b c
```

Cefect, daca compilarea sursei argument (`argvenvp.c`) este corecta, atunci compilatorul gcc creeaza fisierul `ceva` si intoarce cod de retur 0, dupa `ceva` este lansat prin `execv`. Daca esueaza compilarea, se va tipari doar mesajul `"Erori de compilare"`.

Am ales ca şi exemplu de program `argvenvp.c`:

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[], char *envp[]) {
    int i;
    printf("Argumentele:\n");
    for (i = 1; argv[i]; i++) printf("%s\n", argv[i]);
    printf("Cateva variabile de mediu:\n");
    for (i = 0; envp[i]; i++)
        if (strncmp("HOME", envp[i], 4)==0 || strncmp("LOGNAME", envp[i], 7)==0)
            printf("%s\n", envp[i]);
}
```

Secvenţa de execuţie este:

```
florin@ubuntu:~/c$ gcc -o compilerun compilerun.c
florin@ubuntu:~/c$ ./compilerun argvenvp.c a b c
Argumentele:
argvenvp.c
a
b
c
Cateva variabile de mediu:
HOME=/home/florin
LOGNAME=florin
florin@ubuntu:~/c$
```

3.3.4. Capitalizarea cuvintelor dintr-o listă de fişiere text

Se cere un program care primeşte la linia de comandă o listă de fişiere text. Se cere ca toate aceste fişiere să fie transformate în altele, cu acelaşi conţinut, dar în care fiecare cuvânt să înceapă cu literă mare. Se vor lansa procese paralele pentru prelucrarea simultană a tuturor fişierelor.

Pentru aceasta, vom crea mai întâi un program cu numele `cap` din sursa `cap.c`. Acesta primeşte la linia de comandă numele a două fişiere text, primul de intrare, al doilea de ieşire cu cuvintele capitalizate:

```
#include <stdio.h>
#include <string.h>
#define MAXLINIE 100
main(int argc, char* argv[]) {
    printf("Fiu: %d ...> %s %s\n", getpid(), argv[1], argv[2]);
    FILE *fi, *fo;
    char linie[MAXLINIE], *p;
    fi = fopen(argv[1], "r");
    fo = fopen(argv[2], "w");
```

```
    for ( ; ; ) {
        p = fgets(linie, MAXLINIE, fi);
        linie[MAXLINIE-1] = '\0';
        if (p == NULL) break;
        if (strlen(linie) == 0) continue;
        linie[0] = toupper(linie[0]); // Pentru cuvantul care incepe in coloana 0
        for (p = linie; ; ) {
            p = strstr(p, " ");
            if (p == NULL) break;
            p++;
            if (*p == '\n') break;
            *p = toupper(*p); // Caracterul de dupa spatiu este facut litera mare
        }
        fprintf(fo, "%s", linie);
    }
    fclose(fo);
    fclose(fi);
}
```

Al doilea program, numit `master.c` va crea câte un proces pentru fiecare nume de fişier primit la linia de comandă şi în acel proces va lansa `cap fi fi.CAPIT`

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
main(int argc, char* argv[]) {
    int i, pid;
    char argvFiu[200];
    for (i=1; argv[i]; i++) {
        pid = fork();
        if (pid == 0) {
            strcpy(argvFiu, argv[i]);
            strcat(argvFiu, ".CAPIT");
            execl("./cap", "./cap", argv[i], argvFiu, NULL);
        } else
            printf("Parinte, lansat fiul: %d ...> %s %s \n", pid, argv[i], argvFiu);
    }
    for (i=1; argv[i]; i++) wait(NULL);
    printf("Lansat simultan %d procese de capitalizare\n", argc - 1);
}
```

Compilari:

```
>gcc -o cap cap.c
>gcc -o master master.c
```

Lansare `master f1 f2 ... fi ... fn`

3.3.5. Câte perechi de argumente au suma un număr par?

La linia de comandă se dau n perechi de argumente despre care se presupune ca sunt numere întregi si positive. Se cere numărul de perechi care au suma un număr par, numărul de perechi ce au suma număr impar si numărul de perechi în care cel putin unul dintre argumente nu este număr strict pozitiv.

Rezolvarea: In procesul părinte se va crea câte un process fiu pentru fiecare pereche. Oricare dintre fii întoarce codul de retur:

- 0 daca perechea are suma pară,
- 1 daca suma este impară,
- 2 daca unul dintre argumente este nul sau nenumeric.

Parintele aşteaptă terminarea fiilor şi din codurile de retur întoarse de aceştia va afisa rezultatul cerut.

Vom da doua solutii:

1. Solutia 1 cu textul complet intr-un singur fisier sursa
2. Solutia 2 cu doua texte sursa si unul să îl apeleze pe celalalt prin `exec`.

Solutia 1:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
main(int argc, char* argv[]) {
    int pare = 0, impare = 0, nenum = 0, i, n1, n2;
    for (i = 1; i < argc-1; i += 2) {
        if (fork() == 0) {
            n1 = atoi(argv[i]);   // atoi intoarce 0
            n2 = atoi(argv[i+1]); // si la nenumeric
            if (n1 == 0 || n2 == 0) exit(2);
            exit ((n1 + n2) % 2);
        }
    }
    for (i = 1; i < argc-1; i += 2) {
        wait(&n1);
        switch (WEXITSTATUS(n1)) {
            case 0: pare++;   break;
            case 1: impare++; break;
            default: nenum++;
        }
    }
    printf("Pare %d, impare %d, nenumerice %d\n",pare, impare, nenum);
}
```

Solutia 2:

Se creaza programul `par.c` care primeşte la linia de comandă o pereche de argumente. Din această sursă se va consttiui prin `gcc –o par par.c` executabilul `par`:

```
main(int argc, char *argv[]) {
    int n1, n2;
    n1 = atoi(argv[1]);  // atoi intoarce 0
    n2 = atoi(argv[2]);  // si la nenumeric
    if (n1 == 0 || n2 == 0) exit(2);
    exit ((n1 + n2) % 2);
}
```

Se creaza programul `master.c` care primeşte la linia de comandă n perechi de argumente. El va crea n procese fii şi în fiecare va lansa prin `exec` programul `par`. Din aceasta sursa se va consttiui prin `gcc –o master master.c` executabilul `master`:

```
main(int argc, char* argv[]) {
    int pare = 0, impare = 0, nenum = 0, i, n1, n2;
    for (i = 1; i < argc-1; i += 2) {
        if (fork() == 0) {
            execl(“./par”, “./par”, argv[i], argv[i+1], NULL);
        }
    }
    for (i = 1; i < argc-1; i += 2) {
        wait(&n1);
        switch (WEXITSTATUS(n1)) {
            case 0: pare++;   break;
            case 1: impare++; break;
            default: nenum++;
        }
    }
    printf("Pare %d, impare %d, nenumerice %d\n",pare, impare, nenum);
```

```
}
```

Intrebare la ambele solutii: Ce se întamplă daca `wait` si `switch` nu sunt plasate în cicluri `for` succesive ci în acelaşi `for` care crează procesele fii?

3.4. Semnale Unix; exemple de utilizare

3.4.1. Evitarea proceselor zombie

Versiunea Unix System V Release 4 şi Linux: este suficient ca în partea de iniţializare, înainte de crearea proceselor ce pot deveni zombie:

```
# include <signal.h>
- - -
signal(SIGCHLD, SIG_IGN);
```

Versiunea Unix BSD, efectul unui apel sistem `signal` este valabil o singură dată:

```
#include <signal.h>
- - -
void waiter(){     // Functie de manipulare a apelurilor signal
   wait(0);         // Sterge fiul recent terminat
   signal(SIGCHLD, waiter); // Reinstalare handler signal
}// waiter
- - -
signal(SIGCHLD, waiter); // Plasat în partea de iniţializare
```

3.4.2. Schema client / server: adormire si deşteptare

Programul *server* este pus în adormire şi va fi trezit de fiecare *client* ca să-i satisfacă o cerere, după care intră din nou în adormire. prezentăm două variante de server: *iterativ* şi *concurent*.

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
main() {
    for ( ; ; ) {
        printf("%d doarme . . .\n", getpid());
        kill(getpid(), SIGSTOP); // Doarme, asteptand cereri
```

Server iterativ	**Server concurent**

`printf("Servesc cererea . . .");`
`serveşteCerereClient( . . . );`
	`if (fork() == 0) {`
	`  printf("Servesc cererea. . .");`
	`  serveşteCerereClient( . . . );`
	`}`

```
    }
}
```

Trezirea se poate face fie printr-o comandă:

```
$ kill -SIGCONT pidserver
```

fie printr-un program client de forma:

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
main(int argc, char ** argv) {
    int pidserver = atoi(argv[1]);
    // - - - Clientul afla pid-ul serverului si prepara o cerere
```

```
    kill(pidserver, SIGCONT); //Desteptarea, server!
    // - - - Clientul trateaza raspunsul
}
```

3.4.3. Aflarea unor informaţii de stare

Pentru un program care dureaza mult, se vrea din când în când să se afle stadiul calculelor:

```
#include <stdio.h>
#include <signal.h>

// informatii globale de stare
   int  numar;

void  tipareste_stare(int semnal) {
// Tipareste informatiile de stare solicitate
   printf("Numar= %d\n", numar);
}//handlerul de semnal

main() {
   signal(SIGUSR1,tipareste_stare);
    //- - -
   for(numar=0; ; numar++) {

    // - - -

   }//for
}//main
```

Pentru tipărirea stadiului curent, cunoscând (ps) pidul programului, se dă comanda:

```
$ kill -SIGUSR1 pid
```

3.4.4. Tastarea unei linii în timp limitat

SE cere ca tastarea unei linii de la terminal să se facă în timp limitat (în cazul nostru 5 secunde), altfel se anulează citirea şi programul se aduce în starea dinaintea lansării citirii:

```
#include <stdio.h>
#include <setjmp.h>
#include <sys/signal.h>
#include <unistd.h>
#include <string.h>

   jmp_buf tampon;

   void handler_timeout (int semnal) {
      longjmp (tampon, 1);
   }//handler_timeout

   int t_gets (char *s, int t) {
      char *ret;
      signal (SIGALRM, handler_timeout);
      if (setjmp (tampon) != 0)
         return -2;
      alarm (t);
      ret = fgets (s, 100, stdin);
      alarm (0);
      if (ret == NULL)
         return -1;
```

```
   return strlen (s);
}//t_gets

main () {
   char s[100];
   int v;
   while (1) {
      printf ("Introduceti un string: ");
      v = t_gets (s, 5);
      switch (v) {
         case -1:
            printf("\nSfarsit de fisier\n");
            return(1);
         case -2:
            printf ("timeout!\n");
            break;
         default:
            printf("Sirul dat: %s a.Are %d caractere\n", s, v-1);
      }//switch
   }//while
}//t_gets.c
```

3.4.5. Blocarea tastaturii

Se blochează tastatura până când se tastează parola cu care s-a făcut login:

```
#define _XOPEN_SOURCE
#include <stdio.h>
//#include <sys/types.h>
#include <unistd.h>
#include<string.h>
#include <pwd.h>
#include <shadow.h>
#include <signal.h>
   main() {
      char *cpass, pass[15];
      struct passwd *pwd;
      struct spwd *shd;
      signal(SIGHUP, SIG_IGN);
      signal(SIGINT, SIG_IGN);
      signal(SIGQUIT, SIG_IGN);
      setpwent();
      pwd = getpwuid(getuid());
      endpwent();
      setspent();
      shd = getspnam(pwd->pw_name);
      endspent();
      setuid(getuid()); // Redevin userul real
      for ( ;; ) {
         strcpy(pass, getpass("...tty LOCKED!!"));
         cpass = crypt(pass, shd->sp_pwdp);
         if (!strcmp(cpass, shd->sp_pwdp))
            break;
      }//for
   }//main
//lockTTY.c
// Compilare: gcc -lcrypt -o lockTTY lockTTY.c
// User root: chown root.root lockTTY
// User root: chmod u+s lockTTY
//  Executie: ./lockTTY
```

3.5. Probleme propuse

1. Programul apelat compara doua sau mai multe numere primite ca argumente si returneaza cod 0 daca toate sunt egale, 1 altfel. Programul apelant citeste niste numere si spune daca sunt egale.

2. Programul apelat primeste ca argumente un nume de fisier si une sir de caractere si scrie in fisier sirul oglindit. Programul apelat citeste niste siruri de caractere si concateneaza oglindirile lor.

3. Programul apelat primeste ca argumente niste numere si returneaza cod 0 daca produsul lor este pozitiv, 1 daca e negativ si 2 daca e nul. Programul apelant citeste un sir de numere si afiseaza daca produsul lor este pozitiv, negativ sau zero.

4. Programul apelat primeste ca argumente doua numere naturale si un nume de fisier si scrie in fisier cel mai mic multiplu comun al numerelor. Programul apelant citeste un sir de numere naturale si afisieaza cel mai mic multiplu comun al lor.

5. Programul apelat primeste ca argumente doua numere naturale si un nume de fisier si scrie in fisier cel mai mare divizor comun al numerelor. Programul apelant citeste un sir de numere naturale si afisieaza cel mai mare divizor comun al lor.

6. Programul apelat primeste ca argumente doua numere si un nume de fisier si scrie in fisier produsul numerelor. Programul apelant citeste un sir de numere si afiseaza produsul lor.

7. Programul apelat primeste ca argumente trei nume de fisiere, primele doua continand cate un sir crescator de numere intregi, si scrie in al treilea fisier rezultatul interclasarii sirurilor din primele doua fisiere. Programul apelant citeste un sir de numere intregi, le sorteaza si scrie rezultatul sortarii.

8. Programul apelat primeste ca argumente un nume de fisier si niste siruri de caractere si le concateneaza, rezultatul fiind scris in fisierul dat ca prim argument. Programul apelat citeste niste siruri de caractere si le concateneaza.

9. Programul apelat primeste ca argumente niste numere si returneaza cod 0 daca suma lor este para si 1 altfel. Programul apelant citeste un sir de numere si afiseaza daca suma lor este para sau nu.

10. Programul apelat primeste ca argumente doua numere si returneaza cod 0 daca sunt prime intre ele si 1 altfel. Programul apelant citeste un sir de numere si determina daca sunt doua cate doua prime intre ele.

11. Programul apelat primeste ca argument un numar natural si returneaza cod 0 daca este prim si 1 altfel. Programul principal citeste un numar n si afiseaza numerele prime mai mici sau egale cu n.

12. Programul apelat primeste ca argumente doua nume de fisier si adauga continutul primului fisier la al doilea fisier. Programul apelant primeste un sir de nume de fisiere si concateneaza primele fisiere punand rezultatul in ultimul fisier.

13. Programul apelat primeste ca argumente doua numere si un nume de fisier si adauga in fisier toate numerele prime cuprinse intre cele doua numere date. Programul apelant citest un numar si scrie toate numerele prime mai mici decat numarul dat, apeland celalalt program pentru intervale de cel mult 10 numere.

14. Programul apelat primeste ca argumente doua numere si un nume de fisier si scrie in fisier suma numerelor. Programul apelant citeste un sir de numere si afiseaza suma lor.

15. Programul apelat primeste ca argumente doua sau mai multe numere si returneaza cod 0 daca sunt doua cate doua prime intre ele, 1 altfel. Programul apelant citeste niste numere si spune daca sunt doua cate doua prime intre ele.

OPERATING SYSTEMS
– Seminar 4 –

UNIX INTERPROCESS COMUNICATION (IPC)

1. UNIX INTERPROCESS COMUNICATION (IPC)

- can be performed:
  a. between *local processes* (that are running on the same computer):
     - pipes (unnamed/anonymous/half-duplex pipes)
     - FIFOs (named pipes)
     - message queues
     - shared memory segments
  b. between *remote processes* (that are running on different computers):
     - sockets
- reguli de scriere într-un canal unidirecţional de comunicare prin flux de octeţi:

– orice octet scris nu mai poate fi recuperat şi după el se va scrie octetul următor, fie în scrierea curentă, fie în cea următoare

– dacă un proces doreşte să scrie n octeţi, însă canalul mai are doar p (p < n) octeţi liberi, procesul va putea scrie doar primii p octeţi din cei n octeţi şi scrierea se consideră terminată

– procesul care scrie decide dacă doreşte sau nu să scrie şi ceilalţi n-p octeţi printr-o scriere ulterioară

- reguli de citire dintr-un canal unidirecţional de comunicare prin flux de octeţi:

– un octet odată citit este eliminat din flux şi octetul următor este cel care va fi citit, fie în citirea curentă, fie în cea următoare

– dacă un proces doreşte să citească n octeţi, însă canalul de comunicare are doar p (p < n) octeţi disponibili, procesul va putea citi doar cei p octeţi existenţi şi citirea se consideră terminată

– procesul care citeşte decide dacă doreşte sau nu să obţină şi ceilalţi n-p octeţi printr-o citire ulterioară

2. PIPES (UNNAMED/ANONYMOUS/HALF-DUPLEX PIPES)

- a pipe is *a unidirectional, limited-capacity byte stream* that can be used for communication between *related processes*
- creating a pipe (see `man 3 pipe`):

```
#include <unistd.h>

int pipe(int fd[2]);
```

  - on success, `0` is returned
  - on error, `-1` is returned, and `errno` is set appropiately

- data can travel only in one direction through a pipe
- one end of the pipe is used for writing, and the other end is used for reading
- the array `fd[]` is used to return two file descriptors: `fd[0]` refers to the read end of the pipe, and `fd[1]` refers to the write end of the pipe
- the two descriptors are unique and they are inherited due to the `fork()` call
- each process closes the descriptors that it doesn't need for the pipe
- examples: `pipe_1.c, pipe_2.c`

3. FIFOS (NAMED PIPES)

- *a FIFO (First In First Out)* works much like a regular pipe, but does have some noticeable differences:
  - named pipes exist as a device special file in the file system
  - processes of different ancestry can share data through a named pipe
  - when all I/O is done by sharing processes, the named pipe may remain in the file system for later use
- creating a FIFO from command line (see `man 1 mkfifo, man 1 mknod`):

```
mkfifo ./fifo1
mkfifo -m 0640 /tmp/fifo1

mknod ./fifo1 p
mknod -m 0640 /tmp/fifo1 p
```

- deleting a FIFO from command line:

```
rm /tmp/fifo1
```

- creating a FIFO (see `man 3 fifo`):

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *path, mode_t mode);
```

  or (see `man 2 mknod`):

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int mknod(const char *path, mode_t mode, dev_t dev);
```

- deleting a FIFO (see `man 2 unlink`):

```
#include <unistd.h>

int unlink(const char *path);
```

- example: `reader.c, writer.c`

4. MESSAGE QUEUES

- *a message queue* allows processes to exchange data in the form of messages
- the steps required to work with a message queue:
  - create a message queue
  - get the identifier of the message queue *(msqid)*
  - send/receive messages between processes
  - delete the message queue (optional)
- creating a message queue from the command line (vezi `man 1 ipcmk`):

  `ipcmk -Q`
- deleting an existing message queue from the command line (vezi `man 1 ipcrm`):

  `ipcrm -q msgid`
- displaying information on IPC facilities from the command line (vezi `man 1 ipcs`):

  `ipcs`
- create/get the identifier of a message queue (see `man 2 msgget`):

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflags);
```

- send/receive messages (see `man 2 msgsnd, man 2 msgrvc`):

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int     msgsnd(int msqid, const void *msgp, size_t msgsz, int msflags);
ssize_t msgrcv(int msqid, const void *msgp, size_t msgsz, long msgtype,
               int msflags);
```

- performing control operations on a message queue (see `man 2 msgctl`):

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

- example: `msg_client.c, msg_server.c`

5. SHARED MEMORY SEGMENTS

- *a shared memory segment* is a region of physical memory shared between two or more processes

- the steps required to work with a shared memory segment:
  - create a shared memory segment
  - get the identifier of the shared memory segment *(shmid)*
  - attach the shared memory segment to the address space of the calling process
  - processing data from the shared memory segment as desired
  - detach the shared memory segment from the address space of the calling process
  - delete the shared memory segment (optional)
- creating a shared memory segment from the command line (see `man 1 ipcmk`):

  `ipcmk -M` *`size`*
- deleting a shared memory segment from the command line (vezi `man 1 ipcrm`):

  `ipcrm -m` *`shmid`*
- displaying information on IPC facilities from the command line (vezi `man 1 ipcs`):

  `ipcs`
- create/get the identifier of a shared memory segment (see `man 2 shmget`):

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflags);
```

- attach/detach the shared memory segment (see `man 2 shmat, man 2 shmdt`):

```
#include <sys/types.h>
#include <sys/shm.h>

void* shmat(int shmid, const void *shmaddr, int shmflags);
int   shmdt(const void *shmaddr);
```

- performing control operations on a shared memory segment (see `man 2 shmctl`):

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- example: `shm_client.c, shm_server.c`

REFERENCES:

- Course: `http://www.cs.ubbcluj.ro/~rares/course/os/`
- Seminar: `http://www.cs.ubbcluj.ro/~dbota/SO/sem_04`
- Linux IPC: `https://www.tldp.org/LDP/lpg/node7.html`
- IPC tutorial: `http://beej.us/guide/bgipc/html/single/bgipc.html`

4. Comunicarea între procese Unix: pipe, FIFO, popen, dup2

4.1. Principalele apeluri sistem de comunicare între procese

Tabelul următor prezintă sintaxele principalelor apeluri sistem Unix pentru comunicare între procese:

Funcţii specifice comunicării între procese

`pipe(f)`
`mkfifo(nume, drepturi)`
`FILE *popen(c, "r|w")`
`pclose(FILE *)`
`dup2(fo,fn)`

Prototipurile lor sunt descrise, de regula, in <`unistd.h`> Parametrii sunt:

- `c` este o comandă Unix;
- `f` este un tablou de doi intregi – descriptori de citire / scriere din / in pipe;
- `nume` este numele (de pe disc) al fişierului FIFO, iar `drepturi` sunt drepturile de acces la acesta;
- `fo` si `fn` descriptori de fisiere: `fo` deschis in program cu `open`, `fn` poziţia în care e duplicat `fo`.

In caz de eşec, functiile întorc -1 (NULL la `popen`) si poziţionează `errno` se depisteaza ce eroare a aparut.

Funcţia `popen` are comanda completă (un string), interpretabilă de shell şi apoi executată

4.2. Analizaţi textul sursă

Considerând că toate instrucțiunile din fragmentul de cod de mai jos se execută cu succes, răspundeți la următoarele întrebări:

a) Ce va tipări rularea codului așa cum este?
b) Câte procese se creează, incluzând procesul inițial, dacă lipsește linia 8? Specificați relația părinte fiu dintre aceste procese.
c) Câte procese se creează, incluzând procesul inițial, dacă mutăm instrucțiunea de pe linia 8 pe linia 11 (pornind de la codul dat)? Specificați relația părinte fiu dintre aceste procese.
d) Ce va tipări rularea codului, dacă liniile 16 și 17 se mută în interiorul ramurii else, începând cu linia 11 a codului inițial? Justificați răspunsul.

```
1   int main() {
2       int pfd[2], i, n;
3       pipe(pfd);
4       for(i=0; i<3; i++) {
5           if(fork() == 0) {
6               write(pfd[1], &i, sizeof(int));
7               close(pfd[0]); close(pfd[1]);
8               exit(0);
9           }
10          else {
11              // a se vedea punctele c) si d)
12          }
```

```
13      }
14      for(i=0; i<3; i++) {
15          wait(0);
16          read(pfd[0], &n, sizeof(int));
17          printf("%d\n", n);
18      }
19      close(pfd[0]); close(pfd[1]);
20      return 0;
21  }[1]
```

Răspuns:

a)0, 1, 2 pe linii separate în orice ordine.
b)8 procese, arbore cu 8 procese.
c)4 procese, arbore cu 4 procese.
d)0, 1, 2 pe linii separate întodeauna în această ordine.

4.3. Utilizări simple pipe şi FIFO

Pentru a ilustra modul de lucru cu pipe şi cu FIFO, vom pleca de la exemplul cunoscut de **adunare paralelă rea a patru numere**, exemplu care reclamă necesitatea comunicării între procese. Sursa programului **`add4Rau.c`** este:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main () {
    int a[] = {1,2,3,4};
    if (fork()==0) { // Procesul fiu
       a[0]+=a[1];
       exit(0);
    }
    a[2]+=a[3]; // Procesul parinte
    wait(NULL);
    a[0]+=a[2];
    printf("Suma este %d\n", a[0]);
}
```

Se ştie, suma tipărită va fi 8, nu 10, deoarece informaţia din procesul fiu nu ajunge în părinte. Vom da trei soluţii corecte pentru această problemă, toate vor tipări "**`Suma este 10`**".

Soluţia 1: comunicarea prin pipe este dată în programul **`add4p.c`**:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main () {
    int a[] = {1,2,3,4}, f[2];
    pipe(f);
    if (fork()==0) { // Procesul fiu
        close(f[0]); // Nu trebuie
        a[0]+=a[1];
        write(f[1], &a[0], sizeof(int)); // Scrie suma partiala
        close(f[1]);
        exit(0);
    }
    close(f[1]); // Nu trebuie in procesul parinte
```

[1] nblue

```
        a[2]+=a[3]; // Procesul parinte
        read(f[0], &a[0], sizeof(int));
        close(f[0]);
        wait(NULL);
        a[0]+=a[2];
        printf("Suma este %d\n", a[0]);
    }
```

Soluţia 2: comunicarea prin FIFO cu procesele în aceeaşi sursă. FIFO permite comunicarea între două procese care nu sunt, neapărat, înrudite. Din această cauză, se obişnuieşte ca fişierul FIFO să se creeze în directorul /`tmp`. Această creare se poate face, de exemplu, prin comanda:

```
$ mkfifo /tmp/fifo1
```

Evident, crearea se face înainte de a lansa procesele care o utilizează. Natural, atunci când nu mai avem nevoie de acest FIFO, el se şterge cu comanda:

```
$ rm /tmp/fifo1
```

Sursa pentru această soluţie este `add4f.c`:

```
    #include <stdio.h>
    #include <unistd.h>
    #include <stdlib.h>
    #include <fcntl.h>
    #include <sys/wai
    #include <sys/types.h>
    int main () {
        int a[] = {1,2,3,4}, f;
        if (fork()==0) { // Procesul fiu
            f = open("/tmp/fifo1", O_WRONLY);
            a[0]+=a[1];
            write(f, &a[0], sizeof(int)); // Scrie suma partiala
            close(f);
            exit(0);
        }
        f = open("/tmp/fifo1", O_RDONLY);
        a[2]+=a[3]; // Procesul parinte
        read(f, &a[0], sizeof(int));
        close(f);
        wait(NULL);
        a[0]+=a[2];
        printf("Suma este %d\n", a[0]);
    }
```

Soluţia 3: comunicarea prin FIFO între două procese create din surse diferite. Tabelul următor prezintă fişierele add4fTata.c şi add4fFiu.c care vor comunica între ele:

add4fTata.c	**add4fFiu.c**

`#include <stdio.h>`
`#include <unistd.h>`
`#include <stdlib.h>`
`#include <fcntl.h>`
`#include <sys/types.h>`
`int main () {`
`  int a[] = {1,2,3,4}, f;`
`  f=open("/tmp/fifo1",O_RDONLY);`
`  a[2]+=a[3];`
`  read(f, &a[0], sizeof(int));`
`  close(f);`
`  a[0]+=a[2];`
`  printf("Suma este %d\n",`
	`#include <stdio.h>`
	`#include <unistd.h>`
	`#include <stdlib.h>`
	`#include <fcntl.h>`
	`#include <sys/wait.h>`
	`#include <sys/types.h>`
	`int main () {`
	`  int a[] = {1,2,3,4}, f;`
	`  f = open("/tmp/fifo1", O_WRONLY);`
	`  a[0]+=a[1];`
	`  write(f, &a[0], sizeof(int));`
	`  close(f);`

```
a[0]);
}
```

```
}
```

Inainte de a lansa procesele, trebuie creat FIFO. procesele se pot lansa în orice ordine, deoarece se aşteaptă unul după celălalt. Eventual fiul poate fi lansat în background.

4.4. Joc nim între parinte şi fiu; între n fii succesivi

Este vorba de un joc al marinarilor: se considera **g** grămezi, fiecare grămadă având un număr de pietre. La fiecare pas, jucatorul aflat la mutare elimină un numar nenul de pietre (eventual toate) dintr-o singură grămadă. Jucatorii muta alternativ (sau intr-o ordine prestabilită daca sunt mai mult de doi jucători).. Câştigătorul este cel care ia ultimele pietre.

Exemplul de mai jos prezintă jocul între părinte şi un fiu. Numărul de grămezi se ia de la liniia de comandă, iar numărul de pietre din fiecare grămadă se generează aleator.

Indiferent de numărul de jucători şi de dimensiunile grămezilor, strategia câştigătoare este unică: să se dea următorului jucător dimensiunile grămezilor aşa încât suma modulo 2 (sau exclusiv, operatorul ^ din C) să fie zero.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <time.h>
unsigned int pietre[10]; // gramezile de pietre
int p2f[2], f2p[2];      // canalele pipe intre parinte si fiu
int g;                   // numarul de gramezi
void list(char *n) {
    printf("%s [ ",n);
    for (int i = 0; i < g; i++) printf("%u ",pietre[i]);
    printf("]\n");
    fflush(stdout);
}
void joaca() {
    unsigned int max, scot, xor;
    int pmax, pscot, i;
    for (max = 0, i = 0; i < g; i++)
        if (max < pietre[i]) { max = pietre[i]; pmax = i; }
    if (max == 0) return;
    for ( pscot = 0; pscot < g; pscot++) {
        for (scot = 1; scot < max; scot++) {
            if (pietre[pscot] < scot) continue;
            for (i = 0, xor = 0; i < g; i++) {
                if (i == pscot) xor ^= (pietre[i] - scot);
                else            xor ^= pietre[i];
            }
            if (xor == 0) {
                pietre[pscot] -= scot;
                return;
            }
        }
    }
    pietre[pmax]--;
    return;
}
int main (int argc, char **argv) {
    srand(time(0));
    int i, f;
    g = (argc > 1)? atoi(argv[1]): 3;
```

```
    if (g > 10) g = 10;
    for (i = 0; i < g; i++) pietre[i] = rand() % 200 + 1;
    list("start");
    pipe(p2f); pipe(f2p);
    write(p2f[1], pietre, g*sizeof(unsigned int));
    if (fork() == 0) {  // Fiu
        for ( ; ; ) {
            int r;
            read(p2f[0], pietre, g*sizeof(unsigned int));
            if (pietre[0] == -1) {
                close(p2f[0]); close(p2f[1]); close(f2p[0]); close(f2p[1]);
                exit(0);
            }
            for (i = 0, r = 0; i < g; i++) if (pietre[i] == 0) r++;
            if (r == g) {
                printf("Fiu castiga, Parinte pierde!\n"); fflush(stdout);
                pietre[0] = -1;
                write(f2p[1], pietre, g*sizeof(unsigned int));
                close(p2f[0]); close(p2f[1]); close(f2p[0]); close(f2p[1]);
                exit(0);
            }
            list("Fiu_ante"); joaca(); list("Fiu_post");
            write(f2p[1], pietre, g*sizeof(unsigned int));
        }
    } else {            // Parinte
        for ( ; ; ) {
            int r;
            read(f2p[0], pietre, g*sizeof(unsigned int));
            if (pietre[0] == -1) {
                close(p2f[0]); close(p2f[1]); close(f2p[0]); close(f2p[1]);
                wait(NULL); exit(0);
            }
            for (i = 0, r = 0; i < g; i++) if (pietre[i] == 0) r++;
            if (r == g) {
                printf("Parinte castiga, Fiu pierde!\n"); fflush(stdout);
                pietre[0] = -1;
                write(p2f[1], pietre, g*sizeof(unsigned int));
                close(p2f[0]); close(p2f[1]); close(f2p[0]); close(f2p[1]);
                return 0;
            }
            list("Parinte_ante"); joaca(); list("Parinte_post");
            write(p2f[1], pietre, g*sizeof(unsigned int));
        }
    }
}
```

Propuneri de extinderi: Să joace **n** procese care să comunice circular între ele prin **n-1** pipe-uri. Sau un FIFO din care să se alimenteze primul proces, să urmeze o legare prin pipe (|) intre procese, iar ultimul să scrie în acelaşi FIFo ca noi date de intrare pentru primul proces.

Sursa următoare simuleaza jocul la care participă **n** fii. Se construiesc **n** pipe-uri de comunicare între fii. perechea 0 din pipe este destinată comunicării între primul şi ultimul fiu şi la pornire părintele dă datele iniţiale primului fiu. Pentru restul, perechea pipe **i** este folosită pentru comunicarea dintre fiul **i-1** şi fiul **i**.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <time.h>
unsigned int pietre[10]; // gramezile de pietre
int p[100];              // canalele pipe fii
int g, n;                // numarul de gramezi, numar de fii
void list(char *n, int f) {
    printf("[ ");
    for (int i = 0; i < g; i++) printf("%05u ",pietre[i]);
```

```
    printf("] %02d %s\n",f, n);
    fflush(stdout);
}
void joaca() {
    unsigned int max, scot, xor;
    int pmax, pscot, i;
    for (max = 0, i = 0; i < g; i++)
        if (max < pietre[i]) { max = pietre[i]; pmax = i; }
    if (max == 0) return;
    for ( pscot = 0; pscot < g; pscot++) {
        for (scot = 1; scot < max; scot++) {
            if (pietre[pscot] < scot) continue;
            for (i = 0, xor = 0; i < g; i++) {
                if (i == pscot) xor ^= (pietre[i] - scot);
                else            xor ^= pietre[i];
            }
            if (xor == 0) {
                pietre[pscot] -= scot;
                return;
            }
        }
    }
    pietre[pmax]--;
    return;
}
int main (int argc, char **argv) {
    srand(time(0));
    int i, f;
    n = (argc > 1)? atoi(argv[1]): 5;
    g = (argc > 2)? atoi(argv[2]): 3;
    if (n > 50) n = 50;
    if (g > 10) g = 10;
    for (i = 0; i < g; i++) pietre[i] = rand() % 200 + 1;
    list("Start: parinte", -1);
    for (i = 0; i < n; i++) pipe(p + 2*i);
    write(p[1], pietre, g*sizeof(unsigned int));
    for (f = 0; f < n; f++) if (fork() == 0) {  // Fiul f
        for (i = 0; i < 2*n; i++)
            if (i != 2*f && i != (f!=n-1)?p[2*f+3]:p[1]) close(p[i]);
        for ( ; ; ) {
            int r;
            read(p[2*f], pietre, g*sizeof(unsigned int));
            list("Fiu_ante", f);
            if (pietre[0] == -1) {
                close(p[2*f]); close((f!=n-1)?p[2*f+3]:p[1]);
                exit(0);
            }
            for (i = 0, r = 0; i < g; i++) if (pietre[i] == 0) r++;
            if (r == g) {
                list("PIERDE", f);
                pietre[0] = -1;
                write((f!=n-1)?p[2*f+3]:p[1], pietre, g*sizeof(unsigned int));
                close(p[2*f]); close((f!=n-1)?p[2*f+3]:p[1]);
                exit(0);
            }
            joaca();
            write((f!=n-1)?p[2*f+3]:p[1], pietre, g*sizeof(unsigned int));
            list("Fiu_post", f);
        }
    }
    for (i = 0; i < n; i++) wait(NULL);
}
```

După compilare şi lansarea **`./a.out | sort -r > a`** iată o porţiune deînceput şi sfârşit al acestei ieşiri. Intrebare: de ce credeţi că a fost nevoie de o listare aşa de "ciudată" şi de ce a fost nevoie de ieşire cu un sort?

```
[ 00008 00033 00061 ] -1 Start: parinte
[ 00008 00033 00061 ] 04 Fiu_ante
[ 00008 00033 00061 ] 03 Fiu_ante
[ 00008 00033 00061 ] 02 Fiu_ante
[ 00008 00033 00061 ] 01 Fiu_ante
[ 00008 00033 00061 ] 00 Fiu_ante
[ 00008 00033 00041 ] 04 Fiu_post
[ 00008 00033 00041 ] 04 Fiu_ante
[ 00008 00033 00041 ] 03 Fiu_post
[ 00008 00033 00041 ] 03 Fiu_ante
[ 00008 00033 00041 ] 02 Fiu_post
[ 00008 00033 00041 ] 02 Fiu_ante
[ 00008 00033 00041 ] 01 Fiu_post
[ 00008 00033 00041 ] 01 Fiu_ante
[ 00008 00033 00041 ] 00 Fiu_post
[ 00008 00033 00041 ] 00 Fiu_ante
[ 00008 00033 00040 ] 04 Fiu_post

---

[ 00000 00001 00001 ] 02 Fiu_post
[ 00000 00001 00001 ] 02 Fiu_ante
[ 00000 00001 00001 ] 01 Fiu_post
[ 00000 00001 00001 ] 01 Fiu_ante
[ 00000 00001 00001 ] 00 Fiu_post
[ 00000 00001 00001 ] 00 Fiu_ante
[ 00000 00000 00001 ] 04 Fiu_post
[ 00000 00000 00001 ] 04 Fiu_ante
[ 00000 00000 00001 ] 03 Fiu_post
[ 00000 00000 00001 ] 03 Fiu_ante
[ 00000 00000 00001 ] 02 Fiu_post
[ 00000 00000 00001 ] 02 Fiu_ante
[ 00000 00000 00001 ] 01 Fiu_post
[ 00000 00000 00001 ] 01 Fiu_ante
[ 00000 00000 00001 ] 00 Fiu_post
[ 00000 00000 00001 ] 00 Fiu_ante
[ 00000 00000 00000 ] 04 PIERDE
```

4.5. Simulare sh pentru who | sort şi who | sort | cat (dup2)

Problema 4: Simularea unui shell care executa comanda: `$ who | sort`.

Pentru simulare, programul principal va crea doua procese fii in care va lansa, prin `exec`, comenzile `who` si `sort`. Inainte de crearea acestor fii, va crea un pipe pe care il va da celor doi fii ca sa comunice intre ei: `who` isi va redirecta iesirea standard in acest pipe./t cu ajutorul apelului `dup2`, iar `sort` va avea ca intrare standard acest pipe, redirectat de asemenea cu `dup2`.

O extindere naturală este conectarea în pipe a trei programe, de exemplu `who | sort | cat`. (De aici, generalizarea la un pipeline între n comenzi este uşor de făcut). Sursele celor două programe sunt date în tabelul următor:

who | sort	**who | sort | cat**

`#include <stdio.h>`
`#include <unistd.h>`
`#include <sys/wait.h>`
`int main () {`
`  int p[2];`
`  pipe (p);`
`  if (fork() == 0) {`
`  close (p[0]);`
`  dup2 (p[1], 1);`
`  execlp ("who", "who", NULL);`
	`#include <stdio.h>`
	`#include <unistd.h>`
	`#include <sys/wait.h>`
	`int main () {`
	`  int p[2], q[2];`
	`  pipe (p);`
	`  pipe (q);`
	`  if (fork() == 0) {`
	`  close (p[0]);`
	`  close (q[0]);`
	`  close (q[1]);`

```
    } else if (fork() == 0) {
        close (p[1]);
        dup2 (p[0], 0);
        execlp   ("sort",   "sort",
NULL);
    } else {
        close (p[0]);
        close (p[1]);
        wait (NULL);
        wait (NULL);
    }
}
```

```
        dup2 (p[1], 1);
        execlp   ("who",    "who",
NULL);
    } else if (fork() == 0) {
        close (p[1]);
        close (q[0]);
        dup2 (p[0], 0);
        dup2 (q[1], 1);
        execlp   ("sort",   "sort",
NULL);
    } else if (fork() == 0) {
        close (p[0]);
        close (p[1]);
        close (q[1]);
        dup2 (q[0], 0);
        execlp   ("cat",    "cat",
NULL);
    } else {
        close (p[0]);
        close (p[1]);
        close (q[0]);
        close (q[1]);
        wait (NULL);
        wait (NULL);
        wait (NULL);
    }
}
```

Principiul este simplu: dacă avem n comenzi în pipeline, atunci trebuie construite n-1 pipe-uri. Procesul ce execută prima comandă îşi va redirecta ieşirea standard în primul pipe. Procesul ce execută ultima comandă îşi va redirecta intrarea standard in ultimul pipe. Procesele ce execută comenzile intermediare, să zicem procesul i cu i>1 şi i < n-1, va avea ca intrare standard pipe-ul i-1 şi ca ieşire standard pipe-ul i.

Evident, în locul comenzilor `who, sort, cat` pot să apară orice comenzi, cu orice argumente.

4.6. Paradigma client / server; exemple

Problema pe care o vom rezolva folosind paradigma client / server este următoarea:

Să se scrie un program **server** care primeşte în FIFO-ul `/tmp/CERERE` un string de 20 caractere:

`numar` întreg pozitiv fără semn > 1, exact 10 caractere, cu completare de zerouri la stânga	**`nume`** din exact 10 caractere, fără spaţii albe (blank, tab, \n, \r etc.).

Serverul descompune **`numar`** în factori primi şi scrie această descompunere în FIFO-ul **`/tmp/nume`**

Să se scrie un program **client** care construie un string de 10 caractere **`nume`** care să reprezinte unic procesul client (nume putând să fie, de exemplu, PID-ul procesului completat la stânga cu zerouri). Clientul primeşte la linia de comandă un număr întreg de maximum 10 cifre, formează cererea ca mai sus şi o scrie într-un FIFO **`/tmp/CERERE`**. Apoi citeşte răspunsul de la server pe FIFO-ul `/tmp/nume`, mai

întâi lungimea stringului, apoi stringul răspuns. In final tipăreşte răspunsul primit pe ieşirea lui standard, după care şterge FIFO-ul **`/tmp/nume`**.

Vom da două soluţii pentru **server**:

- Un **server iterativ**, care citeşte o cerere, o execută, trimite răspunsul, apoi iarăşi revine la citirea unei noi cereri ş.a.m.d.
- Un **server concurent**, care citeşte o cerere, crează un proces fiu căruia îi trimite cererea, după care revine la citirea unei noi cereri ş.a.m.d. Toată sarcina de tratare a cererii revine procesului fiu, care evoluează în acelaşi timp cu preluarea de către server a unor noi cereri.

Intrebare: de ce este nevoie ca cererea să aibă această structură rigidă?

Vom începe cu **prezentarea `client.c`**. Sursa acestuia este:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
int main (int argc, char* argv[]) {
    char cerere[21] = "01234567890123456789",
    fiforasp[16] = "/tmp/0123456789", rasp[500];
    int lrasp, fc, fr;
    sprintf(rasp, "%010lu", atol(argv[1]));
    memcpy(cerere, rasp, 10);
    sprintf(rasp, "%010d", getpid());
    memcpy(&cerere[10], rasp, 10);
    memcpy(&fiforasp[5], rasp, 10);
    while ((fc = open("/tmp/CERERE", O_WRONLY)) < 0) sleep(1); // !!
    printf("Cerere: %s\n", cerere);
    write(fc, cerere, 20);
    close(fc);
    while ((fr = open(fiforasp, O_RDONLY)) < 0) sleep(1); // !!
    read(fr, &lrasp, sizeof(int));
    read(fr, rasp, lrasp);
    printf("Raspuns: %s\n", rasp);
    close(fr);
    unlink(fiforasp);
}
```

Acţiunea clientului este cea descrisă mai sus. Trebuie să atragem atenţia asupra celor două apeluri sistem `open` care deschid FIFO de cerere şi de răspuns. Sarcina creării FIFO-urilor revine serverului. Ordinea celor două `open` trebuie să fie aceeaşi şi la server, altfel se ajunge la deadlock.

Se observă că `open` se reia până când se reuşeşte deschiderea. Aici reluarea se face din secundă în secundă (se poate şi mai des sau chiar reluare fără pauză). Motivul acestei reluări este acela că dacă serverul ocupă procesorul după client, este posibil ca unul din FIFO-uri să nu fie creat la momentul `open` al clientului. Dacă se întâmplă asta, se va trece de `open`, în ciuda regulilor FIFO, şi se semnalează `"No such file or directory"`.

Cele două servere, serveriter,c şi serverconc.c sunt foarte asemănătoare. Le prezentăm simultan în tabelul de mai jos.

`serveriter.c`	**`serverconc.c`**

`#include <stdio.h>`
`#include <unistd.h>`
`#include <stdlib.h>`
`#include <fcntl.h>`
`#include <sys/stat.h>`
`#include <string.h>`
	`#include <stdio.h>`
	`#include <unistd.h>`
	`#include <stdlib.h>`
	`#include <fcntl.h>`
	`#include <sys/stat.h>`
	`#include <string.h>`
	`# include <signal.h>`

```
int main (int argc, char* argv[])
{

char          cerere[21]          =
"01234567890123456789";
char
fiforasp[16]="/tmp/0123456789";
    char rasp[500], factor[50];
    int lrasp, fc, fr;
    unsigned long n, p, d;
    unlink("/tmp/CERERELs /tmp
");
    mkfifo("/tmp/CERERE", 0777);
    fc    =    open("/tmp/CERERE",
O_RDONLY);
    for ( ; ; ) {
        if  (read(fc,  cerere,  20)
!= 20)
            continue;


memcpy(&fiforasp[5],&cerere[10],10
);
        cerere[10] = '\0';
        n = atol(cerere);
        sprintf(rasp,  "%lu  =  ",
n);
        d = 2;
        while(n > 1) {
            p = 0;
            while(n % d == 0) {
                p = p + 1;
                n = n / d;
            }
            if (p > 0) {
                sprintf(factor,
                    "%lu^%lu     *
",d,p);
                strcat(rasp,
factor);
            }
            d = d + 1;
        }
        rasp[strlen(rasp)  -  3]  =
'\0';
        lrasp = strlen(rasp) + 1;
        unlink(fiforasp);
        fr    =    mkfifo(fiforasp,
0777);
        fr    =    open(fiforasp,
O_WRONLY);
        write(fr,          &lrasp,
sizeof(int));
        write(fr, rasp, lrasp);
        close(fr);
```

```
int main (int argc, char* argv[])
{
    signal(SIGCHLD, SIG_IGN);
char                    cerere[21]=
"01234567890123456789";
char                  fiforasp[16]=
"/tmp/0123456789";
    char rasp[500], factor[50];
    int lrasp, fc, fr;
    unsigned long n, p, d;
    unlink("/tmp/CERERE");
    mkfifo("/tmp/CERERE", 0777);
    fc    =    open("/tmp/CERERE",
O_RDONLY);
    for ( ; ; ) {
        if  (read(fc,  cerere,  20)
!= 20)
            continue;
        if (fork() == 0) {

memcpy(&fiforasp[5],&cerere[10],10
);
            cerere[10] = '\0';
            n = atol(cerere);
            sprintf(rasp,  "%lu  =
", n);
            d = 2;
            while(n > 1) {
                p = 0;
                while(n  %  d  ==  0)
{
                    p = p + 1;
                    n = n / d;
                }
                if (p > 0) {

sprintf(factor,
                        "%lu^%lu   *
",d,p);
                    strcat(rasp,
factor);
                }
                d = d + 1;
            }
            rasp[strlen(rasp) - 3]
= '\0';
            lrasp = strlen(rasp) +
1;
            unlink(fiforasp);
            fr  =  mkfifo(fiforasp,
0777);
            fr   =   open(fiforasp,
O_WRONLY);
            write(fr,      &lrasp,
sizeof(int));
```

```
    }
}
```

```
            write(fr,        rasp,
lrasp);
            close(fr);
        }
    }
}
```

Serverul concurent, după ce primeşte cererea, crează un proces fiu, care implicit primeşte această cerere. Pentru ca fii să nu rămână în starea zombie, prima instrucţiune din server este apelul sistem `signal`.

Client / server ce verifică dreprurile unui fişier. Clientul primeşte la linia de comandă trei cifre octale şi un nume de fişier. Ii trimite serverului aceste informaţii împreună cu PID-ul clientului. Pentru uniformitate, se trimite la server de exact 109 caractere: PID-ul pe 6, drepturile pe 3 şi numele de fişier pe 100. De asemenea, clientul crează un FIFO cu numele PID-ului, unde să primească răspunsul. Serverul verifică drepturile şi trimite pe FIFO de răspuns un string "Ok" sau "Nu" dacă fişierul are sau nu drepturile.

Server.c

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
int main() {
    char   cerere[109],   mode[6],
pid[7];
    unlink("/tmp/cerere");  //  Se
sterge preventiv
    mkfifo("/tmp/cerere",   0777);
//  Se  creaza  FIFO  de  cereri  cu
drepturi rw
    int  f  =  open("/tmp/cerere",
O_RDONLY);
    for ( ; ; ) {
        if  (read(f,  cerere,  109)
== 0) { // Nu este solicitat de
client
          sleep(1);   //   Dupa   o
scurta pauza verifica
          continue;   //   sosirea
unei noi cereri.
        }
        pid[6] = 0;    struct stat
b;
        stat(cerere+9, &b);
        sprintf(mode,"%d",
b.st_mode);
        strncpy(pid, cerere, 6);
printf("Server:  cerere=%s  pid=%s
mode=%s\n",cerere, pid, mode);
        int    g    =    open(pid,
O_WRONLY);
        if      (strncmp(cerere+6,
mode, 3) == 0) write(g, "Ok", 3);
```

Client.c

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
int main (int argc, char **argv) {
    char    cerere[109],    pid[7],
raspuns[3] ;
    sprintf(pid,"%06d",getpid());
    pid[6] = 0;
    strncpy( cerere, pid, 6);
    strncpy( cerere+6, argv[1],3);
    strcpy( cerere+9, argv[2]);
printf("Client:  cerere=%s  pid=%s
",cerere, pid);
    unlink(pid);     //  Se  sterge
preventiv
    mkfifo(pid, 0777); // Se creaza
FIFO cu raspunsul
    int  f  =  open("/tmp/cerere",
O_WRONLY);
    write(f, cerere, 109);
    close(f);
    int g = open(pid, O_RDONLY);
    read(g, raspuns, 3);
printf("raspuns=%s\n", raspuns);
    if (strcmp(raspuns, "Ok") == 0)
printf("Totul este Ok!\n");
    else
printf("Drepturile     au     fost
modificate\n");
    close(g);
    unlink(pid);      // Se sterge
FIFO de raspuns
    return 0;
}
```

```
        else  write(g, "Nu", 3);
        close(g);
    } }
```

4.7. Exemple de utilizare popen

Utilizare popen cu scriere in intrarea standard pentru comanda lansată: de exemplu, scrierea in ordine alfabetică a argumentelor şi a variabilelor de mediu:

```
#include <stdio.h>
main (int argc, char *argv[], char *envp[]) {
    FILE *f;  int i;
    f = popen("sort", "w");
    for (i=0; argv[i]; i++ )
        fprintf(f, "%s\n", argv[i]);
    for (i=0; envp[i]; i++ )
        fprintf(f, "%s\n", envp[i]);
    pclose (f);
}// Rezultatul, pe iesirea standard
```

Utilizare popen cu preluarea iesirii standard a comenzii lansate, de exemplu să se verificare că userul `florin` este logat:

```
#include <stdio.h>
#include <string.h>
main () {
    FILE *f;  char l[1000], *p;
    f = popen("who", "r");
    for ( ; ; ) {
        p = fgets(l, 1000, f);
        if (p == NULL) break;
        if (strstr(l, "florin")) {
            printf("DA\n");
            pclose (f);
            return;
        }
    }
    printf("NU\n");
    pclose (f);
}
```

Lansarea în paralel a mai multor programe filtru, folosind mai multe `popen`. Presupunem ca avem un program lansabil: `$ filtru intrare iesire` care transforma fisierul `intrare` in fisierul `iesire` dupa reguli stabilite de user. Se cere un program care primeste la linia de comandă mai multe nume de fisiere de intrare, care să fie filtrate în procese paralele în fisiere de ieşire. Programul este:

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
main(int argc, char* argv[]) {
    int i;
    char c[50][200];
    FILE *f[50];
    for (i=1; argv[i]; i++) {
        strcpy(c[i], "./filtru ");
        strcat(c[i], argv[i]);
        strcat(c[i], " ");
        strcat(c[i], argv[i]);
        strcat(c[i], ".FILTRU");
```

```
            popen(c[i], "r");
            f[i] = popen(c[i], "r");
            pclose(f[i]);
        }
    }
```

4.8. Probleme propuse

1. Se dă fişierul **grep.c** care conţine fragmentul de cod de mai jos şi care se compilează în directorul personal al utilizatorului sub numele **grep**. Răspundeţi la următoarele întrebări, considerând că toate instrucţiunile se
execut cu succes.

a) Enumeraţi şi explicaţi valorile posibile ale variabilei **n**.

b) Ce vor afişa pe ecran următoarele rulări, considerând că directorul personal al utilizatorului nu se află în variabila de mediu PATH

```
1. grep grep grep.c
2. ./grep grep grep.c
3. ./grep grep
```

```
1  int main(int c, char** v) {
2    int p[2], n;
3    char s[10] = "ceva";
4    pipe(p);
5    n = fork();
6    if(n == 0) {
7      close(p[0]);
8      printf("înainte\n");
9      if(c > 2)
10       execlp("grep","grep",v[1],v[2],NULL);
11     strcpy(s, "dup ");
12     write(p[1], s, 6);
13     close(p[1]);
14     exit(0);
15   }
16   close(p[1]);
17   read(p[0], s, 6);
18   close(p[0]);
19   printf("%s\n", s);
20   return 0;
21 }
```

2. Considerând că toate instrucțiunile din fragmentul de cod de mai jos se execută cu succes, răspundeți la următoarele întrebări:

a) Desenaţi ierarhia proceselor create, incluzând şi procesul părinte.

b) Daţi fiecare linie afişată de program, împreună cu procesul care o tipăreşte.

c) Câte caractere sunt citite din pipe?

d) Cum este afectată terminarea proceselor dacă lipseşte linia 20?

e) Cum este afectată terminarea proceselor dacă lipsesc liniile 20 şi 21?

```
1  int main() {
2      int p[2], i=0;
3      char c, s[20];
4      pipe(p);
5      if(fork() == 0) {
6          close(p[1]);
7          while(read(p[0], &c, sizeof(char))) {
8              if (i<5 || i > 8) {
9                  printf("%c", c);
```

```
10              }
11              i++;
12          }
13          printf("\n"); close(p[0]);
14          exit(0);
15      }
16      printf("Result: \n");
17      strcpy(s, "exam not passed");
18      close(p[0]);
19      write(p[1], s, strlen(s)*sizeof(char));
20      close(p[1]);
21      wait(NULL);
22      return 0;
23  }
```

3. Clientul transmite serverului un nume de fisier iar serverul intoarce clientului continutul fisierului indicat sau un mesaj de eroare in cazul ca fisierul dorit nu exista.

4. Clientul ii transmite serverului un nume de utilizator, iar serverul ii returneaza clientului datele la care utilizatorul respectiv s-a conectat.

5. Clientul ii transmite serverului un nume de server Unix, si primeste lista tuturor utilizatorilor care lucreaza in acel moment la serverul respectiv.

6. Clientul ii transmite serverului un nume de utilizator iar serverul ii intoarce clientului numarul de procese executate de utilizatorul respective.

7. Clientul ii transmite serverului un nume de fisier si primeste de la acesta un mesaj care sa indice tipul fisierului sau un mesaj de eroare in cazul in care fisierul nu exista.

8. Clientul ii transmite serverului un nume de director si primeste de la acesta lista tuturor fisierelor text din directorul respectiv, respectiv un mesaj de eroare daca directorul respectiv nu exista.

9. Clientul ii transmite serverului un un nume de director, iar serverul ii retransmite clientului numarul total de bytes din toate fisierele din directorul respectiv.

10. Clientul ii transmite serverului un nume de fisier iar serverul intoarce clientului numarul de linii din fisierul respectiv.

11. Clientul ii transmite serverului un nume de fisier si un numar octal. Serverul va verifica daca fisierul respectiv are drepturi de acces diferite de numarul indicat. Daca drepturile coincid, va transmite mesajul "Totul e OK!" daca nu va seta drepturile conform numarului indicat si va transmite mesajul "Drepturile au fost modificate".

12. Clientul ii transmite serverului un nume de director iar serverul ii intoarce clientului continutul directorului indicat, respectiv un mesaj de eroare in cazul in care acest director nu exista.

13. Clientul ii transmite serverului un nume de utilizator, iar serverul ii intoarce clientului numele complet al utilizatorului si directorul personal.

14. Clientul ii transmite serverului un nume de fisier, iar serverul ii intoarce clientului numele tuturor directoarelor care contin fisierul indicat.

15. Clientul ii transmite serverului un nume de utilizator, iar serverul ii returneaza informatiile indicate de "finger" pentru utilizatorul respectiv, respectiv un mesaj de eroare daca numele respectiv nu indica un utilizator recunoscut de sistem.

16. Clientul ii transmite serverului o comanda Unix, iar serverul o executa si retransmite clientului rezultatul executiei. In cazul in care comanda este invalida, serverul va transmite un mesaj corespunzator.

OPERATING SYSTEMS
– Seminar 5 –

CONCURRENT PROGRAMMING IN UNIX

1. BASIC CONCEPTS

- a proces is *a calculation that can be executed concurrently or in parallel with other calculations*
- a thread is *the execution entity in a process, composed of an execution context and a sequence of instructions to execute*
- a thread performs a procedure/function, in the same process, concurrent or in parallel with other threads
- all threads within a process share the same address space
- all threads within a process share instructions, most data and the execution context
- each thread has an unique identifier (TID), an unique set of registers and its own stack
- the only space occupied exclusively by a thread is the execution stack
- multithreading advantages:
  - creating a thread takes less time than creating a process
  - context switch time between threads is lower compared to process context switch
  - multiple threads in a single process may execute in parallel on many processors (cores)
  - communication between multiple threads is easier, as the threads within a process share the same address space

2. POSIX THREADS

- are implemented by `pthreads (POSIX threads)` library
- creating a thread (see `man 3 pthread_create`):

```
#include <pthreads.h>

int pthread_create(pthread_t *tid, pthread_attr_t *attr,
                   void*(*start_routine)(void *), void *arg);
```

- waiting for thread termination (see `man 3 pthread_join`):

```
#include <pthreads.h>

int pthread_join(pthread_t tid, void **retval);
```

- terminate the calling thread (see `man 3 pthread_exit`):

```
#include <pthreads.h>

void pthread_exit(void *retval);
```

- send a cancellation request to a thread (see `man 3 pthread_cancel`):

```
#include <pthreads.h>

int pthread_cancel(pthread_t tid);
```

- examples: `/examples/threads/thread_1.c, thread_2.c, thread_3.c, thread_4.c`
- compiling:

```
gcc -Wall -o thread_1 thread_1.c -lpthread
gcc -Wall -o thread_1 thread_1.c -pthread
```

3. THREADS SYNCRONIZATION

THE OBJECTS USED FOR SYNCRONIZATION

Object	Type	Creating/destroying	Operations
Mutex (MUtual EXclusion)	`pthread_mutex_t`	`pthread_mutex_init` `pthread_mutex_destroy`	`pthread_mutex_lock` `pthread_mutex_unlock`

RW lock (Read-Write lock)	`pthread_rwlock_t`	`pthread_rwlock_init`
		`pthread_rwlock_destroy`
			`pthread_rwlock_wrlock`
			`pthread_rwlock_rdlock`
			`pthread_rwlock_unlock`

Condition variable	`pthread_cond_t`	`pthread_cond_init`
		`pthread_cond_destroy`
			`pthread_cond_wait`
			`pthread_cond_signal`
			`pthread_cond_broadcast`

Semaphore	`sem_t`	`sem_init` `sem_destroy`	`sem_wait` `sem_post`

3.1. MUTEXES (Mutual Exclusion)

- are implemented by `pthreads` library
- creating a mutex (see `man 3 pthread_mutex_init`):

```
#include <pthreads.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
```

or by using a macro (the mutex is statically allocated in this case):

```
#include <pthreads.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- locking/unlocking a shared resource (see `man 3 pthread_mutex_lock`):

```
#include <pthreads.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- destroying a mutex (see `man 3 pthread_mutex_destroy`):

```
#include <pthreads.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- example: `/examples/locks/lock_2.c`

3.2. POSIX SEMAPHORES

- are implemented by operating system
- creating a semaphore (see `man 3 sem_init`):

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- locking/unlocking a shared resource (see `man 3 sem_wait`):

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

- destroying a semaphore (see `man 3 sem_destroy`):

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

- example: `/examples/locks/lock_3.c`

3.3. READ-WRITE LOCKS

- are implemented by `pthreads` library
- creating a RW lock (see `man 3 pthread_rwlock_init`):

```
#include <pthreads.h>

int pthread_rwlock_init(pthread_rwlock_t *rwlock,
                        const pthread_rwlockattr_t *attr);
```

- locking/unlocking a shared resource (see `man 3 pthread_rwlock_rdlock`):

```
#include <pthreads.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

- destroying a RW lock (see `man 3 pthread_rwlock_destroy`):

```
#include <pthreads.h>

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

- example: `/examples/locks/lock_4.c`

3.4. CONDITION VARIABLES

- are implemented by `pthreads` library
- creating a condition variable (see `man 3 pthread_cond_init`):

```
#include <pthreads.h>

int phtread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
```

or by using a macro (the condition variable is statically allocated in this case):

```
#include <pthreads.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- locking/unlocking a shared resource (see `man 3 pthread_cond_wait`):

```
#include <pthreads.h>

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- destroying a condition variable (see `man 3 pthread_cond_destroy`):

```
#include <pthreads.h>

int pthread_cond_destroy(pthread_cond_t *cond);
```

- example: `/examples/locks/lock_5.c`

REFERENCES:

- Course: `http://www.cs.ubbcluj.ro/~rares/course/os/`
- POSIX threads:

  `http://pages.cs.wisc.edu/~travitch/pthreads_primer.html`

  `https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html`
- Thread synchronization:

  `http://www.informit.com/articles/article.aspx?p=2085690&seqNum=6`

  `https://docs.oracle.com/cd/E19120-01/open.solaris/816-5137/index.html`

5. Pthreads; sincronizări cu mutex, cond, barrier

5.1. Principalele tipuri de date şi funcţii de lucru cu threaduri

Tabelul următor prezintă principalele fişiere header, tipuri de date şi funcţii care lucrează cu threaduri:

Fişere header	`<pthread.h>`
Specificare biblioteci	`-pthread`

Tipuri de date	`pthread_t`
	`pthread_mutex_t`
	`pthread_cond_t`
	`pthread_barrier_t`
	`pthread_rwlock_t`
	`sem_t`

Funcţii de creare thread şi aşteptare terminare	`pthread_create`
	`pthread_join`
	`pthread_exit`

Variabile mutex	`pthread_mutex_init`
	`pthread_mutex_lock`
	`pthread_mutex_unlock`
	`pthread_mutex_destroy`

Variabile condiţionale	`pthread_cond_init`
	`pthread_cond_wait`
	`pthread_cond_signal`
	`pthread_cond_broadcast`
	`pthread_cond_destroy`

Bariere	`pthread_barrier_init`
	`pthread_barrier_destroy`
	`pthread_barrier_wait`

Variabile reader/writer	`pthread_rwlock_init`
	`pthread_rwlock_wrlock`
	`pthread_rwlock_rdlock`
	`pthread_rwlock_unlock`
	`pthread_rwlock_destroy`

Semafoare	`sem_init`
	`sem_wait`
	`sem_post`
	`sem_destroy`

Funcţia de descriere a acţiunii threadului	`void * work(void* a)`

5.2. Capitalizarea cuvintelor dintr-o listă de fişiere text

Dorim sa transformam un fişier text într-un alt fişier text, cu acelaşi conţinut, dar în care toate cuvintele din el sa înceapă cu literă mare. Un astfel de program / procedura o vom numi **capitalizare** si se apeleaza furnizand doi parametri: **fisierintrare fisieriesire**

Ne propunem să prelucrăm simultan mai multe astfel de fisiere. Programul primeste numele a **n** fisiere de intrare, iar fisierele de iesire vor primi acelasi nume la care i se adauga terminatia .CAPIT. Programul va crea câte un thread pentru fiecare pereche de fişiere. Sursa `capitalizari.c` este:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAXLINIE 1000
pthread_t tid[100];
// pthread_t tid; // Vezi comentariul de la sfârşitul sursei
void* ucap(void* numei) {
    printf("Threadul: %ld ...> %s\n", pthread_self(), (char*)numei);
    FILE *fi, *fo;
    char linie[MAXLINIE], numeo[100], *p;
    strcpy(numeo, (char*)numei);
    strcat(numeo, ".CAPIT");
    fi = fopen((char*)numei, "r");
    fo = fopen(numeo, "w");
    for ( ; ; ) {
        p = fgets(linie, MAXLINIE, fi);
        linie[MAXLINIE-1] = '\0';
        if (p == NULL) break;
        if (strlen(linie) == 0) continue;
        linie[0] = toupper(linie[0]); // Cuvant incepe in coloana 0
        for (p = linie; ; ) {
            p = strstr(p, " ");
            if (p == NULL) break;
            p++;
            if (*p == '\n') break;
            *p = toupper(*p);
        }
        fprintf(fo, "%s", linie);
    }
    fclose(fo);
    fclose(fi);
    printf("Terminat threadul: %ld ...> %s\n", pthread_self(), (char*)numei);
}
int main(int argc, char* argv[]) {
    int i;
    for (i=1; argv[i]; i++) {
        pthread_create(&tid[i], NULL, ucap, (void*)argv[i]);
        //   pthread_create(&tid,   NULL,   ucap,   (void*)argv[i]);   //   Vezi
comentariul de la sfârşitul sursei
        printf("Creat threadul: %ld ...> %s\n", tid[i], argv[i]);
    }
    for (i=1; argv[i]; i++) pthread_join(tid[i], NULL);
    // for (i=1; argv[i]; i++) pthread_join(tid, NULL); // Vezi comentariul de
la sfârşitul sursei
    printf("Terminat toate threadurile\n");
}
```

Compilarea se face: `gcc -pthread capitalizari.c`, iar rularea `./a.out f1 f2 . . .`

In sursa de mai sus sunt trei linii comentariu, în care, în loc de `tid[i]` se foloseşte `tid`. Este vorba de a folosi o singură variabilă în loc de a folosi un tablou în care să se folosească câte o intrare pentru fiecare thread în parte. Din punct de vedere sintactic, folosirea unei singure variabile este perfect valabilă. Insă, din punct de vedere al funcţionării, dacă se foloseşte o singură variabilă, atunci funcţiile `pthread_join` vor aştepta numai după ultimul thread!

5.3. Capitalizarea cuvintelor dintr-o listă de fişiere text - soluţia go

```
//Se capitalizeaza multithreading continutul unor fisiere:./capitalizeWord f1 f2
...
package main
import (
    "bufio"
    "fmt"
    "io"
    "os"
    "strings"
    "bytes"
)
func ucap(numei string, finish chan<- struct{}) {
    print("    Capitalizare: "+numei+"\n")
    f,_ := os.Open(numei)
    fi := bufio.NewReader(f)
    fo,_ := os.Create(numei+".CAPIT")
    for {
        linie,err := fi.ReadString('\n')
        if err == io.EOF { break }
        r := 0
        b := byte(linie[0])
        u := string(bytes.ToUpper([]byte{b}))
        linie = u+linie[1:]
        for {
            i := strings.IndexByte(linie[r:], ' ')
            if i == -1 { break }
            i = r + i + 1
            r = i + 1
            if r >= len(linie) { break }
            b = byte(linie[i])
            u = string(bytes.ToUpper([]byte{b}))
            linie = linie[:i]+u+linie[(i+1):]
        }
        fmt.Fprint(fo, linie)
    }
    f.Close()
    fo.Close()
    print("    Terminat capitalizare "+numei+"\n")
    finish <- struct{}{}
}
func main() {
    finish := make(chan struct{})
    for i :=1; i < len(os.Args); i++ { go ucap(os.Args[i], finish) }
    for i :=1; i < len(os.Args); i++ { <- finish }
    print("Terminat toate capitalizarile\n");
}
```

5.4. Capitalizarea cuvintelor dintr-o listă de fişiere text - soluţia python

```
# Se capitalizeaza multithreading continutul unor fisiere:./capitalizeWord f1 f2
...
import sys
import os
import threading
def ucap(numei):
    print("    Capitalizare: "+numei+"\n")
    fi = open(numei, "r")
    fo = open(numei+".CAPIT", "w")
    while True:
        linie = fi.readline()
        if linie == "": break
        r = 0
        linie = linie[0].upper()+linie[1:]
```

```
        while True:
            i = linie[r:].find(" ")
            if i == -1: break
            i = r + i + 1
            r = i + 1
            if r >= len(linie): break
            linie = linie[:i]+linie[i].upper()+linie[(i+1):]
        fo.write(linie)
    fi.close()
    fo.close()
    print("     Terminat capitalizare "+numei+"\n")
def main():
    t = []
    for i in range(1, len(sys.argv)):
        t.append(threading.Thread(target=ucap, args=(sys.argv[i],)))
        t[i-1].start()
    for i in range(1, len(sys.argv)):
        t[i-1].join()
    print("Terminat toate capitalizarile\n")
main()
```

5.5. De ce sunt necesare variabilele mutex?

Să construim un program care are o variabilă globală **count** şi 1000 de threaduri. Fiecare thread incrementează de câte 1000 de ori variabila count. In acest scop să considerăm programul următor:

```
#include <pthread.h>
#include <stdio.h>
int count = 0;
pthread_t tid[1000];
void* inc(void* nume) {
    for (int i = 0; i < 1000; i++) {
        int temp = count; temp++; count = temp;
    }
}
int main(int argc, char* argv[]) {
    int i;
    for (i=0; i < 1000; i++)
        pthread_create(&tid[i], NULL, inc, NULL);
    for (i=0; i < 1000; i++) pthread_join(tid[i], NULL);
    printf("count=%d\n", count);
}
```

Evident, funcţia inc a threadului se poate scrie: **count += 1000**. In mod intenţionat am "prelungit" execuţia în funcţia threadului. Rezultatele unor execuţii repetate ale programului de mai sus sunt:

```
florin@ubuntu:~/pthreads$ gcc -pthread inc.c
florin@ubuntu:~/pthreads$ ./a.out
count=997000
florin@ubuntu:~/pthreads$ ./a.out
count=997000
florin@ubuntu:~/pthreads$ ./a.out
count=980000
florin@ubuntu:~/pthreads$ ./a.out
count=1000000
florin@ubuntu:~/pthreads$ ./a.out
count=991000
florin@ubuntu:~/pthreads$ ./a.out
count=996000
florin@ubuntu:~/pthreads$ ./a.out
count=980000
florin@ubuntu:~/pthreads$ ./a.out
```

```
count=1000000
florin@ubuntu:~/pthreads$ ./a.out
count=997000
florin@ubuntu:~/pthreads$ ./a.out
count=995000
florin@ubuntu:~/pthreads$
```

De ce acest comportament ciudat? Pentru că execuţia threadurilor se face în paralel, deci este posibil ca cele trei instrucţiuni ale corpului for din funcţia **inc** să se înterpătrundă, aşa încât două threaduri să memoreze succesiv valoarea lui **count** independent de reţinerea ei în variabila locală temp. Astfel, se pierde efectul unuia dintre threaduri.

Pentru a asigura derularea corectă, se impune utilizarea unui mutex:

Se declară variabila globală
`pthread_mutex_lock(&exclusiv);`

Corpul ciclului se va înlocui cu:

```
pthread_mutex_lock(&exclusiv);
int temp = count; temp++; count = temp;
pthread_mutex_unlock(&exclusiv);
```

5.6. Câte perechi de argumente au suma număr par?

La linia de comandă se dau **n** perechi de argumente despre care se presupune ca sunt numere întregi si pozitive. Se cere numărul de perechi care au suma un număr par, numărul de perechi ce au suma număr impar si numărul de perechi în care cel putin unul dintre argumente nu este număr strict pozitiv.

Rezolvarea: Se va crea câte un thread pentru fiecare pereche. Trei variabile globale, cu rol de contoare, vor număra fiecare categorie de pereche.

Important: deoarece threadurile pot incrementa simultan unul dintre contoare, este necesar să se asigure accesul exclusiv la aceste contoare. De aceea, vom folosi o variabilă mutex care va proteja acest acces.

Intrebare: este corect sau nu să fie protejat fiecare contor printr-o variabilă mutex proprie? In exemplul nostru le protejăm simultan pe toate trei cu acelaşi mutex.

Sursa programului este:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define MAXLINIE 1000
typedef struct {char*n1; char*n2;} PERECHE;
pthread_t tid[100];
PERECHE pereche[100];
// PERECHE pereche; // Vezi comentariul de la sfârşitul sursei
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
int pare = 0, impare = 0, nenum = 0;
void* tippereche(void* pereche) {
    int n1 = atoi(((PERECHE*)pereche)->n1);
    int n2 = atoi(((PERECHE*)pereche)->n2);
    if (n1 == 0 || n2 == 0) {
        pthread_mutex_lock(&mut);
        nenum++;
        pthread_mutex_unlock(&mut);
    } else if ((n1 + n2) % 2 == 0) {
        pthread_mutex_lock(&mut);
```

```
            pare++;
            pthread_mutex_unlock(&mut);
        } else {
            pthread_mutex_lock(&mut);
            impare++;
            pthread_mutex_unlock(&mut);
        }
    }
    int main(int argc, char* argv[]) {
        int i, p, n = (argc-1)/2;
        for (i = 1, p = 0; p < n; i += 2, p++) {
            pereche[p].n1 = argv[i];
            pereche[p].n2 = argv[i+1];
            pthread_create(&tid[p], NULL, tippereche, (void*)&pereche[p]);
            // !! Apelul urmator nu este corect, Vezi explicatia la sfârşitul
    sursei
            // pthread_create(&tid[p], NULL, tippereche, (void*)&pereche);
        }
        for (i=0; i < n; i++) pthread_join(tid[i], NULL);
        printf("perechi=%d pare=%d impare=%d nenum=%d\n",n,pare,impare,nenum);
    }
```

In sursa de mai sus există un comentariu unde în loc de `pereche[p]` se foloseşte `pereche`. Este vorba de a folosi o aceaşi variabilă simplă în loc de a folosi un tablou în care să se folosească câte o intrare pentru fiecare thread în parte. Din punct de vedere sintactic, folosirea unei singure variabile este perfect valabilă. Insă, din punct de vedere al funcţionării, dacă se foloseşte o singură variabilă, se comite una dintre cele mai dese erori de logică în lucrul cu threaduri: Dacă se foloseşte o singură variabilă care transmite parametrul de intrare pentru toate apelurile `ucap` ale tuturor threadurilor, este posibil ca `(PERECHE*)pereche)->n1` sau `(PERECHE*)pereche)->n2` să nu preia intrările pregătite, ci să preia valori pregătite pentru threadul următor!

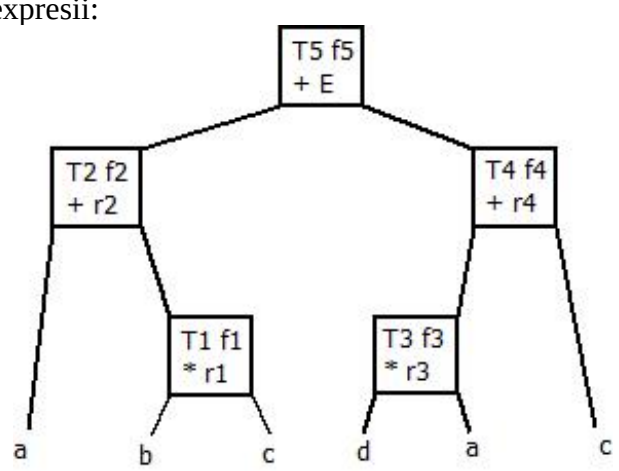
5.7. Evaluarea expresie aritmetică operator / thread şi paralelizare maximă

Se dă expresia aritmetică

`E = ( a + b * c ) + ( d * a + c )`

unde `a, b c, d` sunt numere întregi. Se cere evaluarea acestei expresii executând fiecare operaţie într-un thread separat şi cu lansarea a câtor mai multe threaduri în acelaşi timp.

Pentru rezolvare, vom folosi cinci threaduri `T1, T2, T3, T4, T5`, variabilele intermediare `r1, r3, r3, r4, E` şi funcţiile de thread `f1, f2, f3, f4, f5`. Arborele din figura următoare ilustrează evaluarea acestei expresii:



In cele cinci pătrate sunt ilustrate cele cinci threaduri care contribuie la rezolvarea problemei. Pentru fiecare thread se indică numele threadului, funcţia care îi determină funcţionarea, operaţia executată şi variabila unde se depune rezultatul operaţiei.

Sursa programului este dată mai jos:

```
# include <stdio.h>
# include <pthread.h>
pthread_t t[6];
int a=1, b=2, c=3, d=4, r1, r2, r3, r4, E;
void * f1 (void * x) {r1 = b * c;}
void * f2 (void * x) {r2 = a + r1;}
void * f3 (void * x) {r3 = d * a;}
void * f4 (void * x) {r4 = r3 + c;}
void * f5 (void * x) {E = r2 + r4;}
int main() {
    /* O prima varianta de creare / join
    pthread_create(&t[1], NULL, f1, NULL);
    pthread_create(&t[2], NULL, f2, NULL);
    pthread_create(&t[3], NULL, f3, NULL);
    pthread_create(&t[4], NULL, f4, NULL);
    pthread_create(&t[5], NULL, f5, NULL);
    pthread_join(t[1], NULL);
    pthread_join(t[2], NULL);
    pthread_join(t[3], NULL);
    pthread_join(t[4], NULL);
    pthread_join(t[5], NULL);
    Sfarsit varianta 1 - rezultat E=0! */

    // Varianta corecta:
    // T2 lansat dupa ce se termina T1 si T4 lansat dupa ce se termina T3
    // T5 lansat numai dupa ce se termina T2 si T4
    pthread_create(&t[1], NULL, f1, NULL);
    pthread_create(&t[3], NULL, f3, NULL);
    pthread_join(t[1], NULL);
    pthread_join(t[3], NULL);
    pthread_create(&t[2], NULL, f2, NULL);
    pthread_create(&t[4], NULL, f4, NULL);
    pthread_join(t[2], NULL);
    pthread_join(t[4], NULL);
    pthread_create(&t[5], NULL, f5, NULL);
    pthread_join(t[5], NULL);

    printf("E = %d\n", E);
}
```

Dacă se rulează prima variantă, în care nici un thread nu aşteaptă după altul, rezultatul este 0! Asta pentru faptul că threadurile care încarcă variabilele `r1 - r4` nu apucă să o facă şi threadul `T5` preia din ele valorile 0!

Varianta corectă este aceea în care `T2` şi `T4` aşteaptă să se termine mai întâi `T1` şi `T2`, apoi `T5` aşteaptă să se termine mai întâi `T2` şi `T4`.

5.8. De ce sunt necesare variabilele condiţionale?

In exemplul de mai sus unele threaduri au fost nevoite să aştepte terminarea altora pentru a îşi termina activitatea lor. Ce este de făcut dacă NU poate trece de un punct (să zicem **A**) al execuţiei până când un alt thread nu ajunge cu execuţia într-un punct (să zicem **B**). (In literatură fenomenul se numeşte

rendezvouz). Să schematizăm acest fenomen prin următorul program cu două threaduri, în care evenimentul de rendezvouz este variabila considerăm un exemplu:

```
#include <pthread.h>
#include <stdio.h>
int eveniment = 0;
// - - -
pthread_t tid[2];
void* produceeveniment(void* nume) {
    // - - - Calcule - - -
    // B
    printf("B va produce evenimentul\n");
    eveniment = 1;
    printf("B a produs evenimentul\n");
    // - - - alte calcule - - -
}
void* asteaptaeveniment(void* nume) {
    // - - - Calcule - - -
    // A
    printf("A asteapta evenimentul\n");
    while (eveniment == 0) {
        ;
    }
    printf("A a primit evenimentul\n");
    // - - - alte calcule - - -
}
int main(int argc, char* argv[]) {
    pthread_create(&tid[1], NULL, asteaptaeveniment, NULL);
    pthread_create(&tid[0], NULL, produceeveniment, NULL);
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
}
```

In funcţie de ordinea de creare a celor două threaduri, efectul execuţiei va fi:

A asteapta evenimentul	B va produce evenimentul
B va produce evenimentul	B a produs evenimentul
B a produs evenimentul	A asteapta evenimentul
A a primit evenimentul	A a primit evenimentul

Marea problemă este aşteptarea evenimentului: **`while (eveniment == 0){ ; };`**
In modul de mai sus avem de-a face cu o **aşteptare activă**, threadul care aşteaptă ocupă procesorul cu execuţia lui **while**! Este de dorit ca până la apariţia evenimentului, threadul care aşteaptă **să nu ocupe procesorul**! (Din păcate, soluţia cu semnale, pe care am descris-o la procese NU MERGE, deoarece threadurile producător şi de aşteptare sunt în acelaşi proces.).

Rezolvarea se face printr-o variabilă condiţională. Pentru aceasta se declară o variabilă condiţională şi o variabilă mutex asociată ei:

```
pthread_cond_t variabila = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutexasociat = PTHREAD_MUTEX_INITIALIZER;
```

Secvenţa de provocare a evenimentului se înlocuieşte cu una din variantele (a doua variantă sincronizează şi tipărirea celor două mesaje):

`pthread_mutex_lock(&mutexasociat);`
`eveniment = 1;`
`pthread_cond_signal(&variabila);`
`pthread_mutex_unlock(&mutexasociat);`
	`pthread_mutex_lock(&mutexasociat);`
	`printf("B va produce evenimentul\n");`
	`eveniment = 1;`
	`pthread_cond_signal(&variabila);`

	printf("B a produs evenimentul\n");
	pthread_mutex_unlock(&mutexasociat);

Secvenţa de cedare a procesorului pe durata aşteptării evenimentului se înlocuieşte cu una din variantele (a doua variantă sincronizează şi tipărirea celor două mesaje):

pthread_mutex_lock(&mutexasociat);
while (eveniment == 0) {
pthread_cond_wait(&variabila,
 	
&mutexasociat);
}
pthread_cond_signal(&variabila);
pthread_mutex_unlock(&mutexasociat);
 	
	pthread_mutex_lock(&mutexasociat);
	printf("A asteapta evenimentul\n");
	while (eveniment == 0) {
	pthread_cond_wait(&variabila,
	
	&mutexasociat);
	}
	pthread_cond_signal(&variabila);
	printf("A a primit evenimentul\n");
	pthread_mutex_unlock(&mutexasociat);

In funcţie de ordinea de creare a celor două threaduri, efectul execuţiei va fi:

B va produce evenimentul B a produs evenimentul A asteapta evenimentul A a primit evenimentul	A asteapta evenimentul B va produce evenimentul B a produs evenimentul A a primit evenimentul

5.9. Bariera - exemplu

Pthread_barrier este o construcție care permite ca mai multe threaduri independente să „aștepte” în spatele unei „bariere” pana când toate cele care așteaptă ajung la barieră. Bariera este opțională în standardul POSIX, așa că unele sisteme de operare nu o includ. Iată un exemplu simplu de utilizare a unei bariere:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
#define THREAD_COUNT 4
pthread_barrier_t mybarrier;
void* threadFn(void *id_ptr) {
    int thread_id = *(int*)id_ptr;
    int wait_sec = 1 + rand() % 5;
    printf("thread %d: Wait for %d seconds.\n", thread_id, wait_sec);
    sleep(wait_sec);
    printf("thread %d: I'm ready...\n", thread_id);
    pthread_barrier_wait(&mybarrier);
    printf("thread %d: going!\n", thread_id);
    return NULL;
}
int main() {
    int i;
    pthread_t ids[THREAD_COUNT];
```

```
    int short_ids[THREAD_COUNT];
    srand(time(NULL));
    pthread_barrier_init(&mybarrier, NULL, THREAD_COUNT + 1);
    for (i=0; i < THREAD_COUNT; i++) {
        short_ids[i] = i;
        pthread_create(&ids[i], NULL, threadFn, &short_ids[i]);
    }
    printf("main() is ready.\n");
    pthread_barrier_wait(&mybarrier);
    printf("main() is going!\n");
    for (i=0; i < THREAD_COUNT; i++) pthread_join(ids[i], NULL);
    pthread_barrier_destroy(&mybarrier);
    return 0;
}
```

5.10. Adunarea în paralel a n numere

Vom da, ca exemplu de utilizare a thread-urilor, evaluarea în paralel a sumei mai multor numere întregi. Evident, operaţia de adunare a *n* numere, chiar dacă *n* este relativ mare, nu impune cu necesitate însumarea lor în paralel. O facem totuşi pentru că reprezintă un exemplu elocvent de calcul paralel, în care esenţa este reprezentată de organizarea prelucrării paralele, aceeaşi şi pentru calcule mult mai complicate.

Presupunem că se dă un număr natural *n* şi un vector *a* având componentele întregi *a[0], a[1], . . . a[n-1]*. Ne propunem să calculăm, folosind cât mai multe thread-uri, deci un paralelism cât mai consistent, suma acestor numere. Modelul de paralelism pe care ni-l propunem este ilustrat mai jos, pentru *m* = 8:

Mai întâi sunt calculate, în paralel, următoarele patru adunări:
a[0] = a[0] + a[1]; a[2] = a[2] + a[3]; a[4] = a[4] + a[5]; a[6] = a[6] + a[7];

După ce primele două adunări, respectiv ultimele două adunări s-au terminat, se mai execută în paralel încă două adunări:
a[0] = a[0] + a[2]; a[4] = a[4] + a[6];

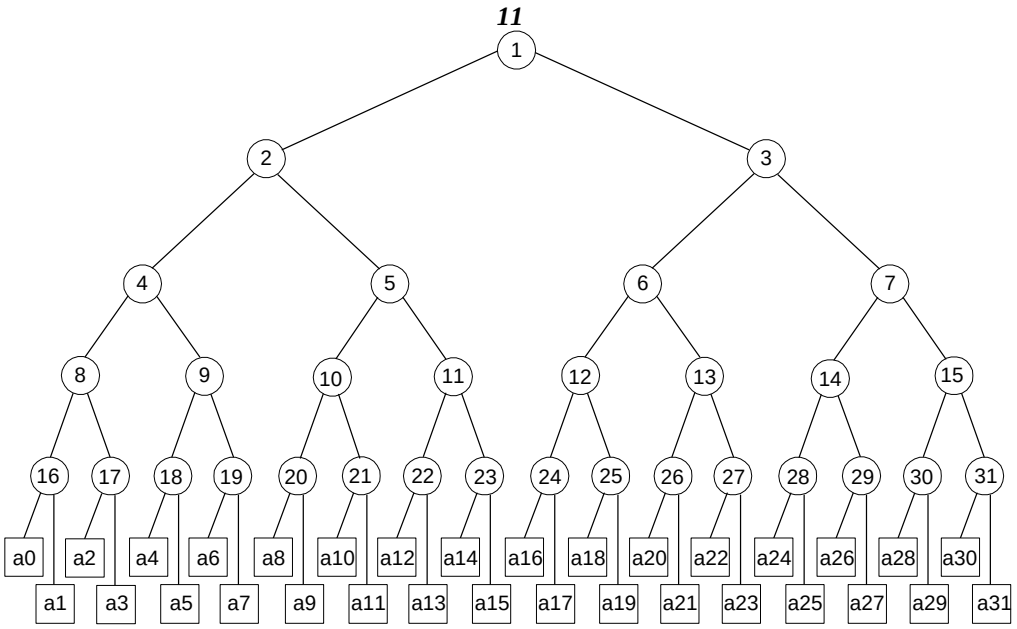
In sfârşit, la terminarea acestora, se va executa:
a[0] = a[0] + a[4];

Operaţiile de adunare se desfăşoară în paralel, având grijă ca fiecare adunare să se efectueze numai după ce operanzii au primit deja valori în adunările care trebuie să se desfăşoare înaintea celei curente. Este deci necesară o operaţie de *sincronizare* între iteraţii. Considerând că fiecare operaţie de adunare se execută într-o unitate de timp, din cauza paralelismului s-au consumat doar 3 unităţi de timp în loc de 7 unităţi de timp care s-ar fi consumat în abordarea secvenţială. In calcule s-au folosit 7 thread-uri, din care maximum 4 s-au executat în paralel.

Să considerăm acum problema pentru *n* numere şi să implementăm soluţia, cu intenţia de a folosi un număr maxim de thread-uri. Mai întâi extindem setul de numere până la *m* elemente, unde *m* este cea mai mică putere *l* a lui 2 mai mare sau egală cu *n*, adică: $2^{l-1} < n <= 2^l = m$, unde *l = partea întreagă superioară a lui* $log_2 n$. Elementele *a[n], . . ., a[m-1]* vor primi valoarea 0. Determinarea valorii lui **m** (cea mai mică putere a lui 2 mai mare sau egală cu **n**) se face uşor prin:

```
for (m = 1; n > m; m *= 2);
```

Pentru organizarea calculelor în regim multithreading, este convenabil să adoptăm o schemă arborescentă de numerotare a thread-urilor, ilustrată în figura de mai jos pentru 32 de numere.



Frunzele acestui arbore, reprezentate în pătrăţele, reprezintă **operanzii** de adunat. Nodurile interioare, reprezentate în cerculeţe, reprezintă **threadurile** care efectuează adunările.

Un thread oarecare **i** are doi fii. Threadurile de pe ultimul nivel interior au ca fii câte doi operanzi din tabloul de însumat. Celelalte le vom numi **threaduri interioare** şi au câte două **threaduri fii**, numerotate **2*i** şi **2*i+1**. Fiecare thread interior îşi va face propria operaţie de adunare numai după ce cei doi fii ai săi îşi vor termina adunările lor.

Fiecare thread **i** face o adunare de forma *a[s] = a[s] + a[d]*. In cele ce urmează vom determina indicii **s** şi **d** în funcţie de **i**. Vom numi **threaduri fraţi** threadurile ce se află pe acelaşi nivel, numerotaţi în ordinea crescătoare a vârstei lor. In cazul nostru, 2 şi 3 sunt fraţi cu 2 cel mai mic, 4, 5, 6 şi 7 sunt fraţi cu 4 cel mai mic, 8, 9, . . ., 15 sunt fraţi cu 8 cel mai mic, 16, 17, . . ., 31 sunt fraţi cu 16 cel mai mic ş.a.m.d. Fraţii cei mici de pe fiecare nivel au ca număr o putere a lui doi.

Este uşor de observat că fraţii de pe acelaşi nivel au acelaşi număr de operanzi: dacă **m** este numărul total de operanzi (putere a lui 2), **i** este numărul unui thread de pe un anumit nivel, iar **j** este numărul fratelui cel mic al acestuia, atunci fraţii de pe acest nivel au fiecare câte **m / j** operanzi. Indicele **s** al primului operand al threadului **i** este egal cu suma numărului de operanzi ai fraţilor lui mai mici, iar pentru **d** se mai adaugă jumătate din numărul de operanzi ai threadului, adică **d = s + m / j / 2**.

Determinarea numărului **j** al fratelui cel mic înseamnă găsirea celei mai mari puteri a lui 2 care este mai mică sau egală cu **i** şi ea se determină uşor prin secvenţa:

```
for (j = m; j > i; j /= 2);
```

Cu aceste precizări, sursa programului de adunare multithreading este:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int n, m; // n = numarul de operanzi; m = min {2^k >= n}
int* a; // valoarea 1 pentru pana la n-1,  0 de la n la m-1
pthread_t *tid; // id-urile threadurilor; -1 thread nepornit
pthread_mutex_t print = PTHREAD_MUTEX_INITIALIZER; // Printare exclusiva

// Rutina thread-ului nr i de adunare
void* aduna(void* pi) {
    int i, j, sa, da, st = 0, dr = 0, k;
    i = *(int*)pi; // Retine numarul threadului
    if (i < m / 2) {
```

```
            st = 2 * i; // Retine fiul stang
            dr = st + 1; // Retine fiul drept
            while (tid[st] == -1);     // Asteapta sa inceapa fiul stang
            // while (tid[st] == -1) sleep(1); // poate asa!
            // Cel mai sanatos este să se utilizeze un set de variabile coditionale
            // care sa semnaleze pornirile threadurilor.
            while (tid[dr] == -1);     // Asteapta sa inceapa fiul drept
            // while (tid[dr] == -1) sleep(1); // poate asa!
            pthread_join(tid[st], NULL); // Asteapta sa se termine fiul stang
            pthread_join(tid[dr], NULL); // Asteapta sa se termine fiul drept
        }
        for (j = m; j > i; j /= 2); // Determina fratele cel mic
        for (k = j, sa = 0; k < i; k++) sa += m / j; // operand stang
        da = sa + m / j / 2; // operand drept
        a[sa] += a[da]; // Face adunarea proppriu-zisa
        pthread_mutex_lock(&print);// Asigura printare exclusiva
        printf("Thread %d: a[%d] += a[%d]", i, sa, da);
        if (st > 0) printf(" (dupa fii %d %d)\n", st, dr); else printf("\n");
        pthread_mutex_unlock(&print);
    }

    // Functia main, in care se creeaza si lanseaza thread-urile
    int main(int argc, char* argv[]) {
        n = atoi(argv[1]); // Numarul de numere de adunat
        for (m = 1; n > m; m *= 2); // m = min {2^k >= n}
        int* pi;
        int i;
        a = (int*) malloc(m*sizeof(int)); // Spatiu pentru intregii de adunat
        pi = (int*) malloc(m*sizeof(int)); // Spatiu pentru indicii threadurilor
        tid = (pthread_t*) malloc(m*sizeof(pthread_t)); // id-threads
        for (i = 0; i < n; i++) a[i] = 1; // Aduna numarul 1 de n ori
        for (i = n; i < m; i++) a[i] = 0; // Completeaza cu 0 pana la m
        for (i = 1; i < m; i++) tid[i] =-1; // Threadurile sunt inca nepornite
        for (i = 1; i < m; i++) pi[i] = i; // Threadurile sunt inca nepornite
        for (i = 1; i < m; i++)
            // De ce folosim mai jos &pi[i] in loc de &i? vezi un exemplu
    precedent!
            pthread_create(&tid[i], NULL, aduna, (void*)(&pi[i])); // Threadul i
        pthread_join(tid[1], NULL); // Asteapta dupa primul thread
        printf("Terminat adunarile pentru n = %d. Total: %d\n", n, a[0]);
        free(a); // Eliberaza tabloul de numere
        free(pi); // Elibereaza tabloul de indici de threaduri
        free(tid); // Elibereaza tabloul de id-uri de threaduri
    }
```

In legătură cu aşteptările active **`while (tid[st] == -1); while (tid[dr] == -1);`**
Am folosit acest mod de aşteptare pentru simplitatea codului; mai sunt şi altele posibile:

- Să se pornească threadurile în maniera "bottom up" în arborele asociat.
- variantă "mai puţin ortodoxă" este de a ceda controlat procesorul pentru un interval de timp: **`while (tid[st] == -1) sleep(1);`**
- Fiecare thread să îşî pornească singur sei doi fii pe care (eventual) îi are.
- Să se folosească un tablou de variabile condiţionale care să semnaleze / să aştepte pornirile threadurilor

OPERATING SYSTEMS
– Seminar 7 –

FILE SYSTEMS IN LINUX

1. BASIC CONCEPTS

- a file system controls how data is stored on and retrieved from a storage medium
- file system concepts: *partition*, *file, directory, path*
- storage mediums: magnetic tape, floppy disk, hard disk, optical disk, SSD (Solid State Drive)
- storing data on disks: *track, sector, disk block*
- file systems:
  - for disks: *FAT (FAT12/FAT16/FAT32)*, *NTFS*, *exFAT*, *HFS*, *HPFS*, *APFS*, *ext2fs*, *ext3fs*, *ext4fs*, *ReiserFS*, *XFS*, *ZFS* etc.
  - for optical disk: *ISO 9660*, *UDF (Universal Disk Format)*

2. JOURNALING FILE SYSTEMS

- issues in classic file systems:
  - the consistency of file system is affected by failures (i.e. system crashes, software crashes, improper shutdowns, power outages)
  - following a crash the *fsck (file system consistency check)* utility has to be run
  - *fsck* will scan the entire file system validating all entries and making sure that blocks are allocated and referenced correctly; if it finds a corrupt entry it will attempt to fix the problem
- solution: fault-resilient file systems
- the first fault-resilient file system was the IBM Journaled File System (JFS)
- it was first released in 1990 by IBM Corp. for its AIX operating system
- a journaling file system = *a file system that maintains a special file called a "journal" that is used to repair any inconsistencies that occur as the result of a failure*
- a journaling file system records changes to the file system; in this way it is able to recover after a failure with minimal loss of data
- a journaling file system write metadata (i.e. data about files and directories) into the journal that is flushed to the disk before each command returns
- examples:
  - Windows: *NTFS* (New Technology File System)
  - Unix/Linux: *ext3fs* (third extended file system), *ext4fs, ReiserFS*, *JFS*/*JFS2*, *XFS etc.*
- the most common journaling strategies:

– writeback mode - only the metadata is journaled, and the data blocks are written directly to their location on the disk

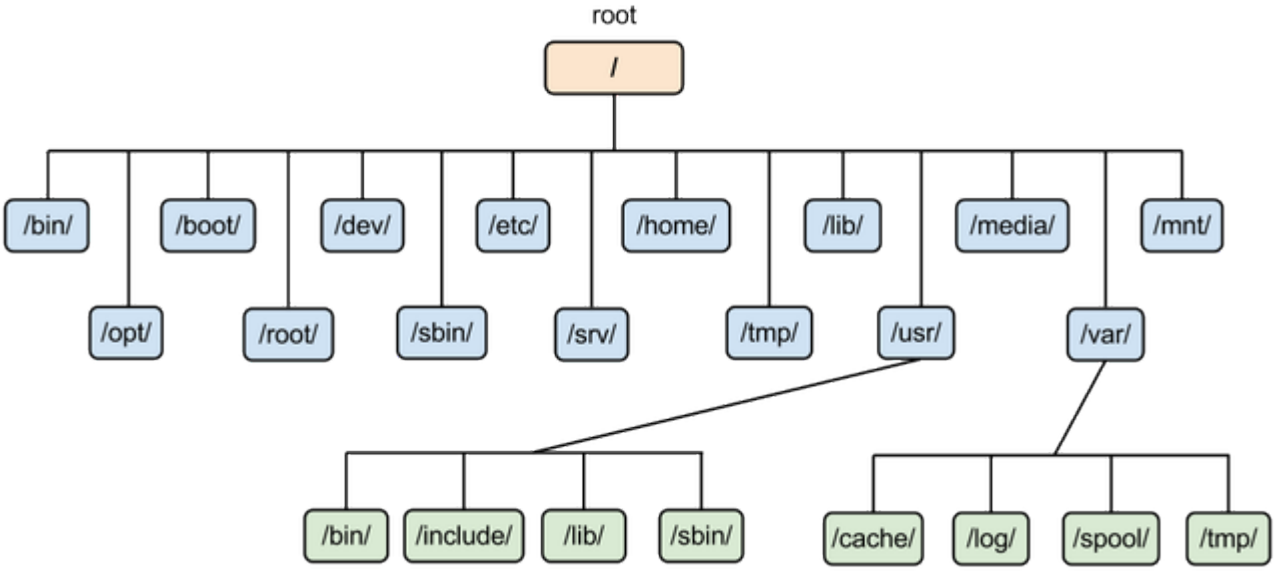
– ordered mode - metadata journaling only, but writes the data before journaling the metadata

– data mode - both metadata and data are journaled

- advantage: the structure of the file system is always internally consistent
- drawbacks: journaling does not give a performance boost; in fact, the journaling operation reduces the speed slightly, in exchange for reliability

3. FILE SYSTEMS IN LINUX

- Linux supports almost 100 types of file systems
- in Linux and many other operating systems, directories can be structured in a tree-like hierarchy; the Linux directory structure is well defined and documented in the *Linux Filesystem Hierarchy Standard (FHS)*



- mount a file system:

  – refers back to the early days of computing when a tape or removable disk pack would need to be physically mounted on an appropriate drive device;

  – after being physically placed on the drive, the filesystem on the disk pack would be logically mounted by the operating system to make the contents available for access by the OS, application programs and users

- *mounting point*, `mount/umount` commands

REFERENCES:

- Storing and organizing data files on disks

  http://ntfs.com/hard-disk-basics.htm

  http://www.mathcs.emory.edu/~cheung/Courses/377/Syllabus/1-files/intro-disk.html

- Anatomy of Linux journaling file systems
  https://www.ibm.com/developerworks/library/l-journaling-filesystems/index.html
- An introduction to Linux filesystems
  https://opensource.com/life/16/10/introduction-linux-filesystems
- Linux Filesystem Hierarchy Standard (FHS)
  http://www.pathname.com/fhs/
- How to Mount and Unmount File Systems in Linux
  https://linuxize.com/post/how-to-mount-and-unmount-file-systems-in-linux/
- ZFS (Zettabyte File System)
  https://arstechnica.com/information-technology/2020/05/zfs-101-understanding-zfs-storage-and-performance/