Fundamentos de Sistemas de Operação

MIEI 2018/2019

Laboratorial Session 2

Objectives

System calls *fork( )* and *wait( )*. Implementation of a simple shell.

Exercise some system calls related to process management

In Unix a process can create a new process, that's a copy of itself, using *fork* system call. The new process is usually called a child of the original process, the father. A process can block, waiting for the termination of any child, using *wait* system call. Try the following example of a program that forks a new process and demonstrates that some parts of your program are executed by two different process. Check also the examples in chapter 5 of your book and the manual pages (volume 2) of the system calls *fork( )* and *wait( )*.

```
int main() {
    printf("I'm process %d\n", getpid());

    switch ( fork() ) {
      case -1: perror("fork"); exit(1);     // error
      case 0: printf("I'm child %d\n", getpid());
              break;
      default: printf("I'm father %d\n", getpid());
               wait(NULL);           // wait for child exit
    }
    printf("Bye!\n");
    return 0;
}
```

Justify the produced output.

As seen in last week, a system call takes more time than a regular function call. Each system call can take more or less time depending on the actions the kernel must complete internally before returning a reply to the processes making the call. Evaluate the time to create and terminate a new process using `forktiming.c`.

Compare the times with a simple system call like *getuid( )* from Lab01.

Command interpreter or Shell

A *command line interpreter (CLI)* or *Shell* is a program that reads commands from a terminal (or a file) and executes them. There are two types of commands: "*external" commands*, which require an executable file somewhere on the file system to be executed on separated processes (e.g. *ls*, *gcc*, or any text editor); and *"internal" commands*, which are executed by the Shell itself (e.g. *cd* or *exit*). Additionally, some Shells interpret a programming language so that you can write small programs designated as *shell scripts*.

Over the years, several different Shells have been developed for the Unix system like the *sh* (*Bourne Shell*), *csh* (*C-Shell*), etc. In the Linux system, the most commonly used Shell is the *bash* (*Bourne Again Shell*). In general, all Shells present the same operation: an activity cycle comprising the reading of a command line, its processing and its execution. Each command line specifies one (or more) commands to be executed along with its (their) operands (either mandatory or optional). A Shell's operation may be summarized by the following code:

```
printf("> "); fflush(stdout);  //writes the prompt on the standard output
while ( fgets( line, LINESIZE, stdin ) != NULL ) {
      if ( makeargv( line, av)>0 ) runcommand( av );
      printf("> "); fflush(stdout);
}
```

Upon reading a command line from the user, this code invokes the function *makeargv* to build the array named *av* so that it points to all words present in the string *line*, similarly to the variable *argv[]* of the main function. As a result, *makeargv* returns the number of words in *av[]*. Subsequently, the function *runcommand* executes the ("internal" or

"external") command(s) specified in that array *av*[]. One possible implementation of the function *makeargv* is shown in the Annex.

Execution of Internal Commands

Implement the function *runcommand(char *argv[])*, assuming that receives as argument a vector with just one string that can be the **internal command "exit"**. Use library function *strcmp* to compare the command (first word) with the "exit" word and, if equal, terminate the program (exit).

Introduction to External Commands

Extend the function *runcommand(char *argv[])* so that, if no internal command is recognized (no "exit"), assumes that *argv* vector defines **one external command and its arguments**, and executes this command in a new child process. For now, the function just uses the *fork* system call to create a new process that should print to standard output all words in *argv*. The father process waits for its child process to terminate by using the *wait* system call before returning to the Shell's main execution cycle.

As an example, consider the command for listing the contents of the current directory, in its "long" version:

```
ls -l
```

In this case, the array built by *makeargv* will contain "**ls**", "**-l**" and **NULL**. Subsequently, the function *runcommand* creates a process that prints those words. Example:

```
switch ( fork() ) {
    case -1: perror("fork"); exit(1);

    case 0: printf("should execute: ");
            for( int i=0; argv[i]!=NULL; i++)
                printf("%s\n", argv[i]);
            exit(0);

    default: wait(NULL); // wait for child exit
}
```

Will produce the following output:

```
should execute:
ls
-l
```

Bibliography

- Chapter 5 and appendix F (Lab Tutorial) of recommended book http://www.ostep.org/
- On-line manual pages for LibC and system call functions: *fork*, *wait, exit* and *strcmp.*

Annex

```
int makeargv(char *s, char *argv[]) {
  // in: s points a text string with words
  // pre: argv is predefined as char *argv[ARGVMAX]
  // out: argv[] points to all words in the string s (*s is modified!)
  // return: number of words pointed to by the elements in argv (or -1 in case of error)
  int ntokens;

  if ( s==NULL || argv==NULL || ARGVMAX==0)
      return -1;

  ntokens = 0;
  argv[ntokens]=strtok(s, " \t\n");
  while ( (argv[ntokens]!= NULL) && (ntokens<ARGVMAX) ) {
      ntokens++;
      argv[ntokens]=strtok(NULL, " \t\n");  // breaks 's' inline at separators
  }
  argv[ntokens] = NULL;  // terminate with NULL reference
  return ntokens;
}
```