Operating Systems

Grado en Informática. Course 2019-2020

Lab Assignment 3

CONTINUE the coding of the shell started in the previous lab assigment. In this lab assignment we'll add to the shell the capability to execute external programs both in foreground and background and without creating process (replacing the shell code). The shell will keep track (using a list) of the processes created to execute programs in background.

priority [**pid**] [**value**]. If both arguments (*pid and value*) are specified, the priority of process *pid* is changed to *value*. If only *pid* is specified, the shell will show the priority of process *pid*.

fork The shell creates a child process with *fork* (this child process executes the same code as the shell) and waits (with one of the *wait* system calls) for it to end.

exec prog arg1 arg2 . . . Executes, without creating a process (**REPLACING the shell's code**) the program *prog* with its arguments. *prog* is a filename that represents an external program and arg1, arg2 . . . represent the program's command line arguments (they can be more than two).

exec @pri prog arg1 arg2. . . Does the same as the previous *exec* command, but before executing *prog* it changes the priority of the proccess to *pri*

pplano prog arg1 arg2. . . The shell creates a process that executes in foreground (waits for it to exit) the program *prog* with its arguments. *prog* is a filename that represents an external program and arg1, arg2 . . . represent the program's command line arguments (they can be more than two).

pplano @pri prog arg1 arg2. . . Does the same as the previous command, but before executing *prog* it changes the priority of the proccess that executes prog to *pri*

splano prog arg1 arg2. . . The shell creates a process tha executes in background the program *prog* with its arguments. *prog* is a filename that represents an external program and arg1, arg2 . . . represent the program's command line arguments (they can be more than two). The process that executes prog is added to the list the shell keeps of the background processes. The command *listarprocs* shows this list.

splano @pri prog arg1 arg2. . . Does the same as the previous command, but before executing *prog* it changes the priority of the proccess that executes prog to *pri*. The process that executes prog is added to the

list the shell keps of the backgroud processes. The command *listarprocs* shows this list.

- The following three items describe what the shell should do if we type as input something that is not one of its predefined *"commands"*. The behaviour is exactly the same as the *pplano* command. When supplying an **&** as the last arg to a program, the execution must be in background: exactly as the *splano* command but without passing the & to the program being executed.

  prog arg1 arg2... The shell creates a process tha executes in foreground the program *prog* with its arguments. *prog* is a filename that represents an external program and arg1, arg2 ... represent the program's command line arguments (there can be more than two). THIS IS EXACTLY THE SAME as doing *pplano prog arg1 arg2...*

  prog arg1 arg2... & The shell creates a process tha executes in background the program *prog* with its arguments. *prog* is a filename that represents an external program and arg1, arg2 ... represent the program's command line arguments (there can be more than two). The process that executes prog is added to the list the shell keeps of the background processes. The command *listarprocs* shows this list. THIS IS EXACTLY THE SAME as doing *splano prog arg1 arg2...*

  @pri prog arg1 arg2... [**&**] Does the same as the previous commands, but before executing *prog* it changes the priority of the proccess that executes prog to *pri*. Execution will be in foreground or background depending on the last argument being &, so *@pri prog arg1 arg2 ...* is the same as *pplano @pri prog arg1 arg2 ...*, and *@pri prog arg1 arg2 ... &* is the same as *splano @pri prog arg1 arg2 ...*

  Examples

```
#) pplano ls -l /usr
#) plano @15 du -a /
#) ls -lisa /home
#) @12 du -a /usr
#) splano xterm -e bash
#) splano @10 xterm -bg yellow
#) xclock &
#) @12 xclock -update 1
```

listarprocs Shows the list of background processes of the shell. For each process it must show (IN A SINGLE LINE):

- The process pid
- The process priority
- The command line the process is executing (executable and arguments)
- The time it started
- The process state (Running, Stopped, Terminated Normally or Terminated By Signal).
- For processes that have terminated normally the value returned, for processes stopped or terminated by a signal, the name of the signal.

This command **USES THE LIST OF BACKGROUND PROCESSES of the shell, it DOES NOT HAVE TO GO THROUGH THE /proc FILESYSTEM**

proc [-fg] id Shows information on process *pid* (provided *pid* represents a background process from the shell). If *pid* is not given or if *pid* is not a background process from the shell, this comand does exactly the same as the comand *listarprocs*. If we supply the argument *-fg* process with pid *pid* is brought to the foreground (the shell waits for it to end), and once the program has ended the shell will inform of how it has ended and remove it form the list

borrarprocs -term Removes from the list the processes that have exited normally.

borrarprocs -sig Removes from the list the processes that have been terminated by a signal.

Information on the system calls and library functions needed to code this program is available through `man`: (*setpriority, getpriority, fork, exec, waitpid . . .).*

- Work must be done in pairs.
- The source code will be submitted to the subversion repository under a directory named **P3**
- A `Makefile` must be supplied so that the program can be compiled with just `make`. The executable produced must be named shell
- Only one of the members of the workgroup will submit the source code. The names and logins of all the members of the group should be in the source code of the main program (at the top of the file)
- For the list of background processes implementation:
  - groups that used one of the array implementations (array or array of pointers) for the previous lab assignments, must now use one

of the linked list implementations (either with or without header node).

- groups that used one of the linked list implementations (with or without header nodearray or array of pinters) for the previous lab assignments, must now use one of the array implementations (array or array of pointers).

DEADLINE: December Friday 13, 2019, 23:00

Assesment: During lab hours

CLUES

The difference between executing in foreground and background is that in foreground the parent process waits for the child process to end using one of the *wait* system calls, whereas in background the parent process continues to execute concurrently with the child process.

Executing in background should not be tried with programs that read from the standard input in the same session. `xterm` and `xclock` are good candidates to try background execution.

To create processes we use the *fork()* system call. *fork()* creates a processes that is a clone of the calling process, the only difference is the value returned by *fork* (0 to the child process and the child's pid to the parent process).

The *waitpid* system call allows a process to wait for a child process to end.

The following code creates a child process that executes funcion2 while the parent executes funcion1. When the child has ended, the parent process executes funcion3

```
.......
if ((pid=fork())==0) {
    funcion2();
    exit(0);
    }
else {
   funcion1();
   waitpid(pid,NULL,0);
   funcion3();
   }
```

As *exit()* ends a program, we could rewrite it like this (without the *else*

```
.......
```

```
if ((pid=fork())==0) {
    funcion2();
    exit(0);
    }
funcion1();
waitpid(pid,NULL,0);
funcion3();
```

In this code both the parent process and the child process execute *funcion3()*

```
.......
if ((pid=fork())==0)
    funcion2();
else
    funcion1();
funcion3();
```

For a process to execute a program WE MUST USE the *execvp()* system call. *execvp* searches the executables in the directories specified in the PATH environment variable. *execvp()* only returns a value in case of error, otherwise it replaces the calling process's code. Here you have an example using *execl*.

```
 ......
 execl("/bin/ls","ls","-l","/usr",NULL);
 funcion(); /*no se ejecuta a no ser que execl falle*/
```

execvp operates the exactly the same but with two small differences

- it searches for executables in the PATH so, instead of specifying `''/bin/ls‘‘` it would suffice to pass just `''ls‘‘`
- we pass a NULL terminated array of pointers, instead of a variable number of pointers to the arguments

To check a process state we can use *waitpid()* with the following flags.

`waitpid(pid, &estado, WNOHANG |WUNTRACED |WCONTINUED)` will give us information about the state of process *pid* in the variable *estado* **ONLY WHEN THE RETURNED VALUE IS pid**. Such information can be evaluated with the macros descibed in `man waitpid` (WIFEXITED, WIFSIGNALED ...)

The following functions allow us to obtain the signal name from the signal number and viceversa. (in systems where we do not have *sig2str or str2sig*)

```
#include <signal.h>
/*****************************SENALES ****************************************/
struct SEN{
  char *nombre;
  int senal;
};
static struct SEN sigstrnum[]={
        "HUP", SIGHUP,
        "INT", SIGINT,
        "QUIT", SIGQUIT,
        "ILL", SIGILL,
        "TRAP", SIGTRAP,
        "ABRT", SIGABRT,
        "IOT", SIGIOT,
        "BUS", SIGBUS,
        "FPE", SIGFPE,
        "KILL", SIGKILL,
        "USR1", SIGUSR1,
        "SEGV", SIGSEGV,
        "USR2", SIGUSR2,
        "PIPE", SIGPIPE,
        "ALRM", SIGALRM,
        "TERM", SIGTERM,
        "CHLD", SIGCHLD,
        "CONT", SIGCONT,
        "STOP", SIGSTOP,
        "TSTP", SIGTSTP,
        "TTIN", SIGTTIN,
        "TTOU", SIGTTOU,
        "URG", SIGURG,
        "XCPU", SIGXCPU,
        "XFSZ", SIGXFSZ,
        "VTALRM", SIGVTALRM,
        "PROF", SIGPROF,
        "WINCH", SIGWINCH,
        "IO", SIGIO,
        "SYS", SIGSYS,
/*senales que no hay en todas partes*/
#ifdef SIGPOLL
        "POLL", SIGPOLL,
#endif
#ifdef SIGPWR
        "PWR", SIGPWR,
```

```
#endif
#ifdef SIGEMT
        "EMT", SIGEMT,
#endif
#ifdef SIGINFO
        "INFO", SIGINFO,
#endif
#ifdef SIGSTKFLT
        "STKFLT", SIGSTKFLT,
#endif
#ifdef SIGCLD
        "CLD", SIGCLD,
#endif
#ifdef SIGLOST
        "LOST", SIGLOST,
#endif
#ifdef SIGCANCEL
        "CANCEL", SIGCANCEL,
#endif
#ifdef SIGTHAW
        "THAW", SIGTHAW,
#endif
#ifdef SIGFREEZE
        "FREEZE", SIGFREEZE,
#endif
#ifdef SIGLWP
        "LWP", SIGLWP,
#endif
#ifdef SIGWAITING
        "WAITING", SIGWAITING,
#endif
        NULL,-1,
        };    /*fin array sigstrnum */

int Senal(char * sen)  /*devuel el numero de senial a partir del nombre*/
{
  int i;
  for (i=0; sigstrnum[i].nombre!=NULL; i++)
        if (!strcmp(sen, sigstrnum[i].nombre))
            return sigstrnum[i].senal;
  return -1;
}
```

```
char *NombreSenal(int sen)  /*devuelve el nombre senal a partir de la senal*/
{                        /* para sitios donde no hay sig2str*/
 int i;
  for (i=0; sigstrnum[i].nombre!=NULL; i++)
        if (sen==sigstrnum[i].senal)
                return sigstrnum[i].nombre;
 return ("SIGUNKNOWN");
}
```