Operating Systems

Grado en Informática. Course 2021-2022

Lab Assignment 3

CONTINUE the coding of the shell started in previous lab assigments. 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 *value* is not given shows the priority of process *pid*. If neither *pid* nor *value* is given specified, the priority of the process executing the shell is shown. If both arguments (*pid and value*) are specified, the priority of process *pid* will be changed to *value*.

rederr [**-reset**] **fich**

- **rederr file** Redirects the standard error of the shell to file *fich*. (this is a redirection done with the open and dup system calls. THIS IS NOT CHANGING THE CALLS TO perror IN THE CODE FOR CALLS TO fprintf)
- **rederr** Shows were the standard error is currently going to
- **rederr -reset** Restores the standard error to what it was originally

entorno [**-environ**] Shows the environment of the shell process. For each variable it shows the string, the address at which the string is stored, and the address at which the pointer to the string is stored. The functions shown in the *entorno* sample programs can be used

- **entorno** Shows all the environment variables of the shell process. Access will be through the third argument of main
- **entorno -environ** Shows all the environment variables of the shell process. Access will be through the external variable *environ*
- **entorno -addr** Shows the value (as pointers) of environ and the third argument of main. Shows also the addresses at with they are stored.

mostrarvar **VAR1** Shows the value of environment variable VAR. It shows the string, the address at which the string is stored, and the address at which the pointer to the string is stored accessing through *environ* and the third argument of main. It also shows the value (and the address) returned by the *getenv* library function. If no var is specified this is equivalent to the entorno command

cambiarvar **[-a|-e|-p] VAR VALUE**. Changes the value of the environment var VAR to value. *-a* means access through main's third argument. -*e* through *environ* and *-p* means access through the library function putenv. Note that if the variable does not previously exist, *putenv* will create it: THIS MUST NOT BE CHECKED FOR.

uid **-get|-set [-l] id**

- **uid -get** o simply **uid** Prints the real and effective user credentialas of the process running the shell (both the number and the associated login)
- **uid -set [-l] id** Establishes the efective user id of the shell process (see notes on uids). *id* represents the *uid* (numerical value). If -l is given *id* represents the login. If no arguments are given to *uid -set*, this command behaves exactly as *uid -get*. **PLEASE NOTE THAT THE *setuid() system call* WILL FAIL UNLESS THE SHELL IS A *setuid file* BEING EXECUTED BY OTHER THAN THE OWNER**

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.

ejec **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 (there can be more than two).

ejecpri **prio prog arg1 arg2. . .** Does the same as the previous *execute* command, but before executing *prog* it changes the priority of the proccess to *prio*

fg **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 (there can be more than two).

fgpri **prio 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 *prio*

back **prog arg1 arg2. . .** The shell creates a process that 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 *litsjobs* shows this list.

backpri **prio 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 *prio*. The process that executes prog is added to the list the shell keps of the backgroud processes. The command *listjobs* shows this list.

ejecas **login prog arg1 arg2. . .** Tries to execute as user *login* the program and arguments *prog arg1 arg2 . . .* . Execution is to be done without creating a process. If the change of user credential can not be achieved (either login does not exists or the process has not enough rights) nothing should get executed. **PLEASE NOTE THAT THE *ejecas, fgas, backas* WILL FAIL UNLESS THE SHELL IS A *setuid file* BEING EXECUTED BY OTHER THAN THE OWNER** (see notes and examples on uids)

fgas **login prog arg1 arg2. . .** Creates a process that tries to execute as user *login* the program and arguments *prog arg1 arg2 . . .* . If the change of user credential can not be achieved (either login does not exists or the process has not enough rights) nothing should get executed. (see notes and examples on uids)

bgas **login prog arg1 arg2. . .** Creates a process that tries to execute in the backgroud and as user *login* the program and arguments *prog arg1 arg2 . . .* . If the change of user credential can not be achieved (either login does not exists or the process has not enough rights) nothing should get executed. (see notes and examples on uids).The process that executes prog is added to the list the shell keeps of the background processes. The command *litsjobs* shows this list.

listjobs Shows the list of background processes of the shell. For each process it must show (IN A SINGLE LINE and not necessarily in that order):

– The process pid

– The process priority

– the user for which the process is running

– 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 PRO-**

CESSES of the shell, it DOES NOT HAVE TO GO THROUGH THE /proc FILESYSTEM

job [-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 *listjobs*. If we supply the argument *-fg* process with pid *pid* mus be brought to the foreground (the shell must wait for it to end with the *waitpid* system call), and once the program has ended the shell will inform of how it has ended and remove it form the list

borrarjobs -term|-sig|-all|-clear

- **borrarjobs -term** Removes from the list the processes that have exited normally.
- **borrarjobs -sig** Removes from the list the processes that have been terminated by a signal.
- **borrarjobs -all** Removes from the list all the processes that have finished (exited normally or by a signal)
- **borrarjobs -clear** Empties the list of background processes

******* The following two 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 *fg* command. When supplying an & as the last arg to a program, the execution must be in background: exactly as the *back* 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 *fg prog arg1 arg2...* For example, to execute `ls -l /home /usr` in the foreground we can do

```
 -> fg ls -l /home /usr
```

or

```
 -> ls -l /home /usr
```

but **WE MUST NOT DO**

```
 -> prog ls -l /home /usr
```

prog arg1 arg2... & The shell creates a process that 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 *listjobs* shows this list. THIS IS EXACTLY THE SAME as doing *back prog arg1 arg2...*

Examples

```
#) fg ls -l /usr
#) fgpri 10  du -a /
#) ls -lisa /home
#) du -a /usr
#) back xterm -e bash
#) backpri 12 xterm -bg yellow
#) back xclock -update 1
#) xclock -update 1 &
#) backas  manolo ls -lisa /home/manolo
.......
#) ejecas josefa du -hs /home/josefa/Desktop
.......
```

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

- Work must be done in pairs.
- The source code will be submitted using moodle in a **ZIP FILE**, *p3.zip* containing a directory **P3** where all the files reside
- 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)

DEADLINE: DECEMBER 10 , 2021, 23:00

ASSESMENT: DURING LAB HOURS **NOTES ON ENVIRONMENT**

Environment variables are strings in the form `''NAME=value``. We can access thes variables three ways

- through main's thrid argument (`char * env[]`). A NULL terminated array of pointers pointing to all the environment variables.
- using the external variable *environ* (`extern char ** environ`). Which,

again, is A NULL terminated array of pointers pointing to all the environment variables.

- through the library functions *getenv* and *putenv*

The following function shows all the environment. It receives the array of pointers and a name to specify when printing

```
void MostrarEntorno (char **entorno, char * nombre_entorno)
{
 int i=0;

 while (entorno[i]!=NULL) {
    printf ("%p->%s[%d]=(%p) %s\n", &entorno[i],
      nombre_entorno, i,entorno[i],entorno[i]);
    i++;
  }
}
```

To search (and/or) change variables in the environment directly (without using the library functions) we can use

```
int BuscarVariable (char * var, char *e[])
{
  int pos=0;
  char aux[MAXVAR];

  strcpy (aux,var);
  strcat (aux,"=");

  while (e[pos]!=NULL)
    if (!strncmp(e[pos],aux,strlen(aux)))
      return (pos);
    else
      pos++;
  errno=ENOENT;   /*no hay tal variable*/
  return(-1);
}

int CambiarVariable(char * var, char * valor, char *e[])
{
  int pos;
  char *aux;
  if ((pos=BuscarVariable(var,e))==-1)
      return(-1);

  if ((aux=(char *)malloc(strlen(var)+strlen(valor)+2))==NULL)
      return -1;
  strcpy(aux,var);
  strcat(aux,"=");
  strcat(aux,valor);
  e[pos]=aux;
  return (pos);
}
```

NOTES ON UIDS A process has three user credentials (real efective and saved uids). The efective represents the process' privileges (what it can atually do: accessing files, sending signals . . .), the real represents what user is actually behind the execution of that process, and the saved is used to decide which changes are allowed. We change the efective user id with the *setuid()* system call.

The usual thing is for *setuid()* to fail to change one process' credentials with `permission denied` or `not owner` errors. That's ok, *setuid()* will only succeed in these cases

- the effective user uid of the process calling *setuid()* is 0 (user `root`), in this case *setuid()* changes all (real, effective and saved) uids
- real, effective and saved uids of the calling process are not the same, in this case the ONLY changes allowed are
  - change effective uid to be the same as the real uid
  - change effective uid to be the same as the saved uid

Executing a file with the *setuid* bit set changes de effective and saved uids of the executing process to that of the owner of the file

How can I get a running process tha has different real and saved uids?

- Compile your program and put it into a directory that everybody has access (for example (`/tmp`)
- Change its mode to rwsr-xr-x (for example `chmod 4755 /tmp/a.out`)
- Login as other user and execute the file

Here you'll find the shell functions that allow you to print and change the uids

```
char * NombreUsuario (uid_t uid)
{
  struct passwd *p;

  if ((p=getpwuid(uid))==NULL)
return (" ??????");
  return p->pw_name;
}

uid_t UidUsuario (char * nombre)
{
   struct passwd *p;
   if ((p=getpwnam (nombre))==NULL)
     return  (uid_t) -1;
   return p->pw_uid;
}


void MostrarUidsProceso (void)
{
   uid_t real=getuid(), efec=geteuid();

   printf ("Credencial real: %d, (%s)\n", real, NombreUsuario (real));
   printf ("Credencial efectiva: %d, (%s)\n", efec, NombreUsuario (efec));
}

void CambiarUidLogin (char * login)
{
  uid_t uid;

  if ((uid=UidUsuario(login))==(uid_t) -1){
        printf("loin no valido: %s\n", login);
        return;
  }

  if (setuid(uid)==.1)
    printf ("Imposible cambiar credencial: %s\n", strerror(errno));
}
```

EXAMPLE

We'll see how it works with an example. First we'll see how it fails when we simply execute our shell

```
antonio@abyecto:~/c/Shell-2020$ ./a.out
-> uid
Credencial real: 1000, (antonio)
Credencial efectiva: 1000, (antonio)
-> uid 1001
Imposible cambiar credencial: Operation not permitted
-> uid -l visita
Imposible cambiar credencial: Operation not permitted
-> fgas visita ls -l /home/visita
Imposible cambiar credencial (Operation not permitted).
 Ejecutable debe ser setuid (rwsr-xr-x)
No ejecutado: Operation not permitted
->
```

Now we do it properly, first we prepare the executable as user antonio (a.out is the executable file obtained from user antonio compiling the shell)

```
antonio@abyecto:~/c/Shell$ ls -l a.out
-rwxr-xr-x 1 antonio antonio 59920 Nov 12 18:58 a.out
antonio@abyecto:~/c/Shell$ cp a.out /tmp
antonio@abyecto:~/c/Shell$ chmod 4755 /tmp/a.out
antonio@abyecto:~/c/Shell$ ls -l /tmp/a.out
-rwsr-xr-x 1 antonio antonio 59920 Nov 12 19:05 /tmp/a.out
antonio@abyecto:~/c/Shell$
```

Note that the executable file has the setuid bit set. Now we enter the machine as user visita (the names of the users need not be the same in your machine, it is assumed you create the users yourself) and we can change credentials between visita and antonio

```
visita@abyecto:~$ cd /tmp
visita@abyecto:/tmp$ ./a.out
->  uid
Credencial real: 1001, (visita)
Credencial efectiva: 1000, (antonio)
-> uid 1001
-> uid
Credencial real: 1001, (visita)
Credencial efectiva: 1001, (visita)
-> uid -l antonio
-> uid
Credencial real: 1001, (visita)
Credencial efectiva: 1000, (antonio)
->
```

In fact we can now execute the run-as command (note that we can only access the files of the effective credential)

```
-> uid
Credencial real: 1001, (visita)
Credencial efectiva: 1000, (antonio)
-> ls /home/antonio
....
bin                     java              Public
c                       ....
-> ls /home/visita
ls: cannot open directory '/home/visita': Permission denied
-> fgas visita ls /home/visita
Desktop    Downloads  mail  Music      Pictures  Templates
Documents  Dropbox    mbox  nohup.out  Public    Videos
-> fgas antonio ls /home/antonio
....
bin                     java              Public
c                       .......
-> fgas visita ls /home/antonio
ls: cannot open directory '/home/antonio': Permission denied
-> fgas antonio ls /home/visita
ls: cannot open directory '/home/visita': Permission denied
->
```

NOTES ON EXECUTION

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();
```

Example of execution of program `/usr/bin/xterm` in the foreground

```
.......
if ((pid=fork())==0){
    if (execl("/usr/bin/xterm","xterm","-l",NULL)==-1)
      perror ("Cannot execute");
    exit(255); /*exec has failed for whateever reason*/
    }
waitpid (pid,NULL,0);
```

Example of execution of program `/usr/bin/xterm` in the background

```
.......
if ((pid=fork())==0){
    if (execl("/usr/bin/xterm","xterm","-l",NULL)==-1)
      perror ("Cannot execute");
    exit(255); /*exec has failed for whatever reason*/
    }
/*parent process continues here..*/
```

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 (the exact format tha our function *TocearCadena* used)

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 example checks the status of process with pid pid, whis is supposedly executing in the background.

```
 if (waitpid (pid,&valor, WNOHANG |WUNTRACED |WIFCONTINUED)==pid){
      /*the integer valor contains info on the status of process pid*/
      }
 else {
      /*the integer valor contains NO VALID INFORMATION, process state has not cha
      since we last checked */
      }
```

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");
}
```