Operating Systems

Grado en Informática. Course 2020-2021

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.

getpriority [**pid**]. Shows the priority of process *pid*. If *pid* is no given specified, priority of the process executing the shell is shown

setpriority [**pid**] [**value**]. If both arguments (*pid and value*) are specified, the priority of process *pid* will be changed to *value*. If only one argument is given the shell's priority will be changed to that value. If no arguments are given the priority of the process executing the shell is to be shown

getuid Prints the real and effective user credentialas of the process running the shell (both the number and the associated login)

setuid [**-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

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. If no arguments are given, this command behaves exactly as *getuid*

execute **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).

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

foreground **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).

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

executes prog to *pri*

background 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.

background prog arg1 arg2...@pri 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.

run-as login prog arg1 arg2...[@pri] [&] Creates a process that tries to execute as user *login* the program and arguments *prog arg1 arg2 ....* Execution is on the background or in the foreground deppending on the & as the last argument. Optionally, priority can be changed with *@pri* like in the previous commands. 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)

execute-as login prog arg1 arg2...[@pri] Tries to execute as user *login* the program and arguments *prog arg1 arg2 ....* Execution is to be done without creating a process. Optionally, priority can be changed with *@pri* like in the previous commands. 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 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 *foreground* command. When supplying an **&** as the last arg to a program, the execution must be in background: exactly as the *background* 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 *foreground prog arg1 arg2...*

For example, to execute `ls -l /home /usr` in the foreground we can do

```
-> foreground 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 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 *background prog arg1 arg2...*

prog **arg1 arg2... @pri** [**&**] 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 *foreground prog arg1 arg2 @pri ...*, and *prog arg1 arg2 ... @pri &* is the same as *background prog arg1 arg2 @pri ...*

Examples

```
#) foreground ls -l /usr
#) foreground du -a / @12
#) ls -lisa /home
#) du -a /usr @12
#) background xterm -e bash
#) background xterm -bg yellow @10
#) xclock &
#) xclock -update 1 @7
#) xclock -update 1 @7 &
#) run-as  manolo ls -lisa /home/manolo @12
.......
#) execute-as josefa du -hs /home/josefa/Desktop
.......
```

listprocs 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 argu-

ments)

- 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* 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

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

deleteprocs -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 January the 7th, 2021, 23:00

Assesment: During lab hours

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 Cmd_getuid (char *tr[])
{
   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 Cmd_setuid (char *tr[])
{
   uid_t uid;
   int u;
   if (tr[0]==NULL || (!strcmp(tr[0],"-l") && tr[1]==NULL)){
         Cmd_getuid(tr);
         return;
   }
   if (!strcmp(tr[0],"-l")){
        if ((uid=UidUsuario(tr[1]))==(uid_t) -1){
             printf ("Usuario no existente %s\n", tr[1]);
             return;
        }
   }
   else if ((uid=(uid_t) ((u=atoi (tr[0]))<0)? -1: u) ==(uid_t) -1){
        printf ("Valor no valido de la credencial %s\n",tr[0]);
        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
-> getuid
Credencial real: 1000, (antonio)
Credencial efectiva: 1000, (antonio)
-> setuid 1001
Imposible cambiar credencial: Operation not permitted
-> setuid -l visita
Imposible cambiar credencial: Operation not permitted
-> run-as 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-2020$ ls -l a.out
-rwxr-xr-x 1 antonio antonio 59920 Nov 12 18:58 a.out
antonio@abyecto:~/c/Shell-2020$ cp a.out /tmp
antonio@abyecto:~/c/Shell-2020$ chmod 4755 /tmp/a.out
antonio@abyecto:~/c/Shell-2020$ ls -l /tmp/a.out
-rwsr-xr-x 1 antonio antonio 59920 Nov 12 19:05 /tmp/a.out
antonio@abyecto:~/c/Shell-2020$
```

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, its assumed you create the users yourself) and we can change credentials between visita and antonio

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

```
-> getuid
Credencial real: 1001, (visita)
Credencial efectiva: 1001, (visita)
-> setuid -l antonio
-> getuid
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)

```
-> getuid
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
-> run-as visita ls /home/visita
Desktop    Downloads  mail  Music      Pictures  Templates
Documents  Dropbox    mbox  nohup.out  Public    Videos
-> run-as antonio ls /home/antonio
....
bin                     java               Public
c                       .......
-> run-as visita ls /home/antonio
ls: cannot open directory '/home/antonio': Permission denied
-> run-as 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");
}
```