Operating Systems

Grado en Informática. Course 2018-2019

Lab Assignment 3: Processes

CONTINUE the coding of the shell started in the 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's code).

The shell will keep track (using a list) of the processes created to execute programs in background. For the implementation of this list (linked, array, array of pointers ...), groups must use the same type of list used the previous lab assignment.

The shell will also keep a list of directories (the comand *searchlist* manages this list) where to find executables for both the *exec* comand and the execution with process commands (either background or foreground) so that for an external program to be executed either a complete pathname to it is given or it resides in one of the directories in the *searchlist*. (We WILL NOT USE THE *execvp* system call, but the *execv* system call that DOES NOT use the system PATH; this *searchlist* is the equivalent in our shell to the PATH in the system's shell). A nearly complete and easy to use implementation of the searchlist with a external variable is given later in this documenmt.

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

searchlist Displays the *searchlist*: List of directories where the shell searchs for executable files.

searchlist +dir Adds directory *dir* to the search list

searchlist exfile Displays the complete pathname to executable file *exfile* (should it exist). Equivalent to the system command *which* (Except that *which* uses the directories in the PATH environment variable)

searchlist -path Directories in the `PATH` environment variable are added to the searchlist

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). **The execv system call should be used**: for *prog* to be executed either it resides in one of the directories of the *searchlist* or a complete pathname to it must be given.

exec 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*

prog arg1 arg2... The shell creates a process that 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 (they can be more than two). As in the previous *exec command*, **The execv system call should be used**: for *prog* to be executed either it resides in one of the directories of the *searchlist* or a complete pathname to it must be given

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 (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 *jobs* shows this list. **PROGRAMS THAT READ FROM THE STANDARD INPUT SHOULD NO BE EXECUTED IN BACKGROUND**.

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 *jobs* shows this list. **PROGRAMS THAT READ FROM THE STANDARD INPUT SHOULD NO BE EXECUTED IN BACKGROUND**. Examples:

```
#) xterm
   cannot execute: xterm not found
#) searchlist
#) searchlist +/bin
#) searchlist +/usr/bin
#) searchlist +.
#) searchlist
  /bin
```

```
  /usr/bin
  .
#) searchlist xterm
  /usr/bin/xterm
#) xterm
#) background xterm -fg green @12
#) background  xterm @10
#) searchlist ls
  /bin/ls
#) exec ls -l
total 112
-rw-r--r-- 1 antonio antonio  4334 Nov 10 13:34 Lists.c
-rw-r--r-- 1 antonio antonio  1304 Nov 10 13:32 Lists.h
-rw-r--r-- 1 antonio antonio 14128 Nov 10 13:34 Lists.o
-rw-r--r-- 1 antonio antonio   121 Nov  2 21:30 Makefile
-rwxr-xr-x 1 antonio antonio 45672 Nov 16 13:28 Shell
-rwx------ 1 antonio antonio 16590 Nov 16 13:28 Shell.c
antonio@abyecto:~/Desktop/OS-LAB$
```

jobs 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**

```
#) jobs
  4793     SIGNALED (SIGKILL) p=-1 Fri Oct 19 2018 12:35 xterm -e bash
  4801     ACTIVE             p=0  Fri Oct 19 2018 12:37 xclock -update 1
  4802     STOPPED (SIGSTOP)  p=-1 Fri Oct 19 2018 12:39 xterm -fg green
  4840     TERMINATED (255)   p=-1 Fri Oct 19 2018 12:40 xterm
->
```

proc pid Shows information on process *pid* (provided *pid* represents a back-

ground 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 jobs.

clearjobs Empties the list of background processes. Process continue to execute, they simply are not shown in the shell's list anymore.

pipe prog arg1 arg2 ... % progB argB1 argB2 ... The shell creates two processes, one of the executing prog with arguments arg1, arg2 ... and othe executing progB with arguments argB1, ... So that the standard output of the first process is redirected to the standard input of the second process.

```
-> searchlist -path
-> pipe -ls -l /usr/local % wc -l -c
     13     676
->
```

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

- Work must be done in pairs.
- The source code will be submitted to the subversion repository under a directory named **P3**
- The name of the main program will be `shell.c`. Tha list of the previous assigment will use files *list.h* and *list.c*, the list for processes in background should use files *listproc.h* and *listproc.c*. An adecuate `Makefile` should be provided.
- Only one of the members of the workgroup will submit the source code. The names, logins and **D.N.I.** 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 Friday 14, 2018, 23:00

Assessment: For each pair, it will be done in its corresponding group, during the lab classes

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 programas 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

```
.......
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 use the *execv()* system call. *execv*

does not search in the PATH for the executable, we must supply execv with a complete *pathname* to the executable file.

An easy implementation of the *searchlist* using an external (global) variable

```
#define MAXNOMBRE 1024
#define MAXSEARCHLIST 128

char *searchlist[MAXSEARCHLIST]={NULL};

int SearchListAddDir(char * dir)
{
  int i=0;
  char * p;

  while (i<MAXSEARCHLIST-2 && searchlist[i]!=NULL)
     i++;

  if (i==MAXSEARCHLIST-2)
    {errno=ENOSPC; return -1;} /*no cabe*/
  if ((p=strdup(dir))==NULL)
     return -1;
  searchlist[i]=p;
  searchlist[i+1]=NULL;
  return 0;
}


void SearchListNew()
{
  int i;

  for (i=0; searchlist[i]!=NULL; i++) {
     free(searchlist[i]);
     searchlist[i]=NULL;
  }
}

void SearchListShow()
{
   int i;

   for (i=0; searchlist[i]!=NULL; i++)
```

```
     printf ("%s\n",searchlist[i]);
}

void SearchListAddPath()
{
  char *aux;
  char *p;

  if ((p=getenv("PATH"))==NULL){
    printf ("Imposible obtener PATH del sistema\n");
    return;
    }
  aux=strdup(p);
  if ((p=strtok (aux,":"))!=NULL && SearchListAddDir(p)==-1)
        printf ("Imposible anadir %s: %s\n", p, strerror(errno));
  while ((p=strtok(NULL,":"))!=NULL)
    if (SearchListAddDir(p)==-1){
        printf ("Imposible anadir %s: %s\n", p, strerror(errno));
        break;
    }
  free(aux);
}

char * BuscarEjecutable(char * ejec)
{
  static char aux[MAXNOMBRE];
  int i;
  struct stat s;

  if (ejec==NULL)
    return NULL;
  if (ejec[0]=='/' || !strncmp (ejec,"./",2) || !strncmp (ejec,"../",2)){
    if (stat(ejec,&s)!=-1)
      return ejec;
    else
      return NULL;
  }
  for (i=0;searchlist[i]!=NULL; i++){
    sprintf(aux,"%s/%s",searchlist[i],ejec);
    if (stat(aux,&s)!=-1)
        return aux;
  }
  return NULL;
```

```
}
```

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");
}
```