

## TP II – Procesos Pesados

**Objetivo:** Desarrollar una serie de funciones que encapsulen cierta funcionalidad vinculada con la gestión de procesos y consola para luego reutilizarlas en la construcción de un Shell.

### **Funcionalidades (features) del Shell propuesto**

- ◆ Ejecución de procesos en primer plano (foreground)
- ◆ Ejecución de procesos en segundo plano (background)
- ◆ Redirección de la entrada y la salida standard en ambos tipos de ejecución de procesos
- ◆ Notificar al usuario acerca del id de proceso creado en segundo plano (background)

### **Introducción**

El shell o intérprete de comandos del SO es un programa que corre como una aplicación más de usuario y nos permite (a través de comandos y acciones definidas en el programa) poder realizar en forma interactiva algunas acciones sobre el SO sin necesidad de escribir un programa. Una de las principales acciones es la ejecución de procesos.

Cuando el usuario interactúa directamente con el shell, el esquema es el siguiente:

**usuario <=> intérprete de comandos (shell) <=> llamadas al sistema (system calls) <=> núcleo SO (kernel) <=> hardware**

Cuando el usuario ejecuta un proceso en primer plano (foreground) a través del shell, el proceso pasa a ser un proceso hijo del proceso shell y el control de la terminal del usuario (que antes estaba bajo control del shell) pasa al proceso hijo. El esquema es el siguiente:

**usuario <=> proceso <=> intérprete de comandos (shell) <=> llamadas al sistema (system calls) <=> núcleo SO (kernel) <=> hardware**

Cuando el usuario ejecuta un proceso en segundo plano (background) a través del shell, el proceso también pasa a ser un proceso hijo del shell, pero en este caso el control del terminal sigue estando en manos del proceso shell y habría una ejecución concurrente entre el proceso shell y hijo ejecutándose en segundo plano. El esquema es el siguiente:

**usuario <=> intérprete de comandos (shell) <=> llamadas al sistema (system calls) <=> núcleo SO (kernel) <=> hardware**

**proceso <=> intérprete de comandos (shell) <=> llamadas al sistema (system calls) <=> núcleo SO (kernel) <=> hardware**

### **Acerca de la Entrada, la Salida y el Error**

Todo proceso tiene asociado tres flujos de datos (streams): la entrada standard (`stdin`, por lo general asociado al dispositivo teclado), la salida standard (`stdout`, por lo general asociado al dispositivo monitor) y el error standard (`stderr`, por lo general asociado al dispositivo monitor). Estos flujos pueden redirigirse hacia otros destinos o bien desde otros orígenes. La redirección



puede hacerse a través de instrucciones dentro del programa:

```
// en un sistema gnu, puede redirigirse directamente, conectando un nuevo flujo
// a stdin, stdout, stderr
// Ejemplo: ahora la salida standard se guardara en el archivo salida.txt
fclose(stdout);
stdout = fopen("salida.txt", "w");
```

Otra forma de uso de redirección por programa, sería duplicar el descriptor de un archivo, podríamos hacer -por ejemplo- que stdout (que habitualmente esta asociado al descriptor de archivo 0 (STDOUT\_FILENO)) también se redirija a "salida.txt". Se puede utilizar la función `fcntl()` (archivo de cabecera `<fcntl.h>`), pero existe la función `dup2()` (archivo de cabecera `<unistd.h>`) de uso más cómodo. Esto es particularmente útil cuando queremos redirigir un proceso hijo del shell:

```
// abre el archivo salida.txt de output, si no existe lo crea, si existe trunca
// su contenido.
// Asocia este archivo con la salida standard del proceso actual
outfile = open("salida.txt", O_WRONLY|O_CREAT|O_TRUNC);
dup2(outfile, STDOUT_FILENO);
close(outfile);
```

En la redirección interactiva se utilizan los signos de mayor (>), menor (<). Ejemplo: enviar la salida de `prog1` al archivo `salida.txt`:

```
$ prog1 > salida.txt
```

que `prog1` tome la entrada del archivo `entrada.txt`:

```
$ prog1 < entrada.txt
```

que `prog1` tome la entrada del archivo `entrada.txt` y la salida la envíe a `salida.txt`:

```
$ prog1 < entrada.txt > salida.txt
```

## Acerca de los Procesos en Primer Plano y Segundo Plano

La ejecución de procesos en primer plano (foreground) implica que el proceso tiene el control de la terminal o consola que esta operando el usuario, por lo general, se trata de procesos interactivos y es la forma habitual de invocar a los procesos.

La ejecución de procesos en segundo plano (background) es un tipo de ejecución no interactiva, que remite a los tiempos de los sistemas de procesos por lotes (batch); muchos procesos no requieren de la interacción con el usuario. No obstante, el usuario, como operador del sistema, puede monitorear todo tipo de procesos a través del comando `ps` (estado de los procesos (process status)) :

```
$ ps -af
UID      PID  PPID  C  STIME TTY          TIME CMD
knoppix  3721  3697  0  18:13 tty5        00:00:00 /bin/bash /usr/bin/adriane
knoppix  4069  3721  0  18:13 tty5        00:00:00 dialog --title ADRIANE --no-ok
knoppix  5095  4690  0  18:27 pts/0    00:00:00 ps -af
$
```

Para ejecutar un proceso en segundo plano (background) se utiliza el símbolo `&` al final del comando; por ejemplo para ejecutar `prog1` en segundo plano, suponiendo que este programa tiene algún tipo de entrada y salida, lo apropiado sería redirigirla<sup>1</sup>:

```
$ prog1 < entrada.txt > salida.txt &
```

En este caso, nuestro shell debería informar al usuario el número de proceso creado a fin de

<sup>1</sup> Si el programa requiriera algún tipo de entrada por teclado y estuviera en segundo plano, quedaría bloqueado esperando una entrada que no se concretaría.



poder seguirlo con el comando `ps`. El comando `jobs` también es utilizado para visualizar los procesos en background:

```
$ jobs -l
[1]+  5564 Stopped (tty output)    vi
$
```

Se recomienda que el shell realice en forma automática la redirección de salida, en caso de que el usuario no lo indique.

## Sugerencias

El programa principal del shell propuesto, podría tener la siguiente forma:

```
// shell - programa principal
int main(void) {
    inicio();           // inicializo variables, estructuras, etc.
    ingreso();          // ingreso comando
    while(!salir) {    // mientras no sea el comando para salir ...
        proceso();     // proceso comando ingresado
        libero();      // libero memoria dinamica usada
        ingreso();     // ingreso comando
    }                  // fin-mientras
    return 0;
}
```

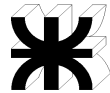
El ingreso trata del ingreso del comando en la consola por parte del usuario y luego su análisis (parsing) para determinar qué tipo de comando es: un comando propio del shell (ejemplo: comando `salir` para salir del programa) o un proceso a ejecutar. Efectivamente, se trata de un programa que realizará spawning de cada proceso que indique el usuario.

Como el shell es interactivo, se supone que no puede haber más de un `ingreso()` al mismo tiempo, es decir, que se procesa "de a un comando ingresado por vez", por lo tanto, podríamos contar con una estructura en donde almacenar toda la información del comando actual:

```
// shell - rutina de ingreso de comando por teclado
struct {
    pid_t pid;           // id de proceso del shell
    pid_t pgid;         // id del grupo de proceso del shell
    int interactivo;    // ¿comando en foreground?
    int input;          // descriptor de archivo de stdin del shell
    char *comando;      // comando ingresado por el usuario
    char *error;        // error (de parsing, etc.) del ultimo comando ingresado
    char *arg[64];      // cada elemento es una parte significativa del comando
                        // ingresado por el usuario
    int nargs;          // cantidad de partes significativas que tiene el
                        // comando actual (no puede superar de 64)
    int op;             // luego del parsing, se determino este tipo de comando
    int salir;          // ¿salgo del shell?
    char *entrada;      // nombre de archivo al cual redirigir entrada
    char *salida;       // nombre de archivo al cual redirigir salida
} sh;
```

El ingreso del usuario podría tener la siguiente forma:

```
// shell - rutina de ingreso de comando por teclado
void ingreso() {
    prompt();           // simbolo a mostrar por pantalla
    ingresoLinea();    // ingreso de una linea de texto por teclado
    parseoLinea();     // separacion de la linea ingresada en "tokens"
}
```



## Resuelva

**1. El código de la función `ingresoLinea()` teniendo en cuenta que no sabemos cuántos caracteres puede ingresar el usuario hasta presionar la tecla `Intro`. Sugerencia: leer de a un caracter de `stdin`:**

```
char c;  
...  
c = (char) getc(stdin);
```

usar `malloc()` para asignar dinámicamente un buffer `n` bytes, ir almacenando en dicho buffer los caracteres leídos, cuando se llene el buffer, volver a pedir más memoria con `realloc()`:

```
// nbuf cuenta los caracteres almacenados en buf  
char *buf = (char *) malloc(nbytes); // puede devolver NULL  
...  
if ( (nbuf % nbytes) == 0 ) {  
    nbufsize+=nbytes;  
    buf = (char *) realloc(buf,nbufsize); // puede devolver NULL  
    ...  
}  
...  
sh.comando = buf; // agregar un caracter nulo al final: buf[nbuf]='\0';
```

**2. El código de la función `parseoLinea()`. Esta función tiene como entrada `sh.comando`, debe hacer algunas validaciones, separar a `sh.comando` en las partes significativas del comando<sup>2</sup> y actualizar: `sh.arg`, `sh.narg`, `sh.entrada`, `sh.salida`, `sh.error` (si encontrara algun error), etc.. Para separar `sh.comando` se recomienda usar `strtok()`, haciendo una copia previa del string a parsear:**

```
// intrucciones posibles, parte de la funcion parseoLinea()  
...  
int i=0;  
// alloco espacio para que buffer pueda contener una copia de sh.comando  
char *buffer = (char *) malloc(strlen(sh.comando)+1);  
...  
strcpy(buffer,sh.comando); // buffer=sh.comando  
// separe a buffer en i partes separadas por uno o mas espacios  
buffer = strtok(sh.comando, " ");  
while (i < 64 && buffer != NULL) { // mientras haya mas partes ...  
    sh.arg[i] = buffer; // apunto a la i-esima parte  
    buffer = strtok(NULL, " "); // busco la parte siguiente  
    i++; // incremento el contador de partes  
} // fin-mientras  
// i es igual a la cantidad de partes en que se dividio sh.comando  
sh.narg=i;  
...
```

**3. Compile y ejecute el siguiente programa C:**

```
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <sys/types.h>
```

<sup>2</sup> El usuario puede tipear `n` espacios entre el nombre del programa a ejecutar y el primer argumento o el segundo, etc. en el arreglo `sh.arg[]` solo deben guardarse el programa, los argumentos, etc., sin espacios innecesarios.



```
int main(void) {
    int a=4;
    printf("padre: id de proceso=%d\n", (int) getpid());
    printf("padre: a=%d\n",a);
    printf("padre:lanzo fork()!\n");
    pid_t pid = fork();
    switch (pid) {
        case -1: // error, padre
            printf("padre:Error en fork()!\n");
            break;
        case 0: // hijo
            printf("hijo: id de proceso=%d\n", (int) getpid());
            printf("hijo: a=%d\n",a);
            printf("hijo: sumo 1 a variable a\n");
            a++;
            printf("hijo: a=%d\n",a);
            break;
        default: // padre
            printf("padre: sumo 10 a variable a\n");
            a+=10;
            printf("padre: a=%d\n",a);
            break;
    }
    return 0;
}
```

Responda:

3.1 ¿Qué sucede luego de la llamada a la función fork()?

3.2 ¿Cuál es el valor final de la variable a para el proceso padre? ¿Por qué?

3.3 ¿Cuál es el valor final de la variable a para el proceso hijo? ¿Por qué?

3.4 Ejecute varias veces el programa. Es posible que no vea todos los mensajes enviados a la consola con printf() de la forma esperada, ¿A qué se debe esto?

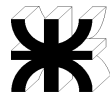
**4. La familia de funciones exec (execv(), execl(), execve(), execl(), execvp(), execlp()) permiten reemplazar la imagen del proceso actual con la imagen de un programa guardado en disco. Por lo tanto, suelen utilizarse luego de hacer un fork() para reemplazar al proceso hijo con una imagen de otro proceso. Esto es lo que debería hacer nuestro shell cuando un usuario invoque un comando. Modifique el programa del punto 3, en la sección de código correspondiente al proceso hijo, entre la última instrucción printf("hijo: a=%d\n",a); y la instrucción break; que le sigue, introduzca el siguiente código C:**

```
...
    switch(pid) {
...
        case 0: // hijo
...
            printf("hijo: a=%d\n",a);
            const char *programa = "ls";
            char *argumentos[] = { "ls","-l",NULL };
            int ret = execvp(programa,argumentos);
            printf("hijo: termine de ejecutar execv()=%d!\n",ret);
            break;
...
    }
...
}
```

*Nota: se pretende ejecutar el comando "ls" (list directory) con el argumento "-l" (guión "ele", long format).*

Responda:

4.1 ¿Por qué no aparece en pantalla el printf() enviado por el proceso hijo luego



de ejecutar la función `execvp()`?

4.2 ¿Existe alguna diferencia en la ejecución de este programa con respecto a la versión anterior, hasta antes de ejecutar la función `execvp()`?

4.3 ¿Hay algún cambio en la secuencia de instrucciones ejecutadas por el proceso padre?

4.4 Averigüe todo lo que más pueda en cuanto a la funciones `exec()`

**4 bis). Si Ud. es muy observador, posiblemente habrá notado que debe presionar Intro para volver nuevamente al shell, luego del cambio agregado en el punto anterior. También podrá observar que el padre imprime todos sus mensajes y luego puede verse la ejecución del hijo, esto implicaría que el padre finalizó su ejecución antes que el proceso hijo (este sería un zombie). Modifique el programa del punto anterior, en la sección de código correspondiente al proceso padre, entre la última instrucción `printf("padre: a=%d\n",a);` y la instrucción `break;` que le sigue, introduzca el siguiente código C:**

```
...
    switch(pid) {
...
        case default:    // padre
...
            printf("padre: a=%d\n",a);
            printf("padre: id proceso hijo=%d\n",(int) pid);
            printf("padre: esperando que hijo termine...\n");
            int estado;
            waitpid(pid,&estado,WUNTRACED);
            printf("padre: hijo termino con estado=%d\n",estado);
            break;
    }
...

```

Responda:

4 bis.1 Observe ahora la ejecución del programa y su salida, ¿Cuál es el último mensaje que aparece en pantalla?

4 bis.2 Averigüe todo lo que más pueda en cuanto a la función `waitpid()`

4 bis.3 Haga un pequeño cambio en el programa y agregue otro argumento más a la lista de argumentos del comando `ls`, por ejemplo `"afafjakfjasfkjasñlfjkasklñ"`, ejecute nuevamente el programa y observe si cambia el estado de retorno del proceso hijo

4 bis.4 En el código padre, ¿Qué valor representa la variable `pid`?

**5. Combinando lo aprendido en los puntos anterior, la función `proceso()` podría tener el siguiente formato:**

```
void proceso() {
...
    switch(sh.op) {
        case 0:                // comando salir
            sh.salir=1;        // salir = true
            break;
        case 1:                // usuario pretende ejecutar comando
            pid_t pid = fork();
            switch (pid) {
                case -1:        // error, padre
                    printf("Error en fork()!\n");
                    break;
                case 0:         // hijo
...
                    setpgrp(); // proceso hijo como nuevo lider del grupo de procesos

```



```
pid_t chpid = getpid(); // id de proceso del hijo
if (sh.interactivo) // si es un proceso interactivo (foreground)
    tcsetpgrp(sh.input, chpid); // debe controlar la terminal
else // proceso batch (background)
    printf("Proceso [%d] ejecutado en background\n", (int) chpid);

ejecutoProceso();
if (printError()) exit(EXIT_FAILURE); // retorno con error
else exit(EXIT_SUCCESS); // retorno Ok
break;
default: // padre
...
    if (sh.interactivo) {
        tcsetpgrp(sh.input, pid); // el nuevo proceso controla la terminal
        // espero por la terminacion del proceso hijo interactivo (foreground)
        int terminationStatus;
        while(waitpid(pid, &terminationStatus, WUNTRACED | WNOHANG) == 0);
        tcsetpgrp(sh.input, sh.pgid); // el shell retoma control de terminal
    }
    break;
}
break;
}
...
}
```

Responda:

- 5.1 ¿Qué operaciones debería llevar a cabo la función `ejecutoProceso()`?
- 5.2 ¿En qué caso se ejecutaría la instrucción `if (printError()) ...`?

**6. Termine de unir las piezas faltantes (`ejecutoProceso()`, `inicio()`, `libero()`, etc.) para lograr un primera versión de este shell de comandos.**

**7. Hasta aquí solo hemos visto una única instrucción vinculada con la sincronización de procesos, ¿A qué instrucción nos estamos refiriendo?**

**8. Modifique el programa para agregarle el comando `cd`, cuando el usuario tipee este comando, deberá mostrar en pantalla el directorio de trabajo actual. Sugerencia: ver función `getcwd()`.**

**9. Modifique el programa para agregarle el comando `cd <nuevo directorio>`, cuando el usuario tipee este comando, el programa deberá cambiar el directorio actual por `<nuevo directorio>`. Sugerencia: ver función `chdir()`.**

**10. Cuando aprenda algo más acerca de la sincronización de procesos y señales, vuelva a esta primera versión del programa shell y modifíquelo para que el mismo le informe al usuario los procesos batch que van finalizando y en qué estado van finalizando.**