



Práctica 1

INTRODUCCIÓN A PROCESOS Y PROGRAMACIÓN DE SISTEMA EN UNIX

Ej. 1. Conteste las siguientes preguntas. Puede ser útil usar **strace** para monitorear las llamadas a sistema de un proceso, o **top** para ver la tabla de procesos del sistema.

- a) Si se cierra el file descriptor de la salida estándar (1) ¿qué pasa al escribir al mismo?
- b) Si se cierra el file descriptor de la entrada estándar (0) ¿qué pasa al intentar leer del mismo?
- c) Si un file descriptor se duplica con **dup()** ¿qué pasa al cerrar una de las copias?
- d) Al hacer **fork()**, ¿cómo cambia el valor de **getpid()**? ¿Y al hacer **exec()**?
- e) Con **fork()**, cree dos procesos y haga que el hijo termine (con **exit()** o retornando del **main**) y que el padre duerma indefinidamente sin hacer **wait()**. ¿Cómo aparece el hijo en la tabla de procesos? ¿Por qué sigue existiendo?
- f) Al hacer un **malloc** de 1GB ¿aumenta el uso de memoria de un proceso? Explique.
- g) ¿Qué pasa con el uso de memoria de un proceso al realizar **fork()**? ¿Y **exec()**?
- h) ¿Qué pasa con los file descriptor de un proceso al hacer **fork()**? ¿Y **exec()**?
- i) El comando de cambio de directorio **cd** suele ser un *built-in* de la shell. ¿Puede implementarse mediante un programa al igual que, ej., **ls**?

Ej. 2. El comando **yes** imprime líneas conteniendo una ‘y’ infinitamente. Es usado para simular una respuesta afirmativa para instaladores o programas similares (i.e. “sí a todo”) haciendo simplemente **yes | ./installer**. ¿Cómo piensa que está implementado? Al ejecutar un pipeline como el anterior ¿cómo es el uso de CPU del proceso **yes**?

Ej. 3. ¿Qué pasa cuando un proceso no libera su memoria (con **free()**) antes de terminar?

Ej. 4. ¿Es **free()** una llamada al sistema? ¿Por qué sí o por qué no?

Ej. 5. ¿Es **getchar()** una llamada al sistema? ¿Por qué sí o por qué no? ¿Cómo funciona **ungetc()**?

Ej. 6 (Mini Shell). Implemente una versión mínima de una shell. El programa deberá esperar líneas por entrada estándar, y al recibir una ejecutar el comando correspondiente, de la misma manera que lo hace, por ejemplo, **bash**.

- a) Implemente una versión básica que simplemente ejecuta el comando y espera que el mismo termine antes de pedir otro. El comando puede especificarse por su path completo o solamente por su nombre si se encuentra en algún directorio del **\$PATH** (pista: ver **execvp()**). Ejemplo:

```
$ ls
Makefile shell.c shell
```

- b) Agregue la posibilidad de pasar argumentos a los comandos, ejemplo:

```
$ ls /  
bin boot dev etc ...
```

- c) Implemente redirección de la salida estándar. Ahora los comandos pueden tener la forma `cmd > file`, causando que la salida de `cmd` sea escrita *directamente* al archivo `file`. La shell **no** debe recibir la salida y escribirla al archivo, sino que a medida que el comando `cmd` escriba a su salida estándar, esta salida vaya directamente al archivo.

Puede usar `open("archivo.txt", O_CREAT | O_WRONLY | O_TRUNC, 0644)` para abrir (o crear) un archivo con permisos usuales. Otras referencias: `man 2 open`, `man 2 close`, `man 2 dup`.

Ejemplo:¹

```
$ ls / > salida.txt  
$ cat salida.txt  
bin  
boot  
dev  
...
```

- d) Implemente pipes. Modifique el mini shell para que acepte dos comandos por vez. Mediante el uso de la función `int pipe(int pip[2])`, haga que la salida del primer comando sea la entrada del segundo.

La shell debe poder tomar comandos de la forma `c1 | c2` causando que la salida del comando `c1` sea dirigida automáticamente a la entrada del comando `c2`. Ningún proceso espera a que el otro termine: ambos inician inmediatamente. Un *pipeline* puede tener longitud arbitraria (i.e. se debe soportar `c1 | c2 | c3`, etc). Ver también `man 2 pipe`. Ejemplo:

```
$ ls / | sort -r  
$ var  
usr  
tmp  
...
```

Ej. 7. El siguiente programa intenta corregir con gracia una división por cero, atrapando la señal correspondiente (`SIGFPE`) y modificando el denominador de la división en ese caso.

```
int denom = 0;  
void handler(int s) { printf("ouch!\n"); denom = 1; }  
int main() {  
    int r;  
    signal(SIGFPE, handler);  
    r = 1 / denom;  
    printf("r = %d\n", r);  
    return 0;  
}
```

¿Qué pasa al correr el programa? ¿Por qué?

¹Nota: `ls` detecta si su salida estándar no está asociada a una terminal, y en ese caso imprime un archivo por línea. Por ello se nota una diferencia con el ejemplo anterior.

Ej. 8. Complete el código para capturar la señal al presionar Ctrl-C.

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>

void INThandler(int);

int main(void)
{
    signal(...);
    while (1)
        sleep(10);
}

void INThandler(int sig)
{
    char c;
    signal(...);
    printf("OUCH, did you hit Ctrl-C?\n"
           "Do you really want to quit? [y/n] ");
    c = getchar();
    if (c == 'y' || c == 'Y')
        exit(0);
    else
        signal(...);
    getchar();
}
```

Ej. 9 (Signal Pong). Hacer un programa que tenga el siguiente comportamiento:

Luego del fork, el padre envía una señal `SIGUSR1` al hijo y entra en un loop infinito. El hijo espera en un loop infinito. Ambos (padre e hijo) cuando reciben una señal `SIGUSR1` responden lo mismo. (el padre al hijo y el hijo al padre). ¿Qué pasa si cambiamos los loops infinitos por `pause()`?. Usar las funciones `signal()` y `signalaction()`.

Ej. 10. Corra en una terminal el comando: `echo "$$"`

¿Qué información recibe?

Investigue los comandos:

- `env`
- `ps`
- `tree`