Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencias de la Computación
R-322 — Sistemas Operativos I

Práctica 1

Introducción a Procesos y Programación de Sistema en UNIX

Ej. 1. Corra en una terminal el comando: `echo "$$"`

¿Qué información recibe?

Investigue los comandos:

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

Ej. 2. 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. 3. 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. 4. ¿Qué pasa cuando un proceso no libera su memoria (con `free()`) antes de terminar?

Ej. 5. ¿Es `free()` una llamada al sistema? ¿Por qué sí o por qué no?

Ej. 6. ¿Es `getchar()` una llamada al sistema? ¿Por qué sí o por qué no? ¿Cómo funciona `ungetc()`?

Ej. 7 (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. 8. 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() {
```

[1] Nota: `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.

```
    int r;
    signal(SIGFPE, handler);
    r = 1 / denom;
    printf("r = %d\n", r);
    return 0;
}
```

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

Ej. 9. 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. 10 (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 reciven 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. 11 (Servidor de turnos). El archivo `skel_server.c` implementa un pequeño servidor que recibe conexiones por un puerto TCP (4040) y responde a cada pedido con un entero *único*. Los pedidos, enviados

por los clientes, son simplemente una línea `NUEVO` terminada por `\n`. Para cerrar una conexión, el cliente envía `CHAU`.

Como está escrito, el servidor sólo puede atender a un cliente a la vez, dejando en espera a todo el resto hasta que se cierre la conexión con el primer cliente.

Para recibir conexiones TCP, el proceso debe:

- Llamar a `socket(AF_INET, SOCK_STREAM, 0)` para conseguir un socket (un fd)
- Usar `bind` para asociarlo a un puerto.
- Usar `listen` para permitir que acepte conexiones
- Llamar a `accept` ahora bloquea hasta que se reciba una conexión. `accept` devuelve un fd representando a la conexión, y puede usar `read`/`write` sobre el mismo.

Tareas:

a) Modifique el servidor para atender concurrentemente a todas las conexiones abiertas levantando un nuevo proceso por cada conexión. Nota: todos los clientes deberán poder hacer pedidos sin esperar a otros, y siempre debe poder conectarse un nuevo cliente.

b) Use memoria compartida entre los procesos para mantener el último entero enviado a un cliente. ¿Qué necesita tener en cuenta para garantizar que dos pedidos nunca reciben el mismo entero?

c) Investigue la función `select` (o su alternativa moderna `epoll`). ¿Qué ventaja trae usarla/s?

d) Opcional: implemente una solución con `select`/`epoll`.

Ej. 12 (Mini memcached). Implemente un servidor que provea un *key-value store* a sus clientes. El servidor debe esperar conexiones en el puerto 3942 TCP (`AF_INET, SOCK_STREAM`) y atender los pedidos de cada cliente. Un pedido es siempre una secuencia de palabras separadas por espacios, terminado por un caracter de nueva línea (`'\n'`). La primer palabra es el comando y el resto (alguna cantidad) son los argumentos al comando. Ninguna palabra contiene espacios ni caracteres no alfanuméricos. Los pedidos posibles son:

- `PUT` k v: introduce al store el valor v bajo la clave k. El valor viejo para k, si existía, es pisado. El servidor debe responder con `OK`.
- `DEL` k: Borra el valor asociado a la clave k. El servidor debe responder con `OK`.
- `GET` k: Busca el valor asociado a la clave k. El servidor debe contestar con `OK` v si el valor es v, o con `NOTFOUND` si no hay valor asociado a k.

Ante cualquier otro mensaje el servidor responde con `EINVAL`. Las respuestas del servidor siempre terminan con `'\n'`. (El servidor puede probarse fácilmente con `netcat`.) Por supuesto, deben soportarse conexiones simultáneas de varios clientes. Ver también:

- Beejs' Guide to Network Programming - `https://beej.us/guide/bgnet/html/`
- Man pages: `man 2 socket`, `man 2 bind`, `man 2 listen`, `man 2 accept`

Referencias:

- Beejs' Guide to Network Programming - `https://beej.us/guide/bgnet/html/`
- man 2 socket, man 2 bind, man 2 listen, man 2 accept