Taller de *syscalls* y señales

Sistemas Operativos

21 de marzo de 2024

Primer cuatrimestre - 2024

1. Ejercicios

1.1. Ejercicio 1

Escribir un programa en `C` que recibe 3 parámetros de entrada: `N` para la cantidad de procesos a crear (menor a 10), `K` la cantidad de rondas y `J` para el número maldito (`J` tiene que ser menor a `N` y mayor o igual 0). Una vez que se crearon los `N` procesos (cada uno con un identificador), empieza el siguiente juego:

- Luego de la creación de todos los hijos, el proceso padre envía una señal de SIGTERM a cada uno de los hijos, con un segundo de diferencia entre ellos.
- Cada hijo saca al azar un número del 0 al `N`.
- Cuando los hijos reciben la señal del padre deben hacer lo siguiente: Si el número sacado al azar es maldito, deben expresar sus últimas palabras y luego deben terminar su ejecución. En caso contrario, sobrevive.
- Se tiene que repetir `K` rondas este procedimiento.
- Una vez finalizadas las rondas, se declara ganador a los hijos que sobrevivieron. El padre debe notificar al usuario que hijos sobrevivieron, con identificador y su PID.
- El padre debe enviar una señal de SIGKILL a los hijos restantes y finalizar su ejecución.

Ayuda: Recordar que la `syscall wait` devuelve el PID del hijo que finalizó (en caso de éxito).

1.2. Ejercicio 2

Utilizar el comando `strace` para analizar el comportamiento del programa correspondiente al archivo binario `hai64` cuyo uso es el siguiente:

```
./hai64 [PROGRAMA]
```

donde `PROGRAMA` es la ruta de otro programa con sus parámetros de entrada[1].
Luego, responder las siguientes preguntas:

- ¿Cuántos procesos se lanzan y qué comportamiento se puede observar de cada uno?
- ¿Utilizan los procesos alguna forma de IPC?¿Cuál es y para qué se usa en este caso?
- ¿Qué puede deducir del programa a partir de las syscalls observadas?

[1] Por ejemplo: ./hai64 echo “imprimir este texto” , ./hai64 ls, etc.

1.3. Ejercicio 3

Escribir un programa en C que presente el *mismo* comportamiento que el programa analizado en el punto anterior. Es decir, que se observe la misma salida al ser ejecutado por un usuario con los mismos argumentos y que las respuestas para el punto anterior sean las mismas.[2]

[2]Sugerencia: Usar como base el programa `hai.c`.

2. Notas útiles para la resolución del taller

2.1. Preliminares

Para la resolución del taller es conveniente repasar la clase práctica 1 "Syscalls y Señales", así como poder entender el código de los programas provistos junto con la misma y haberlos compilado y ejecutado por separado.

2.2. `strace`

`strace` es una herramienta que permite generar una traza legible de las syscalls usadas por un programa dado. Sintaxis:

```
$ strace [opciones] comando [argumentos]
```

Algunas opciones útiles:

- `-q`: Omite algunos mensajes innecesarios.
- `-o <archivo>`: Redirige la salida a `<archivo>`.
- `-f`: Traza también a los procesos hijos del proceso trazado.

2.3. Familia syscalls `exec`

La familia de funciones `Exec` nos permite reemplazar la imagen del proceso actual por una nueva. Siempre toma por primer parámetro la ruta del archivo ejecutable que se quiera reemplazar como imagen. Luego, se le puede indicar si se quieren agregar variables de entorno o parámetros para esa imagen nueva. Las funciones son: `execl`, `execlp`, `execle`, `execv`, `execvp`, `execve`, `execvpe`. Cada letra luego del prefijo `exec`, nos indica un significado particular de lo que hace cada función:

- `l`: Indica que la función es variádica. Toma una secuencia de argumentos que se le pasa a la imagen a reemplazar. Es útil cuando sabemos de antemano la cantidad de parámetros a utilizar. El último parámetro tiene que ser `NULL`.

  ```
  $ execl [pathname] [argumento1] [argumento2] ... [argumentoN] [NULL]
  ```

- `v`: Indica que la función toma un array de punteros a char con los parámetros a usar.

  ```
  $ execv [pathname] [Array Argumentos]
  ```

- `e`: Indica que se le pueden pasar variables de entorno, tanto como variádica como array.

  ```
  $ execve [pathname] [Array Argumentos] [Array Variables de entorno]
  ```

- `p`: Indica que el nombre pasado por filename, por default lo busque en el pathname que indica la variable de entorno `PATH`. Por ejemplo, si utilizamos `execvp('ls', ['-a'])`, buscará el nombre del comando `ls` dentro de los pathname de `PATH`.

  ```
  $ execvp [filename] [Array Argumentos]
  ```

2.4. Argumentos de invocación

Recordar que en `C`, para acceder a los argumentos de invocación del programa se hace de la siguiente forma: En la función `main`, recibimos dos parámetros:

```
int main(int argc, char const *argv[]){
    ...
}
```

- int `argc`: Indica la cantidad de parámetros recibidos (si no pasamos ningún parámetro, recibimos 1 como valor)
- char const `*argv[]`: Array de punteros a los nombres de los argumentos. **Importante**: notar que siempre la primera posición es el nombre del programa

Por ejemplo, si se llama a:

```
./programa_ejemplo parametro1 parametro2
```

La variable `argc` nos devolverá 3 como valor. La variable `argv` sería de la forma ['programa_ejemplo', 'parametro1', 'parametro2'].

2.5. *Includes* recomendados

```
#include <sys/wait.h>
#include <unistd.h>
#include <syscall.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
```

2.6. Otros

En los headers `<sys/syscall.h>` se encuentran definidos símbolos para cada una de las *syscalls* del sistema. Por ejemplo, el número de *syscall* de `write` está definido por el símbolo `SYS_write`.

Recordar que la función `signal`, llama a la `syscall rt_sigaction`. Para más información se puede consultar el manual con el comando `man`.