Lab fork

Índice

El objetivo de este lab es familiarizarse con las llamadas al sistema `fork(2)` (que crea una copia del proceso actual) y `pipe(2)` (que proporciona un mecanismo de comunicación unidireccional entre dos procesos).

> **REQUERIDO**: para la entrega es condición **necesaria** que se haya aplicado el **formato** de código mediante `make format` (ver `README.md` del repositorio).

> **IMPORTANTE**: leer el archivo `README.md` que se encuentra en la raíz del proyecto. Contiene información sobre cómo realizar la compilación de los archivos, y cómo ejecutar el formateo de código.

> **AVISO**: las diapositivas están en fork

Bibliografía sobre syscalls

Tanto en el caso de syscalls del sistema operativo, como funciones de la biblioteca estándar de C, se puede consultar su documentación mediante el comando `man`. Esto es particularmente recomendable en el caso de *syscalls* como `stat(2)`, que son complejas y tienen muchos flags: `man 2 stat`. En las páginas de manual también se indican los *headers* (a.k.a `#include ...`) necesarios para cada syscall. (Las páginas de manual también se pueden consultar online en sitios web como dashsash.io y man7.org.)

En general, una buena referencia sobre sistemas POSIX es **[KERR]**. En particular, para este lab son relevantes:

- Tarea ***primes***: §6.1, §6.2, §24.2, §44.2
- Tarea ***xargs***: §6.6, §24.1, §26.1, §27.1, §27.2

Opcionalmente se puede leer el capítulo 3 a modo de introducción.

Esqueleto

> **AVISO**: El esqueleto se encuentra disponible en **fisop/fork**.

> **IMPORTANTE**: leer el archivo `README.md` que se encuentra en la raíz del proyecto. Contiene información sobre cómo realizar la compilación de los archivos, y cómo ejecutar el formateo de código.

Integración

Suponiendo que ya se **clonó** el repositorio privado en algún directorio:

```
git clone git@github.com:fiubatps/sisop_<año_cuatrimestre>_<apellido_alumno> fork
cd fork
```

Para integrar el esqueleto de la cátedra, hacer:

- **asegurarse** de estar en la rama **main**

  ```
  git checkout main
  ```

- **agregar remoto** de la cátedra

  ```
  git remote add catedra https://github.com/fisop/fork
  ```

- **creación** de la rama **base**

  ```
  git checkout -b base_fork
  git push -u origin base_fork
  ```

- **merge** del esqueleto

  ```
  git fetch --all
  git merge catedra/main --allow-unrelated-histories
  git push origin base_fork
  ```

- **creación** de la rama **entrega**

  ```
  git checkout -b entrega_fork
  git push -u origin entrega_fork
  ```

> **IMPORTANTE**: asegurarse de siempre commitear en la rama **entrega_fork**.

Compilación

Simplemente alcanza con ejecutar `make`.

Pruebas

Alcanza con ejecutar `make test`.

Introducción: pingpong

> Este ejercicio es solo para demostración en clase, y **no** forma parte de la entrega.

Vamos a escribir un programa en C que use `fork(2)` y `pipe(2)` para enviar y recibir (ping-pong) un determinado valor entero (creado con `random(3)`), entre dos procesos.

El programa debe imprimir por pantalla la secuencia de eventos de ambos procesos, en el formato que se especifica a continuación:

```
$ ./pingpong
Hola, soy PID <x>:
  - IDs del primer pipe: [3, 4]
  - IDs del segundo pipe: [6, 7]

Donde fork me devuelve <y>:
  - getpid me devuelve: <?>
  - getppid me devuelve: <?>
  - valor random: <v>
  - envío valor <v> a través de fd=?

Donde fork me devuelve 0:
  - getpid me devuelve: <?>
  - getppid me devuelve: <?>
  - recibo valor <v> vía fd=?
  - reenvío valor en fd=? y termino

Hola, de nuevo PID <x>:
  - recibí valor <v> vía fd=?
```

Tips:

- Nótese que como las tuberías —*pipes*— son unidireccionales, se necesitarán dos para poder transmitir el valor en una dirección y en otra.

- Para obtener números aleatorios que varíen en cada ejecución del programa, se debe inicializar el *PRNG* (generador de números pseudo-aleatorios) mediante la función `srandom(3)` (típicamente con el valor de `time(2)`).
- Si `fork(2)` fallase, simplemente se imprime un mensaje por salida de error estándar (*stderr*), y el programa termina.
- Tener en cuenta el tipo de los valores de retorno para cada una de las *syscalls*/funciones de *libc* a utilizar (por ejemplo: `random(3)`)
- Utilizar `wait(2)` con el único propósito de no dejar al proceso hijo en estado *zombie*.
- **No** se puede utilizar ninguna función similar a `sleep(3)` para sincronizar a los procesos.

Llamadas al sistema: `fork(2)`, `pipe(2)`, `wait(2)`, `getpid(2)`, `getppid(2)`.

Tarea: primes

Descripción

La criba de Eratóstenes (sieve of Eratosthenes en inglés) es un algoritmo milenario para calcular todos los primos menores a un determinado número natural, *n*.

Si bien la visualización del algoritmo suele hacerse "tachando" en una grilla, el concepto de criba, o *sieve* (literalmente: cedazo, tamiz, colador) debe hacernos pensar más en un filtro. En particular, puede pensarse en *n* filtros apilados, donde el primero filtra los enteros múltiplos de 2, el segundo los múltiplos de 3, el tercero los múltiplos de 5, y así sucesivamente.

Si modelásemos cada filtro como un proceso, y la transición de un filtro al siguiente mediante tuberías *(pipes)*, se puede implementar el algoritmo con el siguiente pseudo-código (ver fuente original, y en particular la imagen que allí se muestra):

```
p := <leer valor de pipe izquierdo>

imprimir p // asumiendo que es primo

mientras <pipe izquierdo no cerrado>:
    n = <leer siguiente valor de pipe izquierdo>
    si n % p != 0:
        escribir <n> en el pipe derecho
```

(El único proceso que es distinto, es el primero, que tiene que simplemente generar la secuencia de números naturales de 2 a *n*. No tiene lado izquierdo.)

Requisitos

La interfaz que se pide es:

```
$ ./primes <n>
```

donde *n* será un número natural mayor o igual a 2.

El código debe crear una estructura de procesos similar a la mostrada en la imagen, de tal manera que:

- El primer proceso cree un proceso derecho, con el que se comunica mediante un *pipe*.
- Ese primer proceso, escribe en el *pipe* la secuencia de números de 2 a *n*, para a continuación cerrar el *pipe* y esperar la finalización del proceso derecho.
- Todos los procesos sucesivos aplican el pseudo-código mostrado anteriormente, con la salvedad de que son responsables de crear a su "hermano" derecho, y la tubería (*pipe*) que los comunica.
- Se debería poder ejecutar correctamente el programa con un *N* mayor o igual a 10000.
- **No** se debe realizar ninguna clase de optimización que simplifique el algoritmo.

Ejemplo de uso:

```
$ ./primes 35
primo 2
primo 3
primo 5
primo 7
primo 11
primo 13
primo 17
primo 19
primo 23
primo 29
primo 31
```

Llamadas al sistema: `fork(2)`, `pipe(2)`, `wait(2)`.

Tarea: xargs

Descripción

La utilidad `xargs(1)` permite ejecutar un *binario* una o más veces pasándole como argumentos, los leídos desde la *entrada estándar*. Los argumentos están separados por un *delimitador* como el *espacio* o el `'\n'` (pero también se pueden indicar otros mediante la opción `-d`).

Por ejemplo, si se invoca:

```
$ seq 10 | xargs -n 4 /bin/echo
```

El resultado será:

```
1 2 3 4
5 6 7 8
9 10
```

El comando `seq(1)` genera una secuencia de números hasta *N* y `-n 4` le indica a `xargs` que tiene que *empaquetar* los argumentos leídos de a 4 (en este caso). Finalmente, por cada *paquete* ejecuta el comando `/bin/echo`.

Esto sería equivalente a ejecutar lo siguiente:

```
$ /bin/echo 1 2 3 4
$ /bin/echo 5 6 7 8
$ /bin/echo 9 10
```

Para ampliar y conocer el comportamiento de una implementación de `xargs` moderna, y por ejemplo sus opciones `-r`, `-0`, `-I` y `-P`, consultar la página de manual.

Requisitos

La interfaz que se pide es:

```
$ ./xargs <comando>
```

donde *comando* es un binario que no recibe argumentos extras (como por ejemplo `ls` o `echo`).

El código debe satisfacer los siguientes puntos:

- Leer los argumentos **línea a línea**, **nunca** separados **por espacios** (se recomienda usar la función `getline(3)`). También, es necesario eliminar el caracter `'\n'` para obtener el nombre del archivo.
- Almacenar cuanto mucho `NARGS` argumentos en un *buffer*. Es decir, asumir que el *stream* de datos podría ser "infinito".
- El "empaquetado" vendrá definido por un valor entero positivo disponible en la macro `NARGS` (disponible en esqueleto). [1]
- **Siempre** se pasan `NARGS` argumentos al *comando* ejecutado (excepto en su última ejecución, que pueden ser menos).
- Se debe esperar **siempre** a que termine la ejecución del comando actual.

Ejemplo de uso:

```
$ seq 10 | ./xargs /bin/echo
1 2 3 4
5 6 7 8
9 10
```

Llamadas al sistema: `fork(2)`, `wait(2)`, execvp(3).

Desafíos

> Las tareas listadas aquí no son obligatorias, y refieren a los challenges.

Más utilidades Unix

Implementar, a elección, dos o más de los siguientes comandos. Cada comando tiene una cantidad de puntos basado en la dificultad relativa de su implementación. Como mínimo para cumplir el desafío se deben alcanzar los 10 puntos.

Las opciones son:

- `ps` (5pts): el comando *process status* muestra información básica de los procesos que están corriendo en el sistema. Se pide **como mínimo** una implementación que muestre el pid y comando (i.e. argv) de cada proceso (esto es equivalente a hacer `ps -eo pid,comm`, se recomienda compararlo con tal comando). Para más información, leer la sección `ps0`, de uno de los labs anteriores. Ayuda: leer `proc(5)` para información sobre el directorio `/proc`.
- `find` (2pts): el comando buscará y mostrará por pantalla todos los archivos del directorio *actual* (y subdirectorios) cuyo nombre contenga (o sea igual a) `xyz`. Se pide **como mínimo** una implementación equivalente a la que se indica en la sección `find`, de lab *unix*.
- `ls` (2pts): el comando *list* permite listar los contenidos de un directorio, brindando información extra de cada una de sus entradas. Se pide implementar una versión simplificada de `ls -al` donde cada entrada del directorio se imprima en su propia línea, indicando *nombre*, *tipo* (con la misma simbología que usa `ls`), *permisos* (se pueden mostrar numéricamente) y *usuario* (aquí nuevamente alcanza con mostrar el uid). Si se trata de un *enlace simbólico*, se debe mostrar a qué archivo/directorio apunta el mismo. Ayuda: es similar a `find`, pero deben incorporar `stat(2)` y `readlink(2)`.
- `cp` (3pts): implementar el comando *copy* de forma eficiente, haciendo uso de `mmap` tal y como se describe en la sección `cp1` de uno de los labs anteriores. Solo se pide soportar el caso básico de copiar un archivo regular a otro (i.e. `cp src dst` donde `src` es un archivo regular y `dst` no existe). Si el archivo destino existe, debe fallar; y si el archivo fuente no existe también. Ayuda: `mmap(2)` y `memcpy(3)`. Se recomienda implementar primero una versión simplificada con `write` y `read`; y luego optimizarlo con `mmap`.
- `timeout` (5pts): el comando *timeout* realiza una ejecución de un segundo proceso, y espera una cantidad de tiempo prefijada (e.g. `timeout duración comando`. Si se excede ese tiempo y el proceso sigue en ejecución, lo termina enviándole SIGTERM. Se pide implementar una versión simplificada del comando `timeout`. La implementación debe usar *señales*. Ayuda: `timer_create(2)`, `timer_settime(2)`, `kill(2)`.

strace(1)

La utilidad `strace` (de *system-call trace*) permite ejecutar un proceso y a la vez registrar todas las llamadas al sistema que éste realiza, junto con los parámetros y valores de retorno. Se trata de un programa complejo si se quiere seguir procesos que están compuestos por threads o realizan fork. Sin embargo, en su versión más sencilla (con un único proceso) es bastante fácil de seguir.

Un ejemplo del uso de strace para inspeccionar el proceso `echo` podría ser:

```
$ strace -e trace=read,write,open,execve echo hi
execve("/bin/echo", ["echo", "hi"], 0x7ffcbeba00b8 /* 27 vars */) = 0
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20\35\2\0\0\0\0"...,
832) = 832
read(3, "# Locale name alias data base.\n#"..., 4096) = 2995
read(3, "", 4096)                       = 0
write(1, "hi\n", 3hi
)                    = 3
+++ exited with 0 +++
```

Parte 1

Leer la documentación sobre la utilidad `strace`. Eligiendo al menos otros tres comandos de Unix (que no sean *echo*), mostrar las syscalls que usan, identificando aquellas que conozcan. Explicar lo que strace imprime por pantalla, y si aparecen más llamadas de las esperadas. ¿De dónde provienen?.

Correr `strace` sobre `strace`, e inspeccionar qué syscalls utiliza. Prestar especial atención en `wait(2)` y `ptrace(2)`.

Parte 2

Luego de familiarizarse con la herramienta, se pide *programar* una versión sencilla de `strace`. Los requisitos mínimos de tal implementación son:

- Debe imprimir *todas* las llamadas al sistema, junto con su valor de retorno. Alcanza con indicar el número de syscall, en x86_64 pueden usar ésta tabla para encontrar el *nombre* de la syscall. No hace falta imprimir los argumentos de las syscalls.

- Debe soportar la ejecución de un único proceso, del que asumiremos no realiza `fork` ni `exec`; no maneja threads ni señales especiales.
- No es necesario que soporte flags adicionales, y el formato de salida es libre mientras esté la información esperada.
- Se debe incluir una breve explicación en prosa comentando cada parte del código.

Se debe, además de proveer el código funcionando; incluir evidencia del funcionamiento (e.g. corridas con programas de prueba, comparaciones con `strace` original, etc).

Ayuda: leer detenidamente `ptrace(2)`, ya que el funcionamiento de `strace` se basa fuertemente en esa syscall. Las macros descritas en `wait(2)` (como `WIFSTOPPED`) son de utilidad.

1. Tiene efecto durante el proceso de compilación. Para más información consultar la documentación ↩