Clase del 4 de abril del 2017

Comunicación entre Procesos

Pipes

```
int pipe(int pipefd[2]);
```

Recibe dos file descriptors. El 0 es de lectura y 1 de escritura. Se usa para comunicarse entre procesos. Cada proceso debe cerrar el que no usa luego del fork.

Acepta file descriptors de prácticamente cualquier cosa. Para leer y escribir se usan los read y write de siempre.

Funciones útiles

```
int dup (int oldfd);
```

dup Devuelve un nuevo file descriptor que es equivalente al primero. Cualquier operación que haga sobre el segundo se harán como si se le hicieran al primero. Podría decirse que ambos `fd` apuntan al mismo lugar. Si se hace un `close` de uno ambos descriptores dejan de ser válidos.

```
int dup2 (int oldfd, int newfd);
```

Variante interesante del anterior a la cual le especificamos a qué file descriptor debe ser copiado el `oldfd`. Si el `newfd` estaba abierto, lo cierra.

Para qué puede servir esto? Para implementar los pipes y la redirección de bash, por ejemplo. Por ejemplo, quiero que los `printf` no escriban a salida estandar, sino a un file descriptor abierto por mi que apunta a un archivo (típico caso de redirección). Esto se logra muy fácilmente con `dup2 (fd_propio, fd_stdout);` de forma que ahora el file descriptor que se usa para todas las escrituras a salida estandar ahroa escriban en mi file descriptor. Tip: `fd_stdout` es 1 (0 es entrada standard y 2 es error standard).

FIFOS

Los FIFOS son pipes con nombre. Son interesantes porque son parte del file system (i.e. aparecen cuando ponemos `ls`).

```
mknod (const char * pathname, mode_t mode, dev_t dev);
```

Creación

Operación El primer parámetro dice a dónde se crea. El segundo lo usaremos con `S_IFIFO` (la syscall es mucho más general, hay un wrapper específicamente para fifo que se llama `mkfifo`). El último parámetro no lo usamos.

A diferencia de pipe, que abría ambos file descriptors, corresponde abrirlos con `open`, pasando la ruta, sea para lectura o escritura. El file descriptor lo obtendremos en ese `open`. Se escribirá y leerá con `read` y `write` y se cerrará con `close`.

Destrucción Se lo eliminará del File System como a un archivo común, con `unlink`, que lo elimina cuando no haya ningún proceso que lo tenga abierto (cuando se haya hecho el último close).

Comparación

Un FIFO funciona, como su nombre lo indica, como una cola. Cuando se hace un read algo desaparece. Los pipes tienen capacidad máxima (`PIPE_BUF`) y pueden producir un overflow, acá es un archivo y si hay una capacidad máxima será muy grande y dependerá del filesystem.

Además de ser un mecanismo de comunicación, ambas son, de forma automática, un mecanismo de sincronización (a diferencia de la memoria compartida).

Comportamiento Bloqueante Ambos casos son bloqueantes por default. Se puede cambiar esto con `fcntl(descriptor, F_SETFL, O_NONBLOCK)` en pipes y `O_NONBLOCK` en el open para fifos, pero no es lo recomendado. Si uno lo abre para escribir y escribe, no puede hacer nada hasta que otro lea, y viceversa.

También es bloqueante el `open`. Si se abre para lectura, el `open` bloquea hasta que se abra en otro proceso para escritura y viceversa.

Si se lee de algo no abierto para escritura, devolverá 0 (EOF). Se tiene que haber abierto en algún momento porque el open era bloqueante. Esto simboliza que ya todos los escritores lo cerraron. Si está abierto, pero vacío, se bloquea.

En cuanto a escritura, si está no abierto para lectura se emite una señal `SIGPIPE` (que por default termina el programa - OJO, puede que haya que hacer handler para esto!). Si está abierto la operación de escritura es atómica si `bytes < PIPE_BUF` (y se bloquea hasta que se pueda escribir).

Caso pésimo de uso no bloqueante: hacer un while en un proceso que chequee todo el tiempo (busywait) mientras haya un `EAGAIN` (error de "no hay nada" en un no bloqueante.). Si hacemos esto en un parcial se desaprueb. Terminaríamos teniendo que bloquear por otro método, cuando bien podría hacerlo la FIFO o la Pipe.

Respecto al `SIGPIPE`, si sincronizamos los `close` de forma que solo se lea si hay escritores, nunca recibiremos esta señal. Si hay alguna falta de sincronización y no queremos que se cierre el programa, será obligatorio poner un handler.

Comunicación Bidireccional

Ambos mecanismos recién vistos son unidireccionales, con lo cual si quiero bidireccionalidad, hago uno de ida y uno de vuelta. Esto es así de sencillo, pero tenemos que sincronizar los `open` y `close`, porque si ambos hacen open para la ida y luego para la vuelta, ambos open quedarán bloqueando sus respectivos procesos. Esto es una situación de Starvation irrecuperable.

Múltiples Canales

Uno lee, muchos escriben. La escritura es atómica, con lo cual uno no interrumpirá al otro. Luego para leer no hay problemas de sincronismo.

Caso inverso: muchos leen de uno que escribe.

- Primer caso: quiero que alguno lo lea. Por ejemplo, alertas de cosas. Problema: **la lectura no es atómica**. Se puede interrumpir a uno a media lectura y que otro lea, con lo cual el primero puede leer fragmnentos no consecutivos (basura). Habráque sincronizar esas lecturas de alguna forma.
- Segundo caso: se quiere que todos lean todo. **Tarea para pensar**

Y muchos con muchos? **También tarea**.

Aunque hay patrones como clonador, manager/worker, etc, hay que ver si no hay herramientas del lenguaje que lo permiten.

Queda como duda: qué casos de uso tienen de diferencia pipes y named FIFOS?

Ejemplos en Clase

Ej1: wrapper de pipe. Lo interesante es que al escribir, la primera vez que lo haga cierra el file descriptor de lectura. OJO: ese wrapper no funciona bien en caso de que primero escriba y después lea (y viceversa): se cierran ambos fd y falla tratando de escribir a un fd vacío.

Cuando hacemos `read(cant_bytes)`, si hay menos bytes en el pipe, el `read` sabe que está leyendo de un pipe y el pipe tiene la cantidad de cosas restantes y por eso cierra. No hace falta que llegue un end of file para que devuelva la cantidad de bytes leídos. Está en el usuario decidir qué hacer si no se leyó la cantidad de bytes espeardos.

Ej2: se usa el getter de getFd para hacer un pisado de la entrada standard y usar cin.

Ej3: FIFO. El wrapper tiene especializada la FIFO en dos clases, una para lectura y una para escritura. En este caso tenemos el bloqueo del open, que en los ejs anteriores no tuvimos, ya que eso se hacía una sola vez y antes de hacer el fork.

Erlang

Libro: *"A Concurrent Approach to Software Developement - Programming Earlang - O Reilly"* Otro: "*Joe Amstrong*". El segundo es más ameno y bueno para motivarse, mientras que el primero es más completo.

Erlang es un lenguaje dinámico, similar a Python en muchos aspectos.

Además de la ley de Moore y todos los límites que eso impone a lo no concurrente, está el tema de la similitud: el mundo es concurrente, y si resolvemos problemas del mundo, muchas veces es más fácil pensar en esos términos. Hay lenguajes orientados a la concurrencia, como Erlang o Go que tienen procesos muy livianos (no los del sistema operativo) y los manejan con facilidad.

En Erlang tenemos **módulos** en lugar de clases y podemos lanzar cada uno una o muchas veces de forma concurrente (?). La clave para que funcionen bien es que no compartan estado y se hace todo por paso de mensajes. Además suelen funcionar con colector de memoria, uno por proceso, cada uno con un área de memoria muy chica.

Erlang suma a eso que es un **lenguaje funcional**. Los datos no son mutables (estilo declarativo). Esto significa que una igualdad es una declaración sobre el mundo. Si `A = 7` significa que A es equivalente a 7 y no puede serlo a ninguna otra cosa. Si luego queremos hacer un igual a otra cosa, dará error.

Erlang tiene un intérprete interactivo (comando `erl`).

Detalles sintácticos

- Todas las operaciones terminan en `.` (si no, los enters no cuentan como fin de instrucción).
- Las variables empiezan con mayúscula (las que empiezan por minúscula son átomos).

- Tipos básicos: listas y tuplas (encerrados llaves). Todo puede tener cosas heterogéneas, pero se suelen usar listas homogéneas y tuplas de lo que sea (estilo struct).
- El contador empieza en 1. Para acceder a un elemento de una colección se usa `element(1, coleccion).`.
- Comentarios con `%` estilo LaTeX.

```
Persona1 = {persona, "Ana", "La Plata", "1988"}
{Tipo, Nombre, Ciudad, Fecha} = Persona1
```

Igualdad Las igualdades para cosas asignadas son aseveraciones también, que no asignan nada si la igualdad se cumple, y da error si no se cumple (*"no match"*). Es un pattern matching, de forma que si queremos obtener los elementos de una lista,

El case es como un switch y tiene la sintaxis:

```
case expresion of
  c1 ->
    accion_c1;
  c2 ->
    accion_c2;
  else ->
    accion_default;
```

Pero hay una mejor forma de hacer esto si lo que quiero hacer es chequear el tipo y el tipo es la primera con pattern matching:

```
area({square, Side}) -> Side*Side.
area({circle, Radius}) -> Radius * Radius * math:pi().
```

En este fragmento `square` y `circle` son átomos, y `Side` y `Radius` serán variables. Las instrucciones especifican el valor de la función `area` para 2 patrones. Cuando se haga, por ejemplo, `area({square, 4})`, se hará un pattern matching donde square va a coincidir y Side se asignará a 4, de forma que se devolverá 4*4.

Otro ejemplo es:

```
factorial(0) -> 1
factorial(N) -> N * factorial(N-1)
```