Memoria compartida

Hasta ahora nuestro único mecanismo de sincronización y comunicación eran las syscalls `wait` y `exit`, lo cual no nos daba mucha variedad de oportunidades. Si para nuestros objetivos eso alcanza, se usa eso, no hay por qué complicársela, pero usualmente no sucede.

Cuando se crea un proceso nuevo con `fork` se copia toda la memoria (stack, heap, etc.). Cada proceso tiene su propio espacio en la memoria física. Están aislados entre sí, lo cual es algo fundamental en la seguridad de linux. Por más que tengan variables apuntando a la misma dirección de memoria virtual, se traducirán en direcciones físicas distintas.

System V (system five) y POSIX son los protocolos típicos de memoria compartida que solucionan el problema de comunicar procesos entre sí. Se implementa una región de memoria accesible por ambos procesos. Puede leerse bien del Bach. Vamos a usar el primero, que es más complejo, pero no es exclusivo de UNIX, como es el POSIX. Es el que vamos a usar en el TP y en el parcial.

Crear la región de Memoria Compartida

Para crear una región de memoria compartida tenemos la siguiente syscall:

```
int shmget(key_t key, size_t size, int flags);
```

Solicita al OS un espacio de memoria compartida de tamaño `size`. El Sistema Operativo la reserva en un área fuera del espacio del proceso (a diferencia del `new` de memoria dinámica que reserva en el heap del proceso). La función devuelve un id a nivel sistema operativo que representa esa región. El sistema operativo maneja un área de regiones apuntadas por una tabla de memoria compartida. Cuando hacemos `smhget` se crea una entrada en la tabla que apuntará a una región.

Una ventaja inmediata de esto respect a la dupla `fork`, `wait` es que el hijo no tiene que morir (bíblico el asunto), se puede hacer en cualquier momento de la ejecución. Otra es que pueden ser procesos sin relación (no hace falta que uno haya creado al otro).

La `key` es justamente lo que hace que distintos procesos usen la misma región. Si se llama get desde distintos procesos, pero con la misma clave, el id resultante será el mismo. Más adelante vemos cómo se generan esas claves.

En cuanto a las flags, los 9 bits menos significativos son los permisos (estilo linux, especificados con tres dígitos octales). Las otras flags se le adosan con constantes y pipes (ors). Algunas flags importantes son `IPC_CREAT`, para crear la región crea si no existe y `IPC_EXCL` (exclusividad), que se usa en conjunto con la anterior, y produce que si se intenta crear una región ya existente, de un error.

El `EXCL` sirve también para evitar condiciones de carrera. Uno tiene que crear la región y otro se tiene que unir. Los que se unen no deberían usar ni `CREAT` ni `EXCL`: se supone que la región existe.

Si esto falla se devuelve negativo y en `errno` (variable global) aparece el número de error. Recordar que con `perror(const char *s)` se obtiene el mensaje de error.

Una llamada típica a esta syscall es del estilo

```
shmget(key, sizeof(int), 0664 | IPC_CREAT | IPC_EXCL);
```

Generación de claves

```
key_t ftok(const char * pathname, int proj_id);
```

Genera y devuelve una clave de System V a partir del contenido de un archivo (referido con su path). El siguiente parámetro es un número entero. Se hace alguna operación desconocida estilo función de hashing entre ambos parámetros para producir la clave. La idea es que los elementos de un mismo sistema utilicen el mismo archivo, pero varíen el número. El archivo debe existir y el proceso debe tener permisos para leerlo (usa su contenido).

Attach

```
void * shmat(int shmid, void * shmaddr, int flags)
```

Attach: adosado de la memoria compartida al espacio de memoria virtual del proceso, así ahora lo puede referenciar. Recibe la id que devolvía el `shmget`, opcionalmente le podemos decir en qué lugar de la memoria virtual del proceso queremos el puntero (si ponemos `NULL` se ocupa el Sistema Operativo) y flags (solo lectura, por ejemplo).

Devuelve un puntero al segmento comienzo del segmento de memoria adosado. -1 si falla. El kernel mantiene también en la tabla la cantidad de procesos que hicieron attach.

A partir del attach el puntero se maneja como si hubiéramos hecho un new.

OJO con los structs y la cantidad de espacio. Si le agregamos un campo en un proceso y no recompilamos el otro, se arma la podrida con la cantidad de espacio.

Detach

Así como tenemos new y delete para memoria dinámica, acá tenemos attach y detach.

```
int shmdt(const void * addr);
```

Solo recibe el puntero que había pasado antes. Lo único que hace es romper el vínculo entre mi proceso y la región de memoria compartida. ¿Por qué recibe un puntero? En este esquema podría tener más de un puntero apuntando a la misma región, así que no tendría sentido hacer un `shmdt` de un id.

El detach NO borra la memoria compartida. Eventualmente habría que devolver esas regiones al sistema. No hacerlo es peor que no hacer un delete, porque el segmento de memoria compartida depende del OS (no depende de ningún proceso).

Control y liberación de recursos

`int shmctl(int shmid, int cmd, struct shmid_ds *buf);`

Recibe un id, un comando y una struct. Para borrar le pasamos el comando de borrado `IPC_RMID`. El buf (struct) puede ser null. [Qué es? Para qué sirve?]

En bash hay dos comandos, `ipcs` e `ipcrm` que sirven para monitorear. `ipcs` lista los elementos de memoria compartida. `ipcrm -M key` elmina un recurso sabiendo su clave.

Biblio: Bach y man.

OJO: recordar que ahora que empezamos a compartir memoria. Va a haber que empezar a sincronizar. Podríamos hacer algo con fork y wait, pero sería, como dijimos antes, muy rudimentario.

Ejemplos y Ejercicios

Ejemplos: tiene un objeto MemoriaCompartida para encapsular las cosas de system V. El template permite manejar el tamaño.

static_cast. C++11. Buscar.

En este ejemplo en particular, cuando se usa el "liberar" se chequea que quede alguien usando el recurso y si no, se borra el segmento. Esto vale para este ejemplo, pero no necesariamente para cualquier programa.

Dato de color: System D barre las cosas extra que quedaron. Se pueden buscar.

Patrón RAII: se obtienen recursos en construcción de los objetos y se liberan en la destrucción. No hace falta llamar al liberar. El scope se encarga de los recursos.

Ejercicio posible para casa: primer ejemplo, pero que el que escriba es el hijo, y el padre recibe. Se elimina la necesidad del sleep (leo después del wait).

Ejercicio en clase: padre lanza hijos, los hijos duermen una cantidad aleatoria de segundos y el padre tiene que esperar a que termine cada uno. Pero ahora no hay que pasar el resultado al padre por el exit, sino utilizando memoria compartida.

Erlang es un lenguaje diseñado para concurrencia.

Ojo, el exit finaliza scope, termina el proceso. Si hay RAII no se liberan recursos.

Ojo, el shmat se conserva en fork. Eso tiene sentido, ya que es igual que con los file descriptors que se heredan en fork. El fork se encarga de sumar 1 al contador de atacheados y toda la bola.