Laboratorio 2: Semáforos

Introduccion

En este laboratorio se implementaran Semáforos para espacio de usuario, que son un mecanismo de sincronización interproceso. Se implementarán en espacio de kernel y deberán proveer syscalls para poder utilizarlos desde el espacio de usuario.

Como los semáforos vienen en varios estilos, en este laboratorio vamos a implementar solo un estilo de semáforos llamado semáforos nombrados inspiradonos en los semáforos nombrados que define POSIX.

Caracteristicas de los semáforos nombrados:

- Son administrados por el kernel.
- Estan disponibles globalmente para todos los procesos del SO (ie, no hay semáforos "privados").
- Su estado se preserva mientras el SO este activo (ie, se pierden entre reinicios).
- Cada semáforo tiene un "nombre" que lo identifica con el kernel, en nuestro caso los nombres son números enteros entre 0 y un limite máximo (en idea similar a los file descriptors).

Enunciado

Se deberá:

1. Implementar 4 syscalls: `sem_init`, `sem_up`, `sem_down`, `sem_release`.
2. Experimentar su implementación de semáforos con los programas de usuario: `barrier_init`,`barrier_echo` y `barrier_rise`.
3. Implementar un programa de espacio de usuario `pingpong` que solo funciona correctamente cuando los semáforos están implementados.
4. Hacer un informe sobre lo hecho.

Las syscalls:

- **`int sem_init(int semáphore, int value)`**: Abre y/o inicializa el semáforo con nombre `semáphore`. Si`value` es un número positivo (o cero) inicializa el semáforo a `value`. Si `value` es un número negativo solo abre el semáforo sin inicializarlo. En caso de exito devuelve 0 y en caso de error -1.
- **`int sem_release(int semáphore)`**: Cierra el semáforo con nombre `semáphore`. En caso de exito devuelve 0 y en caso de error -1.
- **`int sem_up(int semáphore)`**: Incrementa (destraba) el semáforo con nombre `semáphore`. En caso de exito devuelve 0 y en caso de error -1.
- **`int sem_down(int semáphore)`**: Decrementa (traba) el semáforo con nombre `semáphore`. En caso de exito devuelve 0 y en caso de error -1.

Para todas las syscalls el valor de `semáphore` es un entero entre 0 y un número máximo a definir por ustedes. Este número máximo debería existir como una constante simbólica y estar disponible al espacio de usuario en un `.h`.

Para implementar las syscalls deberan usar `acquire`, `release`, `wakeup`, `sleep` y `argint`. Es parte del laboratorio que investiguen y aprendan sobre estas funciones del kernel (Ver las ayudas al final).

También es parte del laboratorio que:

- Lo que implementen este libre de problemas de concurrencia y condiciones de carrera.
- Hagan validación de los argumentos de sus syscalls.

Programas `barrier_*`

Son tres programas que sirven para experimentar con semáforos. Implementan la idea de una barrera que detiene la ejecución de un print hasta que se le permita pasar.

Con `barrier_init` se crea el semáforo, con `barrier_echo` se *encola* un print y quedará detenido hasta que con`barrier_rise` le permita la ejecución. Un ejemplo:

```
$ barrier_init
$ barrier_echo hola mundo &
$ barrier_rise
hola mundo
$
```

Un ejemplo más involucrado:

```
$ barrier_init
$ barrier_echo pavarotti &
$ barrier_echo and &
$ barrier_echo friends &
$ mkdir pepito
$ ls pepito
.              1 23 32
..             1 1 512
$ barrier_rise
pavarotti
$ barrier_rise
and
$ barrier_rise
friends
$ barrier_rise
$ barrier_echo linger &
linger
$
```

El código fuente de estos programas se encuentra en la página de la materia. Para comprender exactamente lo que hacen deberán leerlos, son extramadamente cortos.

Deberan replicar estas ejecuciones de ejemplo y puede que la salida no sea exactamente la misma porque encuentren "`$`" y "`zombie!`" escrito varias veces. Esto es normal, y como parte del laboratorio deberán investigar porqué ocurre y explicarlo en el informe.

El programa `pingpong`

En síntesis deberan escribir un programa de usuario que sincroniza la escritura por pantalla de "`ping`" y"`pong`" usando semáforos.

El programa `pingpong` deberá hacer `fork` y con los dos procesos resultantes:

- Uno debera imprimir "`ping`" N veces, pero nunca imprimir dos "`ping`" seguidos sin que haya un"`pong`" al medio.
- El otro debera imprimir "`pong`" N veces, pero nunca imprimir dos "`pong`" seguidos sin que haya un"`ping`" al medio.

Es decir, deberá haber 2N lineas con `"ping"` y `"pong"` intercalados. Esta sincronización entre procesos es posible de realizar usando semáforos, y así deberán hacerlo.

La ejecución de este programa no debería producir la advertencia `"zombie!"` por parte del kernel.

El informe

El informe debe presentarse en un archivo `README.md` el directorio raiz en formato markdown e incluir:

- Una **muy breve** explicación de lo que hicieron (no repetir el enunciado).
- Una explicacion de porqué ocurre "`$`" y "`zombie!`" en la ejecución de los programas barrier.
- Una explicación de qué hacen `acquire`, `release`, `sleep`, `wakeup` y `argint`, esta explicación debe ser más completa que la que aparece en la ayuda.

Ayudas

- Como el "nombre" de un semáforo es un número entero entre 0 y N se pueden poner los semáforos en un arreglo dentro de espacio de kernel.
- Mini explicación:
  - `acquire`: Toma un lock haciendo busy waiting hasta que este disponible, similar al *down* de un semáforo. Util para crear zonas de exlusión mutua (mutex).
  - `release`: Libera un lock tomado, similar a un *up* de un semáforo.
  - `sleep`: Pone a "dormir" el proceso que realizo la syscall hasta un wakeup.
  - `wakeup`: Permite al scheduler volver a ejecutar un proceso que hizo una llamada a sleep.
  - `argint`: Permite leer un argumento dado a la llamada de una syscall.
- Implementar `sem_down` es la parte más desafiante y requiere una buena comprensión de `sleep` y `wakeup`. Pueden buscar ideas e inspiración en la implementación de `pipewrite` y `piperead` en `pipe.c`.
- Si `barrier_echo.c` es difil de entender compararlo con `echo.c`.
- Para escribir en markdown este visor online es util.

Entrega

- Deberán entregar via commits+push a su grupo en bitbucket.
- Deberán crear un repositorio git nuevo `Lab2` dentro del grupo `so2014gXX` con un directorio `xv6` dentro sobre el cual deberán hacer sus modificaciones. No copiar el Lab1, comenzar en limpio.
- El *coding style* deberá respetar a rajatabla las convenciones de `xv6`.