Laboratorio 2: Semáforos en XV6 Programando nuestras propias llamadas al sistema

Sistemas Operativos 2023 - FaMAF - UNC

Original 2014: Rafael Carrascosa

Versión 2017: Matías David Lee

Versión 2018: Ignacio Moretti

Versión 2022: Facundo Bustos

Versión 2023: Matias Cuenca

Introducción

En este laboratorio se implementará un sistema de Semáforos para espacio de usuario, que sirven como mecanismo de sincronización entre procesos. Se implementarán en la versión RISC-V de XV6 (sistema operativo con fines académicos) en espacio de kernel y deberá proveer syscalls accesibles desde espacio de usuario.

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

Características de los semáforos nombrados:

● Son administrados por el kernel.

● Están disponibles globalmente para todos los procesos del sistema operativo (i.e. no hay semáforos "privados").

● Su estado se preserva mientras el SO esté activo (i.e., 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 límite máximo (idea similar a los file descriptors).

Su trabajo

1. Implementar 4 syscalls: sem\_open(), sem\_up(), sem\_down(),sem\_close(). 2. Implementar un programa de espacio de usuario “pingpong” que funcione de la “manera natural”.

Las Syscalls

int sem\_open(int sem, int value) → Abre y/o inicializa el semáforo “sem” con un valor arbitrario “value”.

int sem\_close(int sem) →Libera el semáforo “sem”.

int sem\_up(int sem) →Incrementa el semáforo ”sem” desbloqueando los procesos cuando su valor es 0.

int sem\_down(int sem) →Decrementa el semáforo ”sem” bloqueando los procesos cuando su valor es 0. El valor del semaforo nunca puede ser menor a 0

Para todas las syscalls el valor “sem” es un entero entre 0 y un número máximo a definir por ustedes. Cada una de las funciones anteriores **devuelven 0 en caso de error**. Analizar y definir cuales son los posibles errores para cada una de ellas. Posiblemente surjan preguntas como “que sucede si ejecutamos sem\_close() de un semáforo que aún está siendo utilizado” o “que sucede si ejecutamos un sem\_up() de un semáforo cuando su valor es igual al valor de inicialización”. Ustedes deben decidir qué se hace en estos casos, definiendo así las políticas de su sistema de semáforos.

Para implementar las syscalls deberán 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 esté libre de problemas de concurrencia y condiciones de carrera.

● Hagan validación de los argumentos de sus syscalls.

El programa “pingpong”

Deberán escribir un programa de usuario que sincroniza la escritura por pantalla de la cadena "ping" y "pong" usando nuestro sistema de semáforos.

El comando “pingpong” deberá tomar como argumento un entero N (rally) que será la cantidad de veces que aparecerá la palabra "ping" y la palabra "pong" por pantalla.

El programa pingpong deberá hacer **un solo fork** y con los dos procesos resultantes:

● Uno deberá imprimir "ping" N veces, pero nunca imprimir dos "ping" seguidos sin que haya un "pong" al medio.

● El otro deberá imprimir "pong" N veces, pero nunca imprimir dos "pong" seguidos sin que haya un "ping" al medio.

● La secuencia resultante siempre debe empezar con “ping”. Por ejemplo

$ pingpong 1

ping

pong

● En caso de que haya un error, el comando el mensaje de error debe empezar con ERROR en mayúsculas. por ejemplo Todo error tiene que arrancar con ERROR

$ pingpong -1

ERROR: El numero de rounds tiene que ser mayor a 1

● Se debe retornar el prompt de consola solo cuando el juego haya terminado por completo (obviamente siempre que el comando pingpong se haya ejecutado en modo foreground).

Es decir, deberá haber 2N líneas con "ping" y "pong" intercalados. Esta sincronización entre procesos es posible de realizar usando semáforos, y así deberán hacerlo. Al terminar la ejecución del programa, los semáforos deberán estar liberados para nuevos usos.

Entrega

● Antes de entregar correr make grade para “auto corregir” el código y luego make clean para borrar todos los artefactos

● La entrega es via “commits+push” en bitbucket al repositorio asignado al grupo ● El \*coding style\* deberá respetar las convenciones de XV6.

Instalación de XV6 y puesta en marcha 1. Crear directorio para el lab2

mkdir lab2;cd lab2

2. Clonar repositorio de xv6 en el commit f5b93ef:

git clone https://github.com/mit-pdos/xv6-riscv.git

(cd xv6-riscv; git reset --hard f5b93ef)

3. Copiar los archivos clonados al repo de Bitbucket asignado a su grupo.

git clone https://user@bitbucket.org/sistop-famaf/so23lab2gXX.git cp -a xv6-riscv/\* so23lab2gXX

rm -rf xv6-riscv

4. Bajar archivo kickstarter y descomprimir en lab2 . Luego aplicar el patch (cd so23lab2gXX; patch -t -p1 < ../xv6-riscv-semaphore.patch) 5. Hacer el primer commit+push del proyecto. ← not working

cd so23lab2gXX

git tag -a original\_code -m "Solo el codigo de la catedra" git add \*

git commit -m 'first commit'

git push origin; git push origin --tags

6. Instalar qemu

sudo apt-get install qemu-system-riscv64 gcc-riscv64-linux-gnu

7. Compilar e iniciar xv6: posicionarse en el directorio so23lab2gXX (debe haber un archivo “Makefile”) y ejecutar el comando “make qemu”.

8. *Opcional: En vez de instalar y correr qemu en el SO local se puede correr en un container usando esta imagen pre-definida que ya contiene qemu y todas las dependencias necesarias para compilar XV6. Para instanciar un shell dentro del contenedor hay que correr:*

*docker build --tag 'xv6-env-so' - < Dockerfile.xv6 # Correr esto una sola vez*

*docker run -it --rm -v $(pwd):/home/xv6/xv6-riscv xv6-env-so*

Comandos y shortcuts útiles de XV6

● ls lista todos los programas de usuario disponibles del sistema.

● <CRTL-p>: Lista todos los procesos que están corriendo en el sistema. ● <CRTL-a> x: Sale del sistema operativo.

Ayudas

● Como el "id" de un semáforo es un número entero entre 0 y N, entonces se pueden implementar los semáforos como un arreglo dentro del espacio de kernel.

● Mini explicación de las syscalls que utilizarán:

● acquire(): Toma un lock haciendo busy waiting hasta que esté disponible, similar al down de un semáforo. Útil para crear zonas de exclusión mutua (mutex).

● release(): Libera un lock tomado, similar a un up de un semáforo. ● sleep(): Pone a "dormir" el proceso que realizó 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 en la llamada de la 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.

Bibliografía

● “xv6: a simple, Unix-like teaching operating system”

○ https://pdos.csail.mit.edu/6.828/2022/xv6/book-riscv-rev3.pdf ● Operating Systems: Three Easy Pieces

○ Chapter 31: Semaphores