PARCIALITO 2 Y 3 - S.O.

LAB 2 //

LOCKS

Resuelven el problema de ejecutar series de instrucciones atómicas y de tener múltiples hilos ejecutándose en múltiples procesadores.

Evitan el acceso a ciertas operaciones por parte de más de un hilo de ejecución.

Un lock es una variable (de tipo lock) que sostiene el estado de dicho lock

- disponible, desbloqueado, libre — available, unlocked, free
- adquirido, bloqueado, sostenido — acquire, locked, held

solo un hilo puede adquirir el lock

POSIX ← lock() / acquire() : → ptr. / → xv6

intentará adquirir dicho lock, si este no está heldeado por otro hilo, entonces el hilo actual adquirirá el lock y ejecutará el código a continuación

Si dicho lock sí está heldeado, entonces el hilo actual se quedará esperando hasta que este se libere para poder ejecutar el código a continuación.

Deshabilita las interrupciones //

POSIX xv6

unlock() / release(): libera el lock. Si hay hilos esperando, alguno de ellos adquirirá dicho lock.

initlock(ptr, lockname): inicializa el lock

holding(ptr): Chequea si el lock está siendo sostenido por el core actual

¿Cuándo deberíamos usar locks?

Zonas críticas

- Cuando una variable puede ser escrita por una CPU al mismo tiempo que otra CPU puede leerla o también escribirla.
- Cuando es importante mantener/proteger las invariantes (condiciones que deben cumplirse a lo largo de todo el código)

Tipos de locks en xv6

Spinlocks:

```
struct spinlock {
  uint locked;   // 1 -> no disponible, 0 -> disponible
  char *name;    // debugging
  struct cpu *cpu;
};
```

Si un hilo quiere adquirir un spinlock cuando este no está libre, se queda corriendo en un loop hasta que dicho spinlock esté disponible.

Sleep-locks: Bloquea/duerme al proceso que quiere adquirir el lock cuando este no está libre.

Condition variables

Colas en las que los mismos hilos deben ponerse cuando cierta condición no se cumple.
Estos hilos enviados serán despertados cuando el estado de algún otro hilo cambie, haciendo que se cumpla dicha condición.

wait()
Llamada cuando un hilo quiere ponerse a sí mismo a dormir. Libera el lock y sleepea el hilo. Debe re-adquirir el lock cuando llegue la señal que lo despierte.

signal()
Llamada cuando algún hilo cambia de estado y quiere enviar esa señal al hilo esperando dicho cambio.
El hilo que recibe la señal pasa a ready (runnable en xv6), pero no siempre comienza a correr automáticamente.

se despierta

En xv6 esto se logra mediante **sleep()** y **wakeup()**

Otra forma de lockear, cuando el lock requiere ser sostenido por mucho tiempo.

Semaphores

Objeto con un valor entero el cual puede manipularse con dos rutinas sem_wait() y sem_post().
El comportamiento del semáforo depende de este valor, por lo que debemos inicializarlo.

sem_wait() / sem_down():
decrementa el valor del semáforo y retorna si este es mayor a cero. Sino, suspende la ejecución del hilo que lo llamó hasta que el valor sea mayor a cero.

sem_post() / sem_up():
incrementa el valor del semáforo cuando este es cero. Si hay algún/os hilo/s esperando, es decir, algún hilo que haya llamado a sem_down() con el valor cero, despierta alguno.

Sem_open()
inicializa el semáforo con un valor arbitrario

sem_close()
libera el semáforo

0 si hubo error
1 si se inicializó correctamente

```
int sem_open(int sem, int value)
```

"nombre" del semáforo a inicializar

valor de inicialización

TODAS las funciones mencionadas anteriormente tienen el mismo retorno ⇒ sobre qué semáforo tendrá efecto también se pasa así

Implementación de Syscalls

1. Implementar la syscall en, obvio, espacio kernel. Todas están implementadas en varios archivos (proc.c; file.c; sysfile.c; sysproc.c) y declaradas en syscall.h.
2. Agregar la declaración de la nueva syscall en syscall.h y asignarle un número (preferiblemente el que le suceda a la syscall anteriores)
3. Agregarla al array que mapea dicho número a la función que la maneja (syscall.c)
4. Agregarla en el archivo user.h, en espacio de usuario, para poder utilizarla.
5. Agregarla en makefile para compilarla con las demás.

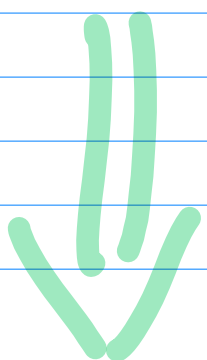
LAB 3

Scheduling

xv6 default scheduler: Round Robin
Ejecuta los procesos según una cola de procesos, en orden, estos se intercalan en cada trap generado por operaciones i/o ó timer interrupts.

MLFQ: los procesos se ejecutarán según una cola de prioridades. Mientras más prioridad tenga un proceso, antes se ejecutará.

- **Regla 1:** Prioridad A > Prioridad B => corre A
- **Regla 2:** Prioridad A = Prioridad B => corre el que fue elegido menos veces por el scheduler
- **Regla 3:** Un proceso nuevo tiene prioridad máxima
- **Regla 4:** Si el proceso agota su quantum => desciende su prioridad
  Si el proceso cede (a CPU antes de agotar su quantum => asciende de prioridad.



Implementación en xv6

- Definir niveles de prioridad como una constante (param.h)
- Agregar un atributo a las estructuras de los procesos que indique su prioridad y la cantidad de veces que fue elegido.
- Modificar el scheduler para que elija el proceso a correr según su prioridad y la cantidad de veces que fue elegido.

Funciones que interfieren en el scheduling

Llama a

yield() => se ejecuta cuando un proceso cede CPU debido a un timer interrupt, o sea, cuando deja de ser ejecutado.

Llama a

sched() => verifica condiciones necesarias para un cambio de contexto del proceso al scheduler exitoso

Llama a

swtch() => lleva a cabo todas las operaciones necesarias para los cambios de contexto entre procesos

Llama a

Scheduler() => encargado de elegir qué proceso se ejecutará a continuación.

Syscalls que interfieren en el scheduling

- **sleep()** => manda a dormir un proceso por cierta cantidad de ticks. Esta syscall llama a la función sleep() que se encarga de pasar un proceso de running a sleeping, liberar el lock y llamar a la función encargada del cambio de contexto al scheduler.

- **wakeup()** => despierta todos los procesos en estado sleeping del canal. No es una syscall en sí, sino una función del kernel

- **uptime()** => devuelve la cantidad de ticks que estuvo corriendo el SO desde su inicio.

- **exit()** => se encarga de marcar el proceso actual como zombie, enviándole una señal al proceso padre.
Si el proceso actual tiene hijos, estos se reasignan al proceso inicial de xv6.
Tmb cierra todos los archivos y libera directorios.

Conceptos relacionados con el scheduling

quantum => tiempo fijo que tiene un proceso para ejecutarse. Un proceso puede consumir o no todo su quantum. Una vez acabado/consumido se produce un timer interrupt. En xv6 está seteado en 1.000.000 de ciclos de CPU.

ticks => en xv6, estos aumentan luego de cada timer interrupt. Podemos decir que es la cantidad de interrupciones de temporizador que hubo desde el inicio del SO.

interrupciones => en xv6, una trap (interrupción) puede deberse a varias razones:

- **timer interrupt:** se generan cuando el temporizador se acaba (quantum)
- **device interrupts:** se generan cuando algún dispositivo de hardware completa una acción o solicita una acción. Por ejemplo el inicio o la finalización de una operación i/o las genera.
- **syscalls:** Cuando algún proceso llama a una systemcall se genera una interrupción.
- **faults/errores:** los errores en los programas generan interrupciones.

Estas interrupciones ceden el control al sistema operativo.

Si son generadas desde el espacio de usuario, se manejan desde usertrap(), sino, desde kerneltrap().

panic() => esta función es llamada cuando ocurre un error que causa que el sistema no pueda seguir ejecutando correctamente, por lo tanto, panic lo detiene, entrando en un loop infinito e indicando que el sistema se "paniqueó".