udp Escuela de Informática y Telecomunicaciones
FACULTAD DE INGENIERÍA

Sincronización (cont.)

Adín Ramírez
adin.ramirez@mail.udp.cl

Sistemas Operativos (CIT2003-1)
1er. Semestre 2015

Adaptado de Dave Eckhardt

Recapitulando

- Dos formas principales de construir programas con hilos
- Tres requisitos para secciones críticas
- Algoritmos que no se utilizan para secciones críticas

Sección crítica

- Protege una secuencia de instrucciones que deben ejecutarse atómicamente
  - Debemos de hacer algo para proteger la ejecución de las secuencias
  - Recordemos, el CPU cambia ejecución entre hilos (y procesos)
  - Un hilo ejecutandos en otro CPU (multi procesador)
- Suposiciones
  - La secuencia de instrucciones atómicas debe de ser pequeña
  - No hay otros hilos que compitan

Objetivos de la sección crítica

- Caso común (no hay competidores) debe de ser rápido
- El caso atípico (no común)
  - Puede ser lento
  - Pero! no debe de malgastar los recursos

Secuencias que interfieren

Cliente	Entrega
`cash = store->cash;`	`cash = store->cash;`
`cash += 50;`	`cash -= 20;`
`wallet -= 50;`	`wallet += 20;`
`store->cash = cash;`	`store->cash = cash;`

- ¿Qué secuencias interfieren unas con otras?
- Cliente interfiere con Entrega
- Y viceversa, Entrega interfiere con Cliente

Objetivos de la clase

- Soluciones (ahora sí) para el problema de secciones críticas
- Mutex
- Implementación
- Ambientes de ejecución

Mutex (Lock o Latch)

- **Mut**ual **ex**clusion
- Limita (y a su vez especifica) el código que interfiere a través de un objeto
  - Los datos están protegidos por el mutex
- Los métodos de los objetos encapsulan los protocolos de entrada y salida

```
mutex_lock(&store->lock);
cash = store->cash;
cash += 50;
personal_cash -= 50;
store->cash = cash;
mutex_unlock(&store->lock);
```

- ¿Qué hay dentro del objeto?

Exclusión mutua

Intercambio atómico

- En los Intel x86 usamos la instrucción `xchg`
- Por ejemplo `xchg (%esi),%edi`

```
int32 xchg(int32 *lock, int32 val){
  register int old;
  old = *lock; // bus is locked
  *lock = val; // bus is locked
  return (old);
}
```

Dentro del mutex

- Inicialización

```
int lock_available = 1;
```

- Intentar asegurar

```
i_won = xchg(&lock_available, 0);
```

- Spin-wait

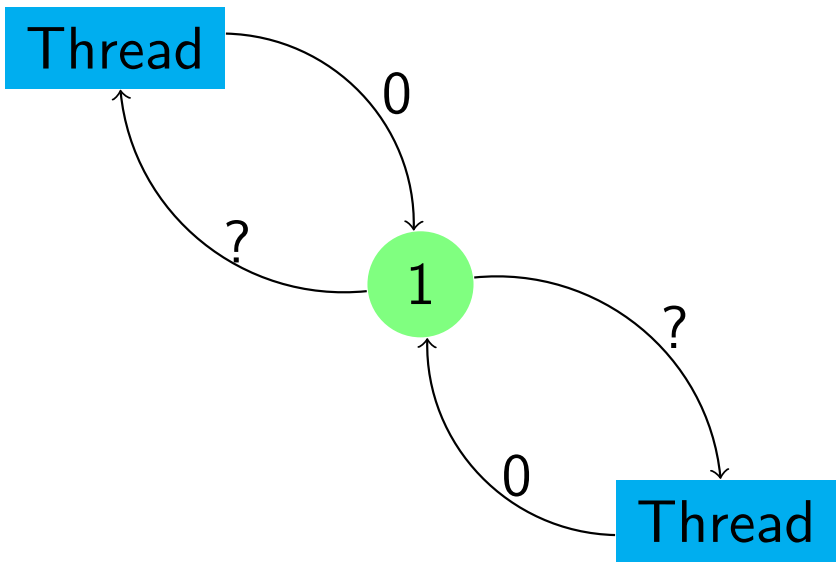
```
while(!xchg(&lock_available, 0))
  continue;
```

- Liberar el seguro (sección crítica)

```
xchg(&lock_available, 1);
```

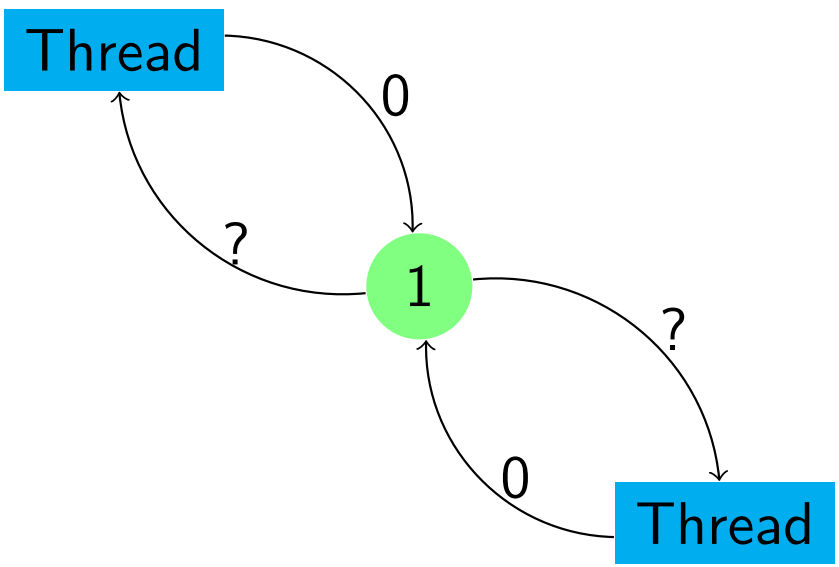
Dos hilos

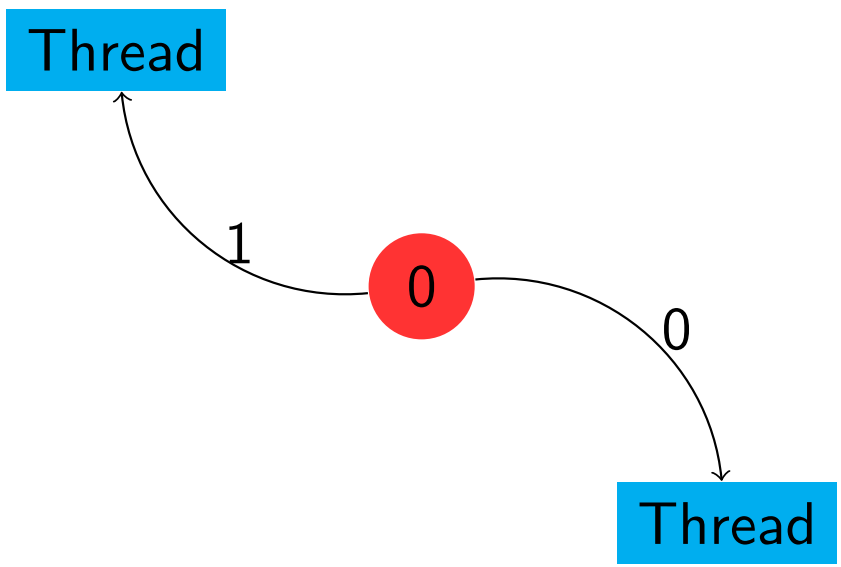
- Imaginemos dos hilos tomando un valor (`lock_available`), y dejando otro en el mismo lugar
- Simultáneamente



Dos hilos

- Imaginemos dos hilos tomando un valor (`lock_available`), y dejando otro en el mismo lugar
- Simultáneamente





¿Mantenemos las características de la sección crítica?

- Exclusión mutua
  - Existe solo un 1 (los 1 se conservan)
  - Solo un hilo puede ver el `lock_available == 1`

¿Mantenemos las características de la sección crítica?

- Exclusión mutua
  - Existe solo un 1 (los 1 se conservan)
  - Solo un hilo puede ver el `lock_available == 1`
- Progreso
  - Cuando `lock_available == 1` algún hilo lo obtendrá

¿Mantenemos las características de la sección crítica?

- Exclusión mutua
  - Existe solo un 1 (los 1 se conservan)
  - Solo un hilo puede ver el `lock_available == 1`
- Progreso
  - Cuando `lock_available == 1` algún hilo lo obtendrá
- Espera acotada
  - No
  - Un hilo puede perder (la toma del valor) arbitrariamente muchas veces

Asegurando la espera acotada

- Intuición
  - Muchos hilos pueden `xchg` simultáneamente
  - Necesitamos un sistema que permita "tomar turnos" (o que tenga un comportamiento similar)
- Posibles soluciones
  - Asegurarnos que cada adquisición de la exclusión en `xchg` tenga una salida *justa*
    - El como, no es necesariamente obvio
  - Y debemos agregar justicia a través del procedimiento de liberación del seguro
    - Alguien está a cargo (quien tiene la sección crítica)
    - Utilicemos eso a nuestro favor

Lock (asegurar)

```
waiting[i] = true; // declaremos interés en la sección
    crítica
got_it = false;
while (waiting[i] && !got_it) //"spin" mientras lo
   obtenemos
  got_it = xchg(&lock_available, false);
waiting[i] = false;
return; // obtuvimos la sección crítica, éxito!
```

Unlock (liberar)

```
j = (j + 1) % n; // recuerden que tenemos n hilos
while ( (j != i) && !waiting[j] ) // ?
  j = (j + 1) % n;
if (j == i) // ?
  xchg(&lock_available, true);
else
  waiting[j] = false;
return;
```

Posibles variaciones

- Intercambiar vs. probar y establecer (`TestAndSet`)
- El nombre de las variables que resguardan la sección crítica pueden cambiar (e.g., `available` en lugar de `locked`)
- Liberación atómica vs. escritura normal (donde hicimos `xchg` en la slide anterior)
  - Algunos hacen una escritura ciega `lock_available = true;`
  - Según la arquitectura, esto puede ser ilegal
  - El que está liberando debe necesitar utilizar acceso especial en la memoria (e.g., `Exchange`, `TestAndSet`, etc.)

Evaluación

- Un requerimiento extraño
  - Todos deben de saber la cantidad de hilos
    - Siempre, e instantáneamente
    - O utilizar una cota superior
- Un comportamiento desafortunado
  - Esperamos cero competidores (debemos correr rápido en forma normal)
  - El algoritmo debe de $O(n)$ en caso de tener máximo número de competidores
- ¿Muy dura nuestra evaluación?
  - El algoritmo de la panadería tenia estos mismos problemas
  - ¿Por qué nos preocupamos?

Veamos más alla de lo evidente

- Más allá de la semántica abstracta
  - Exclusión mutua, progreso, y espera acotada
- Consideremos
  - El patrón de acceso típico
  - Los ambientes de ejecución particulares
- Ambiente
  - Monoprocesador vs. multiprocesador
    - ¿Quién hace que cuando estamos tratando de asegurar/liberar?
  - Los hilos no están misteriosamente ejecutandose o no
    - La decisión de ejecutarlos se hace a través de un algoritmo calendarizador con ciertas propiedades

Ambiente monoprocesador

- Asegurar (lock)
  - ¿Qué pasa si `xchg()` no funcionó la primera vez?
  - Algún otro proceso tiene el seguro (lock)
    - Ese proceso no se está ejecutando (porque nosotros estamos usando el procesador)
    - Estar en el spin (el ciclo de espera) es una pérdida de tiempo
    - Nosotros debemos dejar al hilo que tiene la región crítica ejecutarse, en lugar de nosotros
- Liberar (unlock)
  - ¿Qué pasa con la espera acotada?
  - Cuando nosotros marcamos el mutex como disponible, ¿quién lo toma después?
    - Cualquiera que se ejecute después, solo uno a la vez (es una competencia falsa)
    - ¿Cuán injustos son los calendarizadores reales de los hilos del kernel?
    - Si el calendarizador es muy injusto, el hilo correcto nunca se ejecutará

Ambiente multiprocesador

- Asegurar (lock)
  - El esperar (spin) pude que esté justificado
    - ¿Por qué?
- Liberar (unlock)
  - El siguiente ganador de `xchg()` será escogido por el hardware de memoria
  - ¿Cuán injustos son los controladores de memoria?

Test and Set

```
boolean testandset(int32 *lock){
  register boolean old;
  old = *lock;
  *lock = true;
  return (old);
}
```

- Lo necesitamos en los ambientes multiprocesador
  - ¿Por qué?
- ¿Conceptualmente es más simple que `xchg`?
- Otras instrucciones de `x86`
  - `xadd`, `cmpxchg`, `cmpxchg8b`, ...
  - En la documentación de la arquitectura se detallan todas las posibilidades

Separemos la implementación

- Para ambientes multiprocesador
  - El realizar un lock en el bus es dañino
- Solución: dividimos `xchg` en dos partes
  - `load_linked(addr)` trae el valor antiguo de la memoria
  - `store_conditional(addr, val)` almacena el valor de vuelta
    - Si nadie está tratando de almacenar (o lo hizo) en esa dirección entre medio
    - Si alguien lo hizo la instrucción falla (establece un código de error)

Implementación

```
lock: LA   R1, mutex        ; &mutex in R1
loop: LL   R2, 0(R1)        ; mutex->avail
      BEQ  R2, R0, loop     ; avail == 0
      MOV  R3, R0           ; prepare 0
      SC   0(R1), R3        ; write 0?
      BEQ  R3, R0, loop     ; aborted
```

- Nuestro cache curiosea la memoria compartida
  - El asegurar la escritura de la variable no debe de apagar todo el tráfico en la memoria
  - El curiosear permite que el tráfico pase, y observa tráfico conflictivo
  - ¿Es correcto abortar? ¿Cuando es corecto?

`LA`: load address
`LL`: load linked word
`BEQ`: branch if equal

Intel `i860 lock bit`

- La instrucción coloca al procesador en modo asegurado (lock mode)
  - Lock al bus
  - Desactiva interrupciones
- ¿No es eso peligroso?
  - El timer emite una excepción
  - Cualquier excepción (page faults, divisiones por cero, etc.) libera el bus
- ¿Por qué queremos esto?
  - Implementar `test and set`, `compare and swap`, `semaphore` —su elección

Software misterioso para exclusión mutua

- Algoritmo de Lamport "fast mutual exclusion"
  - 5 escritura, 2 lecturas (si no hay contención)
  - No hay espera acotada (en teoría, si no hay contención)
  - `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-7.html`
- Interesantes algoritmos
  - Una solución para computadores "modernos"
  - Revisen el artículo, para otros puntos de vista

Y si alguién más se encarga

Q: ¿Por qué no pedirle al sistema operativo por una llamada de sistema `mutex_lock()`?

- Sencillo en monoprocesador
  - El kernel automáticamente excluye otros hilos
  - El kernel puede deshabilitar interrupciones fácilmente
  - No hay necesidad de ciclos no acotados, o extraños `xchg`
- El kernel tiene un poder especial en un ambiente multiprocesador
  - Puede enviar interrupciones remotas a otros CPUs
  - No hay necesidad de un ciclo no acotado
- Entonces, ¿por qué no dejar el trabajo en el sistema operativo?
  - Es demasiado caro

Software tramposo

- Exclusión mutua rápida para monoprocesadores
  - Bershad, Redell, Ellis: ASPLOS V (1992)
- Si queremos secuencias de instrucciones ininterrumpibles
  - Pretendemos
  - Monoprocesador: intercalar las instrucciones requiere intercambio de hilos
  - Una secuencia corta, la mayoría del tiempo, no será interrumpida
- ¿Cómo puede funcionar esto?

¿Cómo puede funcionar esto?

- El kernel detecta el cambio de contexto en la secuencia atómica
  - Tal vez un conjunto de instrucciones pequeño
  - Tal vez áreas particulares en la memoria
  - Tal vez una bandera `no_interruption_please = 1;`
- El kernel maneja el caso inusual
  - Entrega más tiempo (¿está bien?)
  - Simula que las instrucciones no han terminado
  - Una secuencia idempotente (realizar la misma secuencia para obtener el mismo resultado)

Puntos importantes

- Secuencia de instrucciones atómicas
  - Nadie puede intercalarse en la secuencia atómica
- Especificar las secuencias que interfieren (sección crítica) a través de un objeto `mutex`
- Dentro del mutex
  - Condiciones de carrera
  - Intercambio atómico, compare and swap, test and set, etc.
  - División en multiprocesadores (load-linked, store-conditional)
  - Los altibajos de este tipo de software
- Estrategia de los mutex
  - Como comportarse según el ambiente de ejecución