udp Escuela de Informática y Telecomunicaciones
FACULTAD DE INGENIERÍA

Sincronización (cont. 2)

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
  - **Ejecución corta de instrucciones atómicas**
  - Descalendarización voluntaria
- Tres requisitos para secciones críticas
  - Exclusión mutua
  - Progreso (respecto del algoritmo de exclusión)
  - Espera acotada
- Algoritmos que no se utilizan para secciones críticas
- Implementación de las secciones críticas con algoritmos que sí se utilizan
  - Mutex (locks)

Mutex

- Es una secuencia de instrucciones atómicas para establecer la región crítica
  - Nadie puede intercalarse en la secuencia atómica
  - Implementaciones a través del hardware (distinción entre mono y multiprocesador)
- 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

Objetivos de la clase

- Descalendarización voluntaria
- ¿Cómo? A través de variables de condición (condition variables)
  - Al igual que los mutex, vamos a ver que no existe la magia (como se implementan)
  - El problema para dormir de manera atómica
- Semáforos, monitores —visión general

Descalendarización voluntaria

- ¿Cuál es la situación?
  - Logramos pasar la entrada a la sección crítica, y tenemos el seguro (`lock`)
  - Pero, no necesariamente está en el modo correcto
  - Otros hilos pueden corregir el estado, pero ¿cómo pueden entrar si nosotros tenemos el seguro?
  - Recuerdan que hablamos de *deadlocks*
- Pasos a seguir
  - Debemos de liberar el recurso compartido
  - Y debemos asegurarnos que nos despierten al cumplirse la condición del recurso
  - Debemos bloquearnos hasta que el recurso esté en el estado correcto

¿Qué no hacer?

```
// aseguramos el recurso y lo liberamos en un ciclo
    infinito esperando el día del juicio
while (!reckoning) {
  mutex_lock(&lock);
  if ( date >= 2015-04-21 && hour >= 10 )
    reckoning = true;
  else
    mutex_unlock(&lock);
}
wreak_general_havoc();
mutex_unlock(&lock);
```

¿Qué problemas tiene esta solución?

- Ciclos y esperas innecesarias
  - No solo nuestras (nuestro hilo)
  - También bloqueamos otros hilos
- ¿En ambiente monoprocesador?
  - No tiene sentido
- ¿Y en multiprocesador?
  - Tal vez dependiendo de la solución
  - No es generalizable, ni escalable

Solución menos mala

```
// aseguramos el recurso y lo liberamos en un ciclo
    infinito esperando el día del juicio
while (!reckoning) {
  mutex_lock(&lock);
  if ( date >= 2015-04-21 && hour >= 10 )
    reckoning = true;
  else {
    mutex_unlock(&lock);
    sleep(1);
  }
}
wreak_general_havoc();
mutex_unlock(&lock);
```

Solución menos mala, aún no funciona

- ¿Por qué no utilizar `sleep(1)`?
  - En general $n - 1$ veces, no será suficiente
  - La n-ésima vez fue demasiado
  - Siempre nos equivocamos, ¿para qué usarla entonces?
- Pero, ¿cuál es el problema?
  - No queremos esperar por un tiempo determinado (porque no sabemos cuanto va a ser ese período)
  - Queremos esperar lo suficiente: cuando la condición cambie

Mención honorífica

```
// aseguramos el recurso y lo liberamos en un ciclo
    infinito esperando el día del juicio
while (!reckoning) {
  mutex_lock(&lock);
  if ( date >= 2015-04-21 && hour >= 10 )
    reckoning = true;
  else {
    mutex_unlock(&lock);
    yield(); // mejor que sleep, entregamos el ciclo
        de CPU (not enough!)
  }
}
wreak_general_havoc();
mutex_unlock(&lock);
```

Nos falta algo

- ☑ Resolvimos el estado protegido de los objetos compartidos
  - Utilizamos un objeto mutex (lock)
  - Encapsulamos el código que interfiere con los recursos
  - Pero, en algoritmos que realizan tareas complementarias tenemos problemas (deadlocks)
- ☐ ¿Cómo resolvemos el problema de bloquear un hilo por la duración exacta?
  - Obtengamos la información de quién sabe
  - Encapsulemos la "duración correcta"
  - Introduciendo: variables de condición

Intentemoslo una vez más

```
mutex_lock(&lock);
while ( cv = wait_on() ) { // acá está la magia
  cond_wait(&cv, &lock);
}
wreak_general_havoc(); // aún tenemos el seguro!
mutex_unlock(&lock);
```

wait_on()

```
if (year < 2015)
  return (&new_year);
else if (month < 4)
  return (&new_month);
else if (day < 21)
  return (&new_day);
else if (hour < 10)
  return (&new_hour);
else
  return (0); // acá terminamos
```

- Este código es conceptual
- Las implementaciones reales pueden variar (lo harán en realidad)

¿Qué es lo que nos despierta?

```
for (year = 1900; year < 3000; year++)
  for (month = 1; month <= 12; month++)
    for (day = 1; day <= days(month); day++)
      for (hour = 1; hour <= 24; hour++)
        // avanzamos las variables
        cond_broadcast(&new_hour);
      cond_broadcast(&new_day);
    cond_broadcast(&new_month);
  cond_broadcast(&new_year);
```

- Este código es conceptual
- En alguna parte en otro hilo ejecutamos el código que cambia las condiciones y nos despertará

Requerimientos de las variables de condición

- Mantener el rastro de los hilos que están bloqueados esperando que cierta condición cambie
- Permitir que los hilos que notifican puedan desbloquear a otros hilos
- Debe de ser seguro al ejecutarse entre hilos (garantizar la exclusión mutua)
  - Muchos hilos pueden llamar a la función `condition_wait()` simultáneamente
  - Muchos hilos pueden llamar a la función `condition_signal()` simultáneamente
  - Nuevamente, necesitamos atomicidad para resolver la atomicidad

Firma de las funciones

- `condition_wait(&cv, &mutex);`
- El `mutex` es usado para revisar el estado de la región crítica
  - Si encontramos un estado no asegurado sabemos que está en proceso de cambio
- Quien nos despierte necesitará el seguro para poder trabajar
  - Debemos entregar el `mutex` cuando dormimos
- A quién despertemos debemos de entregar el `mutex` de vuelta
  - Debe de ser conveniente que `condition_wait()` libere y re asegure la región crítica

Dentro de variable de condición

- `cv->queue`
  - Cola de todos los hilos bloqueados
  - FIFO, o algo más exótico (depende de la implementación)
- `cv->mutex` (`lock` o similares)
  - Protege la cola contra señales simultáneas (`wait()`/`signal()`)
  - No es el mutex de los hilos que llaman, es un mutex interno de la variable de condición que protege la cola
  - Está encapsulado dentro de la implementación

Implementación

```
condition_wait(cv, mutex){ // estos están definidos
    como punteros ya
  lock(cv->mutex);
  enqueue(cv->queue, my_thread_id());
  unlock(mutex); // este es el de fuera de VC
  ATOMICALLY {
    unlock(cv->mutex);
    kernel->pause(my_thread_id());
  }
  lock(mutex);
}
```

- ¿Qué es ATOMICALLY?
- Recordemos que sucedia con la implementación de los mutex

Que esperamos

cond_wait(cv, l)	cond_signal(cv)
enqueue(cv->queue, me);	
unlock(l);	
unlock(cv->lock);	
kernel->pause(me);	
lock(l);	
	lock(cv->lock);
	id = dequeue(cv->queue);
	kernel->wake(id);
	unlock(cv->lock)

Lo que puede suceder (ejecución con problemas)

cond_wait(cv, l)	cond_signal(cv)
`enqueue(cv->queue, me);`	
`unlock(l);`	
`unlock(cv->lock);`	
	`lock(cv->lock);`
	`id = dequeue(cv->queue);`
	`kernel->wake(id);`
	`unlock(cv->lock)`
`kernel->pause(me);`	
`lock(l);`	

- Error en el kernel `ERR_NOT_SLEEP`: `kernel->wake(id)`

Logrando atomicidad en `wait()`

- Reglas del juego
  - No existe una primitiva (en el hardware) para hacer `unlock_and_block()`
  - Tenemos el `unlock()`, y `block()`, y otras instrucciones
  - Afuera de `condition_wait()` y de `condition_signal()` debemos de parecer que estamos haciendo un `unlock` y `block` de manera atómica
- Soluciones
  - Deshabilitar interrupciones (si estamos en el kernel)
  - Dependemos en el sistema operativo para implementar variables de condición
    - ¿Es una buena idea?
  - Tener mejores interfaces de bloqueo de hilos
  - ¿Más mutexes?
  - Varias implementaciones
    http://www.cs.wustl.edu/~schmidt/win32-cv-1.html

Concepto de semáforos

- Un semáforo es otro objeto de encapsulamiento
  - Puede producir exclusión mutua
  - Puede producir el comportamiento de "bloquear hasta que sea el momento"
- Intuitivamente podemos pensar en el semáforo como un conteo de recursos
  - Un entero representa el número de recursos disponibles
    - Número de buffers, número de asientos disponibles, etc.
    - El semáforo se inicializa con algún número específico
  - El hilo se bloquea hasta que exista una instancia disponible

Operaciones del semáforo

- `wait()` o `P()`, en holandés *probeer te verlagen* o *tratar de decrementar*
  - Esperar hasta que el valor sea mayor que cero
  - Después decrementamos el valor (*tomamos* una instancia del recurso)
- `signal()` o `V()`, en holandés *verhogen* o *incrementar*
  - Incrementamos el valor del semáforo (*liberamos* la instancia del recurso)
- Y nuevamente `wait()` y `signal()` deben ser atómicas

Semáforo como un mutex

```
semaphore m = 1;

do {
   wait(m); // mutex_lock()
   // sección crítica
   signal(m); // mutex_unlock()
   // otras operaciones
} while (1);
```

Semáforo como una condición

Thread 0	Thread 1
	`wait(c);`
`result = 42;`	
`signal(c);`	
	`use(result);`

- Funciona como una variable de condición

Condición con memoria

- Los semáforos tienen memoria de los eventos que fueron avisados (`signal()`)
- Por ejemplo, un bit de vacío o lleno
- A diferencia de las variables de condición (sino están escuchando pierden las señales)

Thread 0	Thread 1
`result = 42;`	
`signal(c);`	
	`wait(c);`
	`use(result);`

Semáforo vs. Mutex–Condición

- El semáforo es una construcción de alto nivel
- Buenas noticias
  - Integra la exclusión mutua y la espera
  - Evita errores comunes en la API de los mutex–condición
    - Una `signal()` que llega antes del `wait()` se pierde
- Malas noticias
  - Integra la exclusión mutua y la espera
    - Algunos semáforos son como un mutex
    - Algunos semáforos son como una condición
    - ¿Cómo debe la librería saber que hacer? ¿Esperar o no?

Distintos tipos de semáforos

- Semáforo binario
  - Cuenta de 0 a 1 (disponible, no disponible)
  - Consideremoslo como un indicador al que lo implementa (un mutex)
- Semáforos que no se bloquean
  - `wait(semaphore, timeout);`
- Semáforos que evitan los deadlocks
  - Lo veremos en la clase de deadlocks

Evaluación del semáforo

- Puede parecer simple e intuitivo
  - Pero tiene demasiadas variaciones que mantener y entender
- Según Andrew Tanenbaum: “Lo bueno de los estándares es que tienes varios de donde escoger”
- Conceptualmente simple que tener dos objetos
  - Uno para exclusión mutua
  - Uno para esperar
  - Después de que entendimos que estaba sucediendo para garantizar la exclusión mutua

wait()

```
wait(semaphore s) {
  // Obtenemos acceso exclusivo
  --s->count;
  if (s->count < 0) {
    enqueue(s->queue, my_id());
    ATOMICALLY {
      // liberamos el acceso exclusivo
      thread_block();
    }
  } else
    // liberamos el acceso exclusivo
}
```

signal()

```
signal(semaphore s) {
  // Obtenemos acceso exclusivo
  ++s->count;
  if (s->count <= 0) {
    tid = dequeue(s->queue);
    thread_unblock(tid);
  } else
    // liberamos el acceso exclusivo
}
```

- ¿Qué es lo que tenemos acá?
- Un algoritmo de exclusión similar como un mutex, o
- Un sistema del sistema operativo de descalendarización (para despertar)

Concepto básico de monitor

- Problemas existentes
  - El semáforo elimina algunos errores de los mutex–condiciones
  - Sin embargo aún existen errores comunes
    - Intercambiar `signal()` y `wait()`
    - Accidentalmente omitir alguno
- Una abstracción de alto nivel: el monitor
  - Es un módulo de lenguajes de alto nivel
  - Todos accesan algunos recursos compartidos
  - El compilador agrega el código de sincronización
    - Piensen: Java
    - Un hilo que se ejecuta en un procedimiento bloquea la entrada de todos los otros hilos

Ejemplo de monitor en una tienda

```
int cash[STORES] = {0};
int wallet[CUSTOMERS] = {0};

boolean buy (int cust, int store, int price){
  if (wallet[cust] >= price) {
    cash[store] += price;
    wallet[cust] -= price;
    return (true);
  } else
    return (false);
}
```

Y la espera

- La exclusión mutua automática es buena, pero es demasiado fuerte
- A veces algún hilo necesita esperar por otro
  - La exclusión mutua automática no nos permite este comportamiento
  - Debemos de salir del monitor, y volver a entrar, ¿cuándo?
- ¿Les parece familiar el problema de determinar cuándo dormir y despertar?

Espera de los monitores

```
void cash_check(int acc, int chck) {
  while (account[acc].balance < check[chck].value) {
    // tengo que esperar, cuánto tiempo?
    // cómo era la solución? :S
  }
  account[acc].balance -= check[chck].value;
}
```

Solución incorrecta de espera de los monitores

```
boolean try_cash_check(int acc, int chck) {
  if (account[acc].balance < check[chck].value) {
    return (false); // no es mi problema, que alguién
        más se encarge
  }
  account[acc].balance -= check[chck].value;
  return (true);
}
```

Variables de condición de monitor

- Similares a las variables de condición anteriores
- `condition_wait(cv)`
  - Solamente un parámetro
  - El mutex que tenemos que compartir es implícito (dentro del monitor)
  - Operación
    - Salir temporalmente del monitor —liberamos el mutex
    - Esperamos hasta que nos envíen una señal
    - Re-entramos al monitor, y aseguramos el mutex

Espera del monitor con variables de condición

```
void cash_check(int acc, int chck) {
  while (account[acc].balance < check[chck].value)
    condition_wait(account[acc].activity);
  account[acc].balance -= check[chck].value;
}
```

- ¿Quién envía la señal a través de `signal()`?

Variables de condición del monitor

- `signal()`: políticas de ejecución
- ¿Quién se ejecuta después?
- El que llamó, o el llamado?
- ¿Qué pasa con el seguro del monitor? En la señal, salimos de él (efecto secundario)
- Diferentes políticas implican distintos tipos de monitor (problemas similares con los semáforos)

Puntos importantes

- Dos operaciones fundamentales
  - Exclusión mutua para secuencias que deben ser atómicas
  - Debemos descalendarizar atómicamente (y despertar después)
- Mutex–variables de condición (estilo de `pthreads` POSIX)
  - Dos objetos
  - Uno para cada operación
- Semáforos y monitores
  - Semáforo: un objeto
  - Monitor: objetos generados transparentemente por el compilador
  - Las mismas ideas están encapsuladas

¿Qué recordar?

- Deben de manejar
  - Los problemas y los objetivos de los distintos objetos
  - Las técnicas que están bajo ellos
  - ¿Cómo se involucra el diseño en la solución de problemas?
- Ya tenemos nuestras herramientas para poder sincronizar
  - Aún tenemos que resolver el problema de los deadlocks

Lecturas extra

- http://research.microsoft.com/pubs/64242/implementingcvs.pdf
- http://www.cs.cornell.edu/courses/cs4410/2012fa/papers/commandments.pdf