udp Escuela de Informática y Telecomunicaciones
FACULTAD DE INGENIERÍA

# Sincronización

Adín Ramírez
adin.ramirez@mail.udp.cl

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

Adaptado de Dave Eckhardt

# Recapitulando

- Capítulo 5 de OS:P+P
- Los hilos son como los procesos
- Distinta multiplicidad (programa vs. kernel)
- ¿Preguntas sobre condiciones de carrera?

# Objetivos de la clase

- Problemas reales
- Operaciones fundamentales
- Propiedades necesarias de la sección crítica
- Solución a dos procesos
- $N$ procesos: Algoritmo de la panadería

# Cuando el compilador es inteligente

- Nuestra computadoras son “inteligentes”, y eso crea problemas
- Imaginemos el código

```
choosing[i] = true;
number[i] = max(number[0], number[1], ...) + 1;
choosing[i] = false;
```

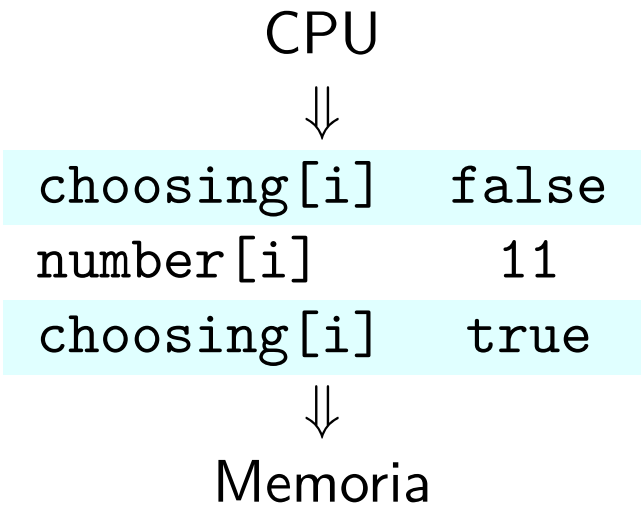
- Sin embargo, cuando revisamos lo que está en la memoria es

```
number[i] = 11;
choosing[i] = false;
```

- ¿Qué paso?

# ¿La computadora se equivocó?

- Nuestras computadoras son modernas: “inteligentes”
  - El procesador escribe comandos en colas
  - La memoria almacena esas colas
  - Pero, comandos redundantes son fusionados
- ¿Por qué?
  - Optimización
  - Algunas operaciones no necesitan que se realicen
  - ¿Qué pasa con las que sí necesitamos?



# Soluciones

- Barreras en la memoria
  - Instrucciones que pueden detener al procesador
  - Esperar que la lista de escritura este vacía
  - Magia (no!)
- Los modelos de memoria simples no funcionan en la presencia de múltiples procesadores
  - `http://www.cs.umd.edu/~pugh/java/memoryModel/`
  - `http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html`
- Para nuestras explicaciones un algoritmo de exclusión mutua y el modelo de memoria será simple
  - Lo que sucedía en los computadores pre-modernos
  - En el mundo real, no funciona tan bien
  - Entonces, tener en cuenta: compiladores, arquitectura, lenguaje, etc.

# Fundamentos de la sincronización

- Dos operaciones fundamentales
  - Secuencia de instrucciones atómicas
  - Descalendarización voluntaria
- Múltiples implementaciones de cada una
  - Procesador único vs. varios procesadores
  - Hardware especial vs. algoritmos especiales
  - Técnicas diferentes en los sistemas operativos
  - Ajustar el rendimiento en algunos casos especiales
- Las características en cada uno son claras
  - Las operaciones son más diferentes
  - Son opuestas, más que similares
- Las abstracciones en los clientes utilizan las dos operaciones
- Nuestra literatura prefiere la sección crítica (semáforos, monitores)
- Relevante
  - Variables mutex (mutual exclusion) y de condición (POSIX threads)
  - Java `synchronized` (3 formas)

# Operaciones fundamentales

- Secuencia de instrucciones atómicas
- Descalendarización voluntaria

# Secuencia de instrucciones atómicas

- Dominio del problema
  - Es una secuencia corta de instrucciones
  - Nadie más puede intercalar la misma secuencia
  - O una secuencia similar
  - Típicamente no existe competencia

# Ejemplos de no interferencia

- Simulación en multiprocesador (imaginemos Sim City, Civilization)
  - Grano grueso de cada turno (e.g., por hora)
  - Mucha actividad en cada turno
  - Imaginemos $M : N$ hilos, $M$ objetos, $N$ procesadores
- La mayoría de autos no interactúan en un juego por turnos
  - Debemos modelar aquellos que sí lo hacen
  - Una intersección no puede ser procesada por varios automóviles al mismo tiempo

# Comercio

| Cliente 0 | Cliente 1 |
|---|---|
| `cash = store->cash;` | `cash = store->cash;` |
| `cash += 50;` | `cash += 20;` |
| `wallet -= 50;` | `wallet -= 20;` |
| `store->cash = cash;` | `store->cash = cash;` |

- ¿Qué pasa con la ejecución?
- ¿Puede fallar?
- ¿Qué puede hacer la tienda?

# Observaciones del problema de comercio

- El conjunto de instrucciones es pequeño
  - Está bien el excluir mutuamente a los competidores
  - Podemos hacerlos esperar
- La probabilidad de colisión es baja
  - Muchas invocaciones que no colisionan (imaginemos muchas tiendas utilizando el sistema)
  - No debemos usar un mecanismo caro para evitar la colisión
  - El caso común (de no colisión) debe de ser rápido

# Descalendarización voluntaria

- Dominio del problema
  - Esperar voluntariamente
  - ¿Ya llegamos?
- Ejemplo: el demonio del desastre en Sim City

```
while (date < 2015-04-07)
  cwait(date);
while (hour < 5)
  cwait(hour);
for (i=0; i<max_x; i++)
  for (j=0; j<max_y; j++)
    wreak_havok(i,j);
```

# Propiedades

- Cooperativa
- Anti atómica
  - Queremos estar intercalados lo más posible
- Ejecutarme y hacer que otros esperen es malo
  - Malo para ellos: no estaremos listos en mucho tiempo
  - Malo para nosotros: no podemos estar listos si ellos no están listos
- No queremos exclusión
- Queremos que otros se ejecuten, ellos nos ayudan
- La descalendarización desde el CPU es un servicio del sistema operativo

# Patron de espera

- Nosotros

```
LOCK WORLD
while ( !(ready = scan_world()) ) {
  UNLOCK WORLD
  WAIT_FOR(progress_event)
  LOCK WORLD
}
```

- Nuestros compañeros

```
SIGNAL(progress_event)
```

# Un poco más técnico

```
do {
   entrada a la seccion
   sección crítica:
     computación en el espacio compartido
   fin de la sección crítica
   resto de la sección:
     computación privada
} while(1);
```

# Nomenclatura estándar

- ¿Qué apagamos en el código anterior?
  - ¿Qué está en la sección crítica?
  - Una secuencia atómica rápida
  - Necesitamos dormir por un largo tiempo
- Por ahora
  - Asumamos que la sección crítica es una secuencia atómica corta
  - Estudiemos la entrada y salida de la sección crítica

# Requerimientos de la sección crítica

- Exclusión mutua
  - Cuando mucho, un hilo se está ejecutando en la sección crítica
- Progreso
  - El protocolo escogido debe de tener un tiempo acotado de trabajo
  - Una forma de fallar: para escoger el siguiente candidato necesitamos que existan candidatos (un ejemplo a continuación)
- Espera acotada
  - No podemos esperar por siempre una vez que entramos a la sección crítica
  - La entrada es acotada por los otros participantes
  - No necesariamente acotada por un número de instrucciones

# Notación para 2 procesos

- Suposiciones
  - Múltiples hilos (1 CPU con un timer, o múltiples CPU's)
  - Memoria compartida, pero no tenemos locks, o instrucciones atómicas
- Hilo $i$ somos nosotros
- Hilo $j$ otro hilo
- $i$, $j$ son variables locales de hilos

# Primera idea: turnarnos

```
int turn = 0;
while (turn != i)
  continue;
// sección crítica
turn = j;
```

- Exclusión mutua: si (¿la ven?)
- Progreso:no
  - El tomar turnos estrictos es fatal
  - Si $T_i$ nunca intenta entrar a la sección crítica, $T_1$ espera para siempre
  - Viola la regla de “depender en que no existan participantes”

# Otra idea: mostrar interés

```
boolean want[2] = {false, false};

want[i] = true;
while (want[j])
  continue;
// sección crítica
want[i] = false;
```

- Al inicio nadie esta interesado en la región crítica
- Si queremos entrar a la región crítica, mostramos interés
- Exclusión mutua: sí
- Progreso: casi (¿por qué?)

# Intuición de exclusión mutua

| Hilo 0 | Hilo 1 |
|---|---|
| `want[0] = true;` | |
| `while(want[1]);` | |
| entramos SC | |
| | `want[1] = true;` |
| | `while(want[0]);` |
| | `while(want[0]);` |
| | `while(want[0]);` |
| | `while(want[0]);` |
| `want[0] = false;` | |
| | `while(want[0]);` |
| | entramos SC |

# Otra ejecución

| Hilo 0 | Hilo 1 |
|---|---|
| `want[0] = true;` | |
| | `want[1] = true;` |
| `while(want[1]);` | |
| | `while(want[0]);` |

- Falla en el progreso de la ejecución
- Funciona para cualquier otra forma de intercalar las instrucciones
- Resultado: sad panda, and waiting panda

# Sollución de Peterson (1981)

- Tomar turnos cuando sea necesario

```
boolean want[2] = {false, false};
int turn = 0;

want=[i] = true;
turn = j;
while (want[j] && turn == j)
  continue;
// sección crítica
want[i] = false;
```

- ¿Por qué `turn = j` y no `turn = i`?

# Prueba de la exclusión

- Asumamos lo contrario: dos hilos en la sección crítica
- Ambos en la sección crítica implica que `want[i] == want[j] == true`
- Entonces ambos ciclos terminaron porque `turn != j`
- No podemos tener `turn == 0 && turn == 1`
  - Entonces alguno termino antes
- Sin perder generalidad, $T_0$ termino primero porque `turn == 1` falló
  - Entonces `turn == 0` antes que `turn == 1`
  - Entonces $T_1$ tiene que establecer `turn = 0` antes que $T_0$ establezca `turn = 1`
  - Entonces $T_0$ no puede ver `turn == 0`, no puede salir del ciclo antes

# Pistas de la prueba

- `want[i] == want[j] == true`
  - `want[]` no significa nada, entonces nos enfocamos en `turn`
- `turn[]` versus quien salió del ciclo
  - ¿Qué pasa si intercalamos de distintas maneras?

| Hilo 0 | Hilo 1 |
|---|---|
| `turn = 1;` | `turn = 0;` |
| `while(turn == 1);` | `while(turn == 0);` |

# Contexto

- Si hay más de dos procesos
  - Generalización basada en el mostrador de la panadería
  - Obtenemos números de tickets que aumentan monotónicamente
  - Esperamos hasta que el número de cliente (que aumenta monotónicamente) sea el nuestro
- Versión multiproceso
  - A diferencia de la realidad, dos personas (procesos) pueden obtener el mismo número de ticket
  - Ordenamos por el número de ticket, con alguna forma de romper empates
    - Número de ticket, y número de proceso

# Algoritmo

- Fase 1: escoger un número
  - Obtener el número de procesos/clientes
  - Agregar 1 al número
- Fase 2: esperar a que llegue tu turno
  - No necesariamente cierto, varios procesos pueden tener el mismo número
  - Utilizamos el número del proceso, y el ticket para seleccionar
    - (ticket 7, proceso 99) ¡ (ticket 7, proceso 101)
- Tomamos turno cuando tenemos el ticket y el proceso menor

# Algoritmo

- Fase 0: contexto

```
boolean choosing[n] = {false, ...};
int number[n] = {0, ...};
```

- Fase 1: escoger un número

```
choosing[i] = true;
number[i] = max(number[0], number[1], ...) + 1;
choosin[i] = false;
```

- En el peor de los casos todos escogen el mismo número
- Pero en la siguiente ronda, todos escogen uno mayor

# Algoritmo

- Fase 2: escoger el menor número

```
for(j=0; j<n; j++) {
  while(choosing[j])
    continue;
  while( (number[j]!=0) &&
         ( (number[i], i) > (number[j],j) )
       )
    continue;
}
// sección crítica
number[i] = 0;
```

# Puntos importantes

- La memoria en condiciones de carrera puede ser compleja
- Debemos entender dos operaciones fundamentales:
  - Ejecución breve para operaciones atómicas
  - Debemos ceder nuestra ejecución a largo plazo para poder obtener resultados
- Tres condiciones necesarias para una sección crítica
  - Exclusión mutua
  - Progreso en la ejecución
  - Espera debe ser acotada
- Debemos entender los algoritmos básicos de exclusión
  - La solución para dos procesos
  - $N$ procesos: la panadería

# Práctica

- Para entender la exclusión mutua hay que practicar
- El practicar cimienta los conceptos en aprendizaje
- No es lo mismo leer el algoritmo, a entenderlo y poder escribirlo
- Ejercicio:
  - Dos hilos
  - Comparten una variable `monto`
  - Cada hilo recibe un parámetro `cantidad`
  - Un hilo (el padre) toma la `cantidad` y la suma a `monto`
  - El otro hilo (el hijo) toma la `cantidad` y la resta a `monto`
  - ¿Cómo se implementa esta interacción para que compartan `monto`?