6: Sincronización

Sistemas Operativos 1
Ing. Alejandro León Liu

- Sincronización de Threads o Procesos
  - Aplica para ambos

- Threads independientes
  - No comparten estado (data)
  - No importa el orden de la calendarización
  - Reproducible: Volver a recrear el escenario
  - Determinístico: Input determina el resultado
  - Ideal: pero son poco probable
    - Threads siempre comparten sistema, file system, recursos, i/o
- Threads cooperativas
  - Comparten estado (data)
  - No reproducible
  - No determinístico
  - BUGS: Heisenbugs :S

- Threads cooperantes pueden dar problemas, por qué usarlas?
  - Compartir recursos
  - Mejorar throughput
    - Overlap I/O con CPU
  - Modularidad

T1 y T2 comparten balance

- T1: balance+=100
  register1 = balance
  register1 = register1 + 100
  balance = register1

- T2: balance+=200
  register2 = balance
  register2 = register2 + 200
  balance = register2

- Si “balance = 100”:

T1: register1 = balance	{register1 = 100}
T1: register1 = register1 + 100	{register1 = 200}
T2: register2 = balance	{register2 = 100}
T2: register2 = register2 + 200	{register2 = 300}
T1: balance = register1	{balance = 200}
T2: balance = register2	{balance = 300}

- Race condition: varios procesos manipulando misma data. Orden de acceso determina resultado
- Ley de Murphy: "Anything that can go wrong will go wrong".

CRITICAL SECTION

- Segmento de código dónde se manipulan variables, tablas, archivos, etc…
- Dos threads no pueden ejecutar Critical Section al mismo tiempo
- Cada thread debe pedir permiso para entrar a Critical Section: entry section
- Exit section: notificar que se terminó de ejecutar Critical Section
- Remainder section: resto del código

- Solución al problema de Critical Section debe satisfacer:
  - Mutual exclusion
    - Si una thread puede ejecutar la C.S. a la vez.
  - Progress
    - Si nadie ejecuta C.S. y varias threads quieren entrar, la decisión de qué thread entra no puede postponerse
  - Bounded waiting
    - No hay starvation

- Preemptive Kernels
  - Ejecutar procesos en tiempo real. Minimizar tiempo de respuesta.
  - Difícil implementar, especialmente en SMP (Simmetric multiprocessing)
  - Linux 2.6+, Solaris
- Nonpreemptive kernels
  - Proceso ejecutándose en kernel mode no puede ser interrumpido
  - Libre de race conditions en el kernel ya que solo uno se ejecuta a la vez
  - Windows, UNIX
  - Sti y cli en ASM Intel 8086.

SINCRONIZACIÓN EN LA VIDA REAL

- Se acabó la leche!

Hora	Persona A	Persona B
3:00	Ver refrigerador: no hay leche	
3:05	Ir a la tienda	
3:10	Llega a la tienda	Ver refrigerador: no hay leche
3:15	Comprar leche	Ir a la tienda
3:20	Llegar a casa. Colocar leche	Llega a la tienda
3:25		Comprar leche
3:30		Llegar a casa. Colocar leche

- ……. Ahora tenemos mucha leche!

- Problema: leche
  - Solo uno compra leche
  - Si no hay leche, alguien compra leche
- Toda sincronización requiere espera

- Solución #1

```
1:      if (noMilk) {
2:              if (noNote) {
3:                      put Note
4:                      buyMilk
5:                      remove Note
6:              }
7:      }
```

- Funciona?
  - Casi siempre. (Heisenbug)
  - Ejemplo: A hace contextSwitch antes de 3.

- Solución #2

```
1:    put note
2:    if (noMilk) {
3:            if (noNote) {
4:                    put Note
5:                    buyMilk
6:            }
7:    }
8:    remove Note
```

- Funciona?
  - Nadie compra leche

- Solución #3

	Persona A	Persona B
1:	put note A	put note B
2:	if (noNote B) {	if (noNote A)
2:	if (noMilk) {	if (noMilk) {
4:	buyMilk	buyMilk
6:	}	}
7:	}	}
8:	remove note A	remove note B

- Funciona?
  - A ejecuta 1.
  - Context switch
  - B ejecuta 1.
  - Nadie compra leche
  - Improbable, pero puede pasar

- Solución #4

	Persona A	Persona B
1:	put note A	put note B
2:	while (Note B) {	if (noNote A)
3:	do nothing	if (noMilk) {
4:	}	buyMilk
5:	if (noMilk) {	}
6:	buyMilk	}
7:	}	remove note B
8:	remove note A	

- Funciona?
  - Si

- Solución complicada
  - Compleja
  - Código diferente. ¿Qué pasa si hay más personas?
  - A está busy waiting
    - Consume CPU

SOLUCIÓN DE PETERSON

- Solución en software
- Compartir los siguientes datos:
  - Int turn
  - Bool flag[2]
- Si Pi quiere ejecutar C.S.
  - Establece flag[i]
  - Turn = j
- Si ambos procesos quieren ejecutar C.S., ambos establecerán *turn*
  - Turn solamente conservará un valor
  - Este valor determina qué proceso ejecuta C.S.

```
Proceso i
do {
        // entry section
        flag[i] = true
        turn = j
        while (flag[j] && turn == j)
                do nothing;

        // critical section
        // …

        // exit section
        flag[i] = false

        // remainder section
        // …
} while (1)
```

HARDWARE DE SINCRONIZACIÓN

- Deshabilitar interrupciones
  - No funciona en ambiente multiprocesador
  - Utilizado por nonpreemptive kernels
  - Afectar el reloj y el sistema en general
- Hardware proveen instrucciones atómicas (no interrumpidas) de sincronización
  - TestAndSet
  - Swap

- TestAndSet
  - Atómica
  - Establece la bandera y devuelve el valor que tenía target.

```
boolean TestAndSet (boolean *target)
{
   boolean rv = *target;
   *target = TRUE;
   return rv:
}
```

- Variable compartida lock inicializada en false

```
while (true) {
   while ( TestAndSet (&lock ))
         ;   /* do nothing

   //    critical section

   lock = FALSE;

   //      remainder section
}
```

- Swap
  - Atómica
  - Intercambia el contenido de dos variables

```
void Swap (boolean *a, boolean *b)
{
   boolean temp = *a;
   *a = *b;
   *b = temp:
}
```

- Variable compartida lock. Variable key por cada thread

```
while (true)  {
   key = TRUE;
   while ( key == TRUE)
         Swap (&lock, &key );

   //    critical section

   lock = FALSE;

   //      remainder section
}
```

- TestAndSet y Swap resuelven el problema de C.S.
  - Las soluciones anteriores no cumplen la condición de bounded waiting
    - TestAndSet y Swap son ejecutadas por una thread a la vez, pero puede ser cualquier thread.
  - Existe solución más compleja que provee bounded waiting
  - Complejo para desarrolladores

SEMÁFOROS

- Variable que puede ser accedida por los métodos wait y signal. (Excepto al inicializar)
  - Atómicos
- Dos tipos
  - Counting semaphore
    - Proveer exclusión mutua entre n procesos
  - Binary semaphore (0 y 1) o mutex lock
    - Proveer acceso a recurso con n instancias
  - Valor de semaphore indica instancias disponibles
    - Nunca puede ser negativo

- wait (S) {
  while S <= 0
  ; // no-op
  S--;
  }
- signal (S) {
  S++;
  }

```
While(true) {
   wait (mutex)
   // critical section
   signal(mutex)
   // remainder section
}
```

- Sincronizar dos procesos de forma que se ejecute S2 (en thread 2) si y solo si S1 (en thread 1) se ha ejecutado
  - Inicializar sync en 0
  - En Thread1

    ….

    S1

    Signal(sync)

    ….
  - En Thread2

    ….

    Wait(sync)

    S2

    ….

- Definiciones anteriores
  - Busy waiting
  - Costo de oportunidad (Perdemos oportunidad de ejecutar proceso que haga más que while)
  - Spinlock
- Implementación sin busy waiting
  - Semaphore compuesto por valor entero y cola de procesos.

```
wait (S){
        value--;
        if (value < 0) {
                Agregar este proceso a waiting queue
                block();
        }
}

Signal (S){
        value++;
        if (value <= 0) {
                Remover proceso P de la waiting queue
                wakeup(P);
        }
}
```

PROBLEMA PRODUCTOR-CONSUMIDOR

- Recordando…
  - Bounded buffer
  - Productor

```
while (true) {
        /* Produce an item */
       while (((in = (in + 1) % BUFFER SIZE count)  == out) // FULL
          ;   /* do nothing -- no free buffers */
         buffer[in] = item;
         in = (in + 1) % BUFFER SIZE;
    }
```

  - Consumidor

```
while (true) {
         while (in == out) // EMPTY
               ; // do nothing -- nothing to consume

          // remove an item from the buffer
          item = buffer[out];
          out = (out + 1) % BUFFER SIZE;
      return item;
 }
```

- Solución con Semaphores
  - Semaphore mutex inicializado en 1: Sincronización a buffer
  - Semaphore full inicializado en 0
  - Semaphore empty inicializado en N. (tamaño del buffer)
- Productor

```
while (true)  {

        //   produce an item

        wait (empty);
        wait (mutex);

        //  add the item to the  buffer

        signal (mutex);
        signal (full);
    }
```

- Consumidor

```
while (true) {
        wait (full);
        wait (mutex);

            //  remove an item from  buffer

        signal (mutex);
        signal (empty);

            //  consume the removed item

    }
```

- Solución mucho más sencilla
- Solución simétrica

READERS – WRITERS PROBLEM

- Por ejemplo lecturas y escrituras a bases de datos.
- Es posible realizar varias lecturas simultaneas
- Si una thread escribe, nadie puede leer ni escribir

- Solución
  - Int readcount=0: número de threads leyendo
  - Semaphore mutex=1: semaphore para acceder a readcount
  - Semaphore wrt=1: exclusión para escribir

- Writer

```
while (true) {
        wait (wrt) ;

        //    writing is performed

        signal (wrt) ;
}
```

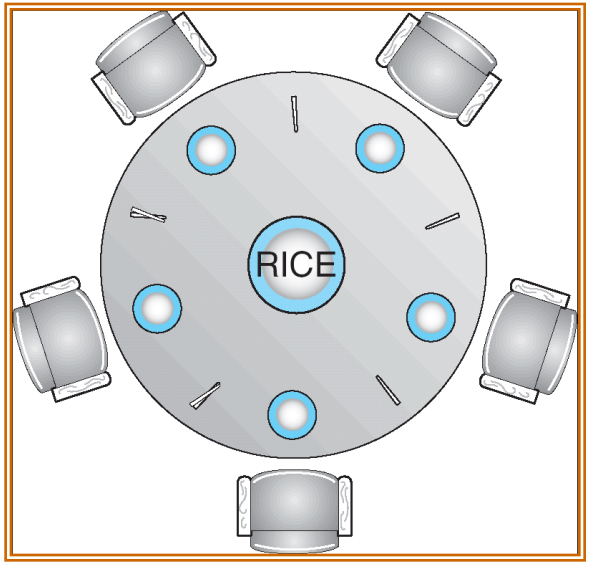
- Reader

```
while (true) {
        wait (mutex) ;
        readcount ++ ;
        if (readercount == 1)
                wait (wrt) ;
        signal (mutex)

        // reading is performed

        wait (mutex) ;
        readcount  - - ;
        if (redacount  == 0)
                signal (wrt) ;
        signal (mutex) ;
}
```

DINING PHILOSOPHERS



- Philosopher i

Compartir semaphore chopstick[5]

```
While (true)  {

        // think

    wait ( chopstick[i] );
     wait ( chopStick[ (i + 1) % 5] );

        //  eat

     signal ( chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );

}
```

- ¿Qué pasa si los cinco les da hambre al mismo tiempo?
  - Todos agarran el chopsick izquierdo y esperan a que el derecho se libere
  - Semaphores pueden producir Deadlock

MONITORS

- Herramienta de sincronización de alto nivel
- Solamente un proceso puede invocar métodos del monitor a la vez
- Variables de tipo condition
  - Wait y signal
- Dining philosophers:

dp.pickup (i)

EAT

dp.putdown (i)

```
monitor DP
   {
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];

    void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING) self [i].wait;
    }

       void putdown (int i) {
          state[i] = THINKING;
                  // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
        }
```

```
    void test (int i) {
            if ( (state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
                state[i] = EATING ;
              self[i].signal () ;
             }
     }

        initialization_code() {
           for (int i = 0; i < 5; i++)
           state[i] = THINKING;
    }
}
```

SINCRONIZACIÓN EN JAVA

- java.lang.object
  - notify() : Wakes up a single thread that is waiting on this object's monitor.
  - notifyAll() : Wakes up all threads that are waiting on this object's monitor.
  - wait() : Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
  - wait(long timeout) : Causes current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
  - wait(long timeout, int nanos) : Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

- Synchronized
  - Métodos

```
public synchronized void increment()
{
    c++;
}
```

  - Statements

```
public void addName(String name)
{
    synchronized(this)
    {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

- Acceso atómico a variables
  - Reads y writes atómicos para variables por referencia (ref)
  - Reads y writes atómicos para variables declarados con la palabra reservada volatile

```
volatile int counter;
```

- Interfaz lock
  - Posee lista de conditions con wait y notify
  - Trylock: Prueba si tiene acceso inmediato o después de timer.
  - Varias implementaciones
    - ReentrantLock: Similar a un semaphore
    - ReadLock
    - WriteLock

- http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html

TRANSACCIONES

- Colección de instrucciones que realizan una misma función lógica
- Garantiza que las operaciones se realizan como una unidad lógica de trabajo
  - Completa: commit
  - Fallida: rollback
    - abort
    - Falla en el sistema

- Relacionado a base de datos
  - Obtener y actualizar id

```
select id from table
update table set id = id + 1
```

- Transacciones financieras
  - Reserva/bloqueo de fondos
  - Realización de operación
    - Exitosa: confirmar fondos
    - Fallida: desbloquear fondos

- Dispositivos de almacenamiento
  - Volatil: No se preserva después de un error del sistema
    - Memoria principal
    - cache
  - No volatil: Usualmente sobrevive un error del sistema
    - Discos
    - Discos magnéticos
  - Estable: información nunca se pierde (teóricamente)
    - Construir aproximación a partir de replicar información en medios no volátiles. Distintos modos de fallos.
    - Raid
- File system
  - Corrupción de tabla de directorio
  - Corrupción de archivos

- Log based recovery
  - Almacenar en medio no volatil todas las modificaciones realizadas
  - Write ahead log
    - Start: escribir en log al iniciar transacción
    - Commit: escribir en log al terminar transacción
    - Modificaciones
      - Nombre de transacción
      - Data item o variable modificada
      - Valor anterior
      - Valor nuevo
  - Después de falla, consultar log completo
    - Tiempo de búsqueda costoso
    - Muchas transacciones tienen data que luego ha sido actualizada
    - Transacciones que son anuladas (rollback)

- Checkpoints
  - Mantener un write ahead log
  - Almacenar registros de medio volatil a medio no volatil
  - Almacenar registros de medio no volatil a medio estable
  - Almacenar registro de checkpoint en log
  - Recuperación
    - Obtener registros de medio estable
    - Ya no es necesario rehacer transacciones previas al último checkpoint

- Concurrencia de transacciones atómicas
  - Múltiples transacciones ejecutadas simultáneamente.
  - Resultado de ejecutar simultáneamente o separado debe ser igual.
    - Ejecutar transacciones dentro de Critical Sections
    - Buscar calendarizaciones que no tengan conflictos (Diferente resultado a ejecución serial)

- Resumen
  - Problema de Sección Crítica
  - Soluciones por Hardware
    - Acceso exclusivo a memoria
    - Deshabilitar interrupciones
    - TestAndSet
    - Swap
  - Soluciones por Software
    - Peterson´s Algorithm
    - Semaphores
    - Monitors