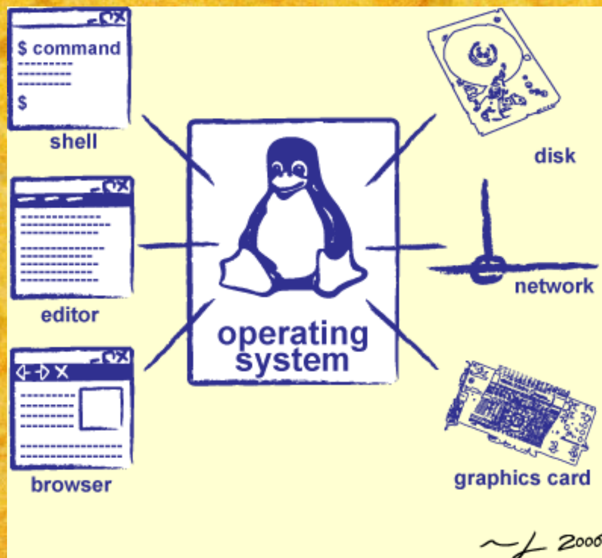
6.- Sincronización de Procesos



Sincronización de Procesos

- Introducción
- El problema de la sección crítica
- La solución de Peterson
- Hardware de Sincronización
- Semáforos
- Problemas clásicos
- Monitores
- Ejemplos de Sincronización
- Transacciones Atómicas

Introducción

- El acceso concurrente a datos compartidos puede producir datos inconsistentes
- Para mantener la consistencia de datos se requiere mecanismos que aseguren la ejecución ordenada de procesos cooperativos
- Se requiere una solución al problema consumidor-productor con buffers limitados.
- Se implementa mediante un entero count que cuenta los buffers usados. Initialmente, count = 0.
- El productor lo incrementa cuando produce un nuevo buffer y el consumidor lo decrementa cuando consume un buffer.

Productor

```
while (true) {
    /*  produce an item and put in nextProduced  */
    while (count == BUFFER_SIZE) ; // do nothing
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
}
```

Consumidor

```
while (true)  {
        while (count == 0); // do nothing
              nextConsumed =  buffer[out];
                  /*  consume the item in nextConsumed
               out = (out + 1) % BUFFER_SIZE;
              count--;
   }
```

Condición de Carrera (Race Condition)

- Compilación de count++

  register1 = count
  register1 = register1 + 1
  count = register1

- Compilación de count--

  register2 = count
  register2 = register2 - 1
  count = register2

- Ejecución Consider this execution alternada iniciando con “count = 5”:

  S0: productor ejecuta register1 = count {register1 = 5}
  S1: productor ejecuta register1 = register1 + 1 {register1 = 6}
  S2: consumidor ejecuta register2 = count {register2 = 5}
  S3: consumidor ejecuta register2 = register2 - 1 {register2 = 4}
  S4: productor ejecuta count = register1 {count = 6 }
  S5: consumidor ejecuta count = register2 {count = 4}

Solución al Problema de la Sección Crítica

1. Exclusión Mutua - Si el proceso P_i está ejecutando su sección crítica (SC), ningún otro proceso puede ejecutar su sección crítica
2. Progreso – Si ningún proceso está en su SC y existe algún proceso que desea entrar a su SC, la selección del proceso que entrará en su SC no puede ser postpuesto indefinidamente
3. Espera Acotada - Debe existir una cota en el número de veces que otros procesos entrar sus SCs después de que un proceso solicita entrar a su SC y después de que se le autoriza
   - Asumir que cada proceso se ejecuta a velocidad mayor que cero
   - No asumir nada en cuanto a la velocidad relativa de los N procesos

La Solución de Peterson

- Solución para dos procesos
- Asume que LOAD y STORE son instrucciones atómicas; i.e., no pueden ser interrumpidas.
- Los dos procesos comparten dos variables:
  - int turn;
  - Boolean flag[2]
- La variable turn indica a quien le toca entrar en la SC
- El arreglo flag se usa para indicar si un proceso se encuentra listo para entrar en la SC. flag[i] = true → P_i está listo

Algoritmo para el Proceso P_i

```
while (true) {

        SECCIÓN INICIAL

    flag[i] = TRUE;
    turn = j;
    while ( flag[j] && turn == j);

        SECCIÓN CRÍTICA

    flag[i] = FALSE;

        SECCIÓN FINAL

}
```

Hardware de Sincronización

- Muchos sistemas proporcionan soporte de HW para el código de la SC
- Uniprocesadores – pueden deshabilitar interrupciones
  - Ejecutar código sin desalojo
  - En general, muy ineficiente para sistemas multiprocesadores
    - Los SOs que lo usan no son escalables
- Las máquinas actuales proporcionan instrucciones especiales atómicas (Atómica = no interrumpible)
  - Probar dato y asignar valor (test and set)
  - Intercambiar dos datos (swap)

TestAndSet

- Definición:

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

Solución usando TestAndSet

- Variable lógica compartida: lock., initializada a falso.
- Solución:

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

        //    critical section

        lock = FALSE;

        //      remainder section
}
```

Instrucción Swap

- Definición:

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

Solución usando Swap

- Compartir variable lógica lock, inicializada a FALSE
- Cada proceso tiene una variable local lógica key.
- Solución:

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

         //    critical section

         lock = FALSE;

         //      remainder section
 }
```

Semáforos

- No requiere busy waiting
- Semaphore *S* – variable entera
- Operaciones estándar (modifican a S): wait() y signal()
  - ```
    wait (S) {
        while S <= 0  ; // no-op
        S--;
    }
    ```
  - ```
    signal (S) {
        S++;
    }
    ```
  - Originalmente se llamaban P() y V()

Semáforos

- semáforo contador – entero, valores sin restricción
- semáforo binario – valores 0 o 1 (+ simple)
  - Aka mutex locks
- Se puede implementar un semáforo contador S como un semáforo binario
- Proporciona exclusion mutua
  - Semaphore S;   // inicializado en 1
  - wait (S);

        Sección Crítica

    signal (S);

Implementación

- Debe garantizarse que ningún par de procesos ejecuten wait () y signal () en el mismo semáforo al mismo tiempo
- La implementación se convierte en el problema de la SC, donde el código de wait y signal son la SC
  - Podríamos usar busy waiting en la implementación de la SC
    - Código corto
    - No se usa mucho busy waiting si la SC no se usa frecuentemente
- Algunas aplicaciones pueden pasar mucho tiempo en sus SCs.
- No es una buena solución

Implementación sin Busy Wait

- A cada semáforo se le asocia una cola. Cada elemento en la cola tiene dos datos:
  - valor (entero)
  - apuntador al siguiente elemento
- Operaciones:
  - block – coloca al proceso que invoca la operación en la cola de espera del semáforo
  - wakeup – quita al proceso de la cola de espera del semáforo y lo pone en la cola de ejecución (ready queue)

Implementación sin Busy Wait

- wait (S){

```
        value--;
        if (value < 0) {
          add this process to waiting queue
          block();  }
  }
```

- Signal (S){

```
        value++;
        if (value <= 0) {
          remove a process P from the waiting queue
          wakeup(P);  }
  }
```

Deadlocks e Inanición

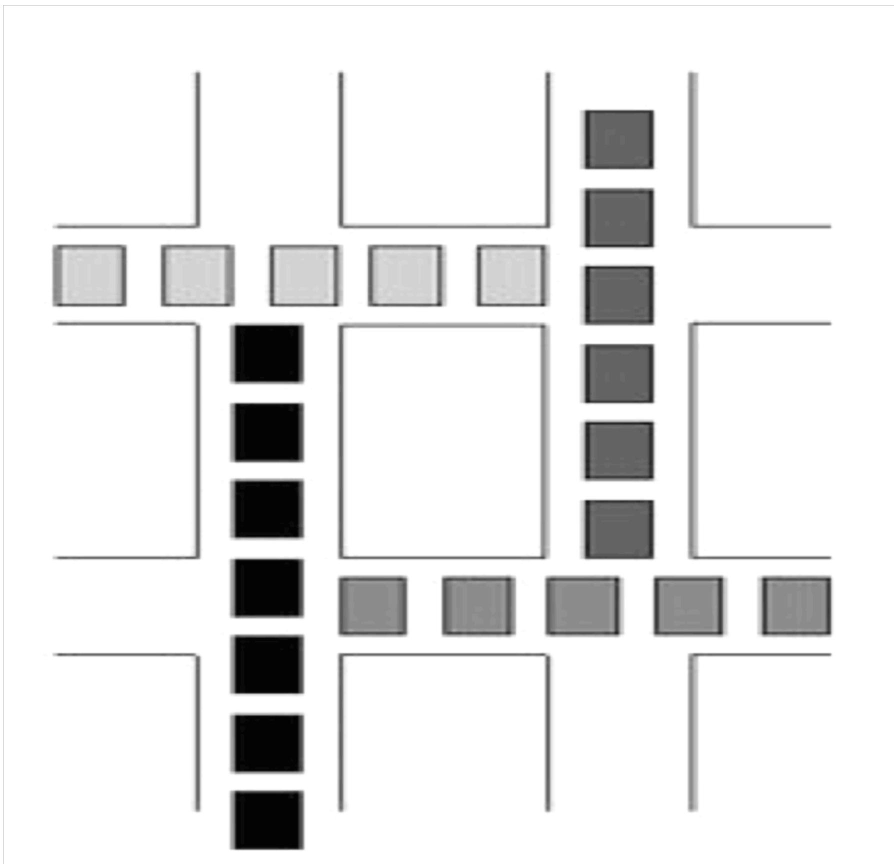
- Deadlock – 2+ procesos esperan indefinidamente por un evento que puede ser causado por uno de los procesos en espera
- S y Q - semáforos inicializados en 1

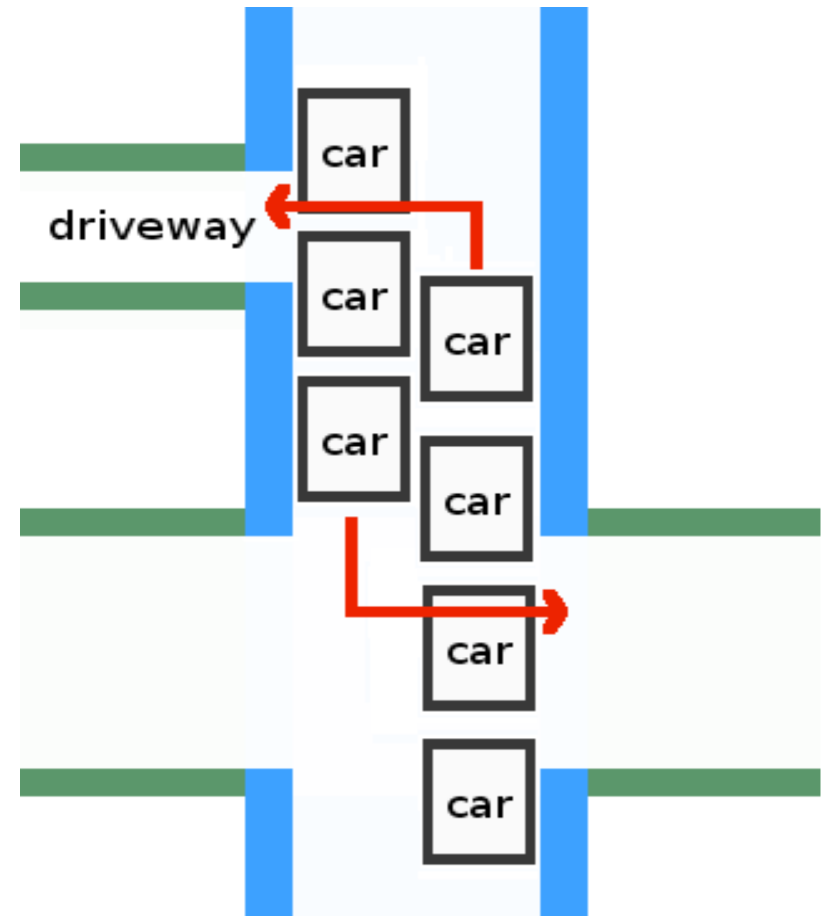
P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (Q);	signal (S);
signal (S);	signal (Q);

- Inanición (Starvation) – bloqueo indefinido. Un proceso nunca es removido de la cola de un semáforo, en la cual está suspendido.

Deadlocks e Inanición

Deadlocks e Inanición





Problemas Clásicos de Sincronización

- Problema del buffer limitado (Bounded-Buffer)
- Problema de los Lectores y Escritores
- Problema de los Fiósofos comensales

Bounded-Buffer

- *N* buffers, cada uno puede almacenar un dato
- Semaphore mutex inicializado en 1
- Semaphore full inicializado en 0
- Semaphore empty inicializado en N.

Bounded-Buffer (Cont.)

- Estructura del productor

```
while (true)  {

        //   produce un dato

    wait (empty);
    wait (mutex);

        //  añade el dato al buffer

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

Bounded-Buffer (Cont.)

- Estructura del consumidor

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

              //  elimina un dato del buffer

       signal (mutex);
       signal (empty);

              //  consume el dato eliminado

}
```

Readers-Writers

- Un dato se comparte por varios procesos concurrentes
  - Readers – solo leen los datos
  - Writers – pueden leer y escribir

- Problema – varios lectores pueden leer al mismo tiempo. Solo un escritor puede accesar el dato en un momento dado.

- Datos compartidos:
  - El dato
  - Semaphore mutex inicializado en 1.
  - Semaphore wrt inicializado en 1.
  - Integer readcount inicializado en 0.

Readers-Writers (Cont.)

- writer

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

            //    writing is performed

        signal (wrt) ;
}
```

Readers-Writers (Cont.)

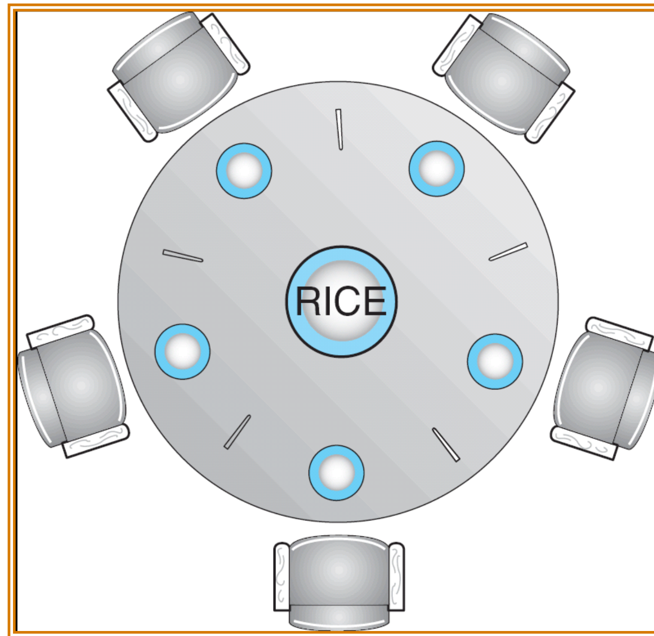
- reader

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

                // reading is performed

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

Filósofos Comensales



- Comparten:
  - Arroz (dato)
  - Semaphore chopstick [5] inicializado en 1

Filósofos Comensales (Cont.)

- Estructura del Filósofo *i*:

```
while (true)  {
        wait ( chopstick[i] );
         wait ( chopStick[ (i + 1) % 5] );
              //  eat
         signal ( chopstick[i] );
         signal (chopstick[ (i + 1) % 5] );
             //  think
}
```

Problemas con Semáforos

- Uso incorrecto de operaciones de semáforos:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitir wait (mutex) o signal (mutex) (o ambos)

Monitores

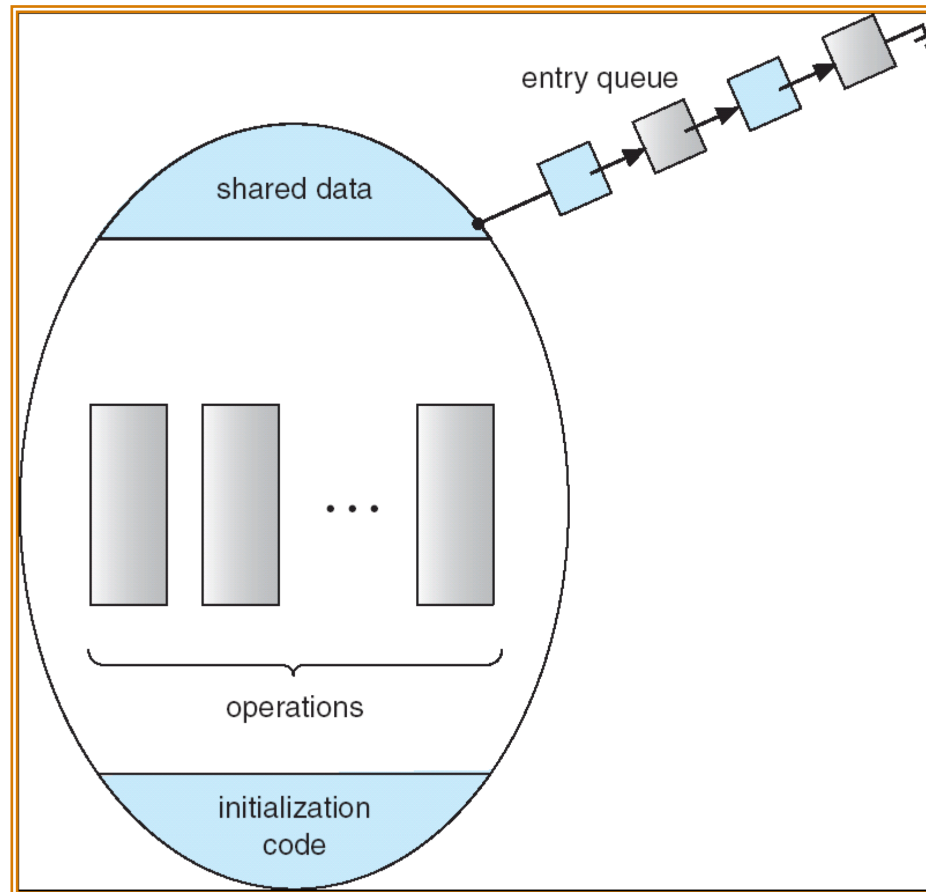
- Mayor abstración – Mecanismo efectivo y conveniente para sincronización de procesos
- Solo un proceso puede estar activo en un monitor al mismo tiempo

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (…) { …. }
     …

    procedure Pn (…) {……}

   Initialization code ( ….) { … }
     …
    }
}
```

Vista Esquemática de un Monitor

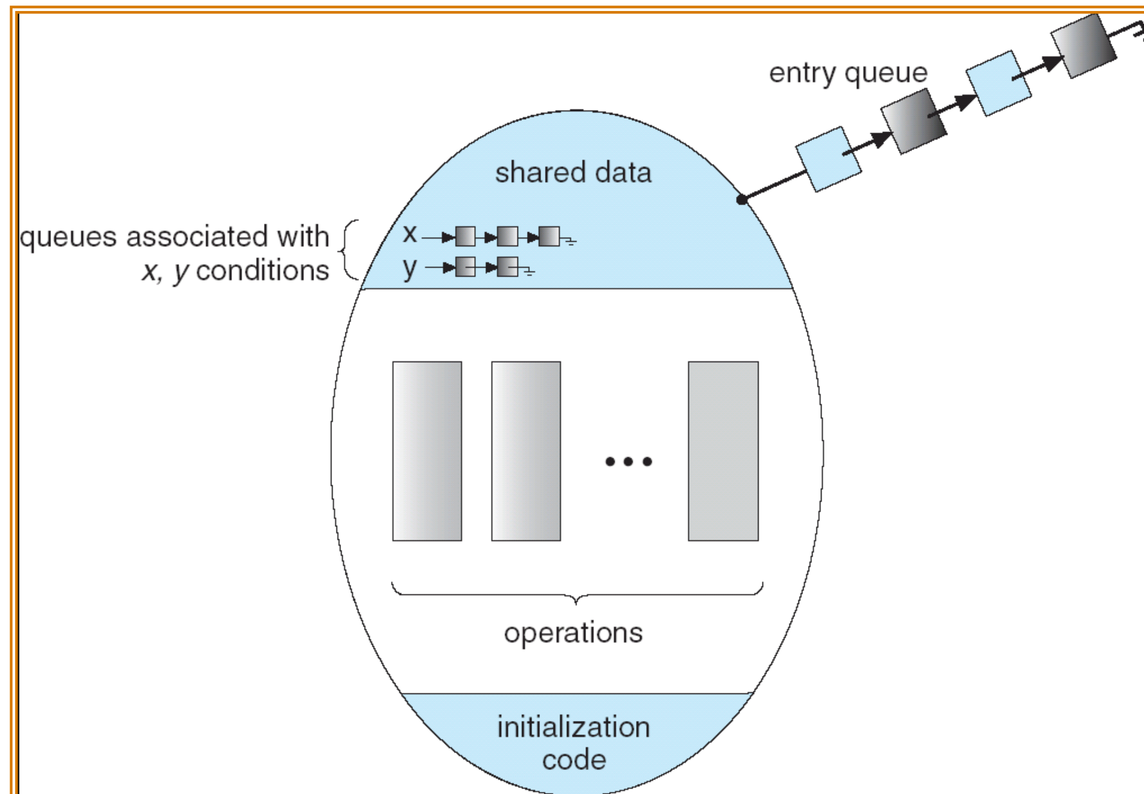


Variables de Condición

- condition x, y;

- Dos operaciones en variables de condición:
  - x.wait () – el proceso que invoca esta operación es suspendido
  - x.signal () – resume uno de los procesos (si existe) que invocó x.wait ()

Monitor con Variables de Condición



Solución a los Filósofos Comensales

```
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);
    }
```

Solución a los Filósofos Comensales

```
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;
}
}
```

Solución a los Filósofos Comensales

- Cada filósofo *i* invoca las operaciones pickup() y putdown() en la secuencia siguiente:

```
dp.pickup (i)

    EAT

dp.putdown (i)
```

Implementación de Monitores Usando Semáforos

- Variables

```
semaphore mutex;  // (valor incial = 1)
semaphore next;   // (valor incial = 0)
int next-count = 0;
```

- Cada procedimiento ***F*** será reemplazado por

```
wait(mutex);
   …
      body of F;
   …
if (next-count > 0)
  signal(next)
else
  signal(mutex);
```

- Asegura exclusión mutua en el monitor.

Implementación de Monitores Usando Semáforos

- Para cada variable de condición ***x***:

```
semaphore x-sem; // (valor incial = 0)
int x-count = 0;
```

- La operación x.wait puede ser implementada por:

```
x-count++;
if (next-count > 0)
      signal(next);
else
      signal(mutex);
wait(x-sem);
x-count--;
```

Implementación de Monitores Usando Semáforos

- La operación x.signal puede ser implementada por:

```
if (x-count > 0) {
    next-count++;
    signal(x-sem);
    wait(next);
    next-count--;
}
```

Ejemplos de Sincronización

- Solaris
- Windows XP
- Linux
- Pthreads

Sincronización en Solaris

- Implementa una variedad de candados (locks) para soportar multitasking, multithreading (incluyendo real-time threads), y multiprocesamiento
- Usa adaptive mutexes para mejorar la eficiencia cuando se protegen datos de segmentos de código cortos
- Usa variables de condición y candados readers-writers cuando secciones de código más largas necesitan acceso a los datos
- Usa turnstiles para ordenar la lista de threads en espera de tomar un adaptive mutex o un candado reader-writer

Sincronización en Windows XP

- Usa interrupt masks para proteger el acceso a recursos globales en sistemas uniprocesadores
- Usa spinlocks en sistemas multiprocesadores
- Proporciona dispatcher objects, los cuales pueden actuar como mutexes o semáforos
- Dispatcher objects pueden también producir events
  - Un evento actúa como una variable de condición

Sincronización en Linux

- Linux:
  - Deshabilita interrupciones para implementar secciones críticas cortas
- Linux proporciona:
  - Semáforos
  - spin locks

Sincronización con Pthreads

- El API de Pthreads API es independiente del SO
- Proporciona:
  - Candados mutex
  - Variables de condición
- Algunas extensiones no portables incluyen:
  - Candados read-write
  - Candados spin

Transacciones Atómicas

- Modelo del Sistema
- Recuperación por bitácoras (Log-based Recovery)
- Checkpoints
- Transacciones atómicas concurrentes

Modelo del Sistema

- Asegura que las operaciones sucedan como una sola unidad lógica de trabajo, o que no sucedan
- El reto es asegurar atomicidad, en presencia de fallas del sistema
- Relacionado con BDs
- Transaction – colección de instrucciones u operaciones que realizan una sola función lógica
  - Cambios a almacenamiento estable – disco
  - Transacción = serie de operaciones read y write
  - Terminadas por la operatción commit (OK) o abort (error)
  - Las transacciones abortadas deben ser rolled-back para deshacer todo cambio que haya sido realizado

Tipos de Almacenamiento

- Volátil – no sobrevive a system crashes
  - Ejemplo: memoria principal, cache
- No volátil – sobrevive a system crashes
  - Ejemplo: disco y cinta
- Estable – Nunca se pierde la información
  - No es posible aún. Aproximado mediante réplicas o RAID

Meta: asegurar atomicidad en transacciones, donde las fallas pueden causar pérdida de información en almacenamiento volátil

Recuperación por Bitácora (Log-Based)

- Grabar a un medio estable las modificaciones hechas por una transacción
- El más común es write-ahead logging
  - Log en almacenamiento estable, cada registro describe una operación write, incluyendo
    - Nombre de la transacción
    - Nombre del dato
    - Valor anterior
    - Valor nuevo
  - <T_i starts> escrito cuando T_i comienza
  - <T_i commits> escrito cuando T_i se realiza
- El registro debe ser grabado en el medio estable antes de que la operación ocurra.

Recuperación Log-Based Algoritmo

- Mediante la bitácora, el sistema maneja errores de memoria volátil
  - Undo(T_i) restablece los valores de los datos actualizados pot T_i
  - Redo(T_i) asigna valores de los datos en la transacción T_i a los nuevos valores
  - Undo(T_i) y redo(T_i) deben ser idempotentes
  - Ejecuciones múltiples deben tener el mismo resultado que una sola ejecución
- Si el sistema falla, restablece el estado de todos los datos actualizados (vía log)
  - Si el log contiene <T_i starts> sin <T_i commits>, undo(T_i)
  - Si el log contiene <T_i starts> y <T_i commits>, redo(T_i)

Checkpoints

- Las bitácoras pueden hacerse muy grandes, y la recuperación puede llevar mucho tiempo
- Los Checkpoints acortan la bitácora y el tiempo de recuperación.
- Esquema de Checkpoints:
    1. Escribir todos los log records que se encuentren en memoria volátil a estable
    2. Escribir todos los datos modificados de volátil a estable
    3. Escribir un log record <checkpoint> al log en estable
- Ahora la recuperación solo incluye a los Ti, que comenzaron ejecución antes del último checkpoint y todas las transacciones después de Ti. Todas las demás transacciones ya están en memoria estable

Transacciones Concurrentes

- Deben ser equivalentes a ejecución en serie – serializabilidad
- Se podrían realizar todas las transacciones en sección crítica
  - Ineficiente, muy restrictivo
- Algoritmos de control de concurrencia proporcionan serializabilidad

Serializabilidad

- Considera datos A y B
- Considera transacciones T_0 y T_1
- Ejecutar T_0, T_1 atómicamente
- Secuencia de ejecución = schedule
- Orden de transacciones ejecutadas atómicamente = serial schedule
- Para N transacciones, hay N! órdenes seriados válidos

Orden 1: T_0, T_1

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Orden No en Serie

- Un orden no en serie permite ejecuciones traslapadas
  - Ejecución no incorrecta
- Considerar orden S, operaciones O_i, O_j
  - Conflicto si accesan el mismo dato, con al menos un write
- Si O_i, O_j son operaciones consecutivas de transacciones diferentes y O_i y O_j no presentan conflicto
  - S' con orden O_j O_i es equivalent a S
- Si S se puede convertir e S' via intercambio (swapping) de operaciones sin conflicto
  - S es serializable

Orden 2: Concurrente Serializable

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Protocolo de Candado (Locking)

- Asegura serializabilidad asociando un candado con cada dato
  - Seguir el protocolo de candado para control de acceso
- Locks
  - Shared – T_i tiene candado compartido lock (S) en Q, T_i puede leer Q pero no escribir en Q
  - Exclusive – Ti tiene candado exclusivo lock (X) en Q, T_i puede leer y escribir Q
- Requiere que cada transacción en Q adquiera el candado apropiado
- Si el candado está en uso (held), otras solicitudes pueden tener que esperar
  - Similar al algoritmo readers-writers

Protocolo de candado de Dos Fases

- Generalmente asegura la serialización de conflictos
- Cada transacción emite lock y unlock en dos fases
  - Growing – adquiere locks
  - Shrinking – libera locks
- No previene deadlock

Protocolos Basados en Timestamp

- Selecciona un orden de transacciones por adelantado – timestamp-ordering
- Transacción T_i asociada con timestamp $TS(T_i)$ antes que T_i comience
  - $TS(T_i) < TS(T_j)$ si Ti entra al sistema antes que T_j
  - TS puede ser generada del reloj del sistema o con un contador lógico, incrementado con cada entrada a transacción
- Timestamps determinan serializabilidad
  - Si $TS(T_i) < TS(T_j)$, el sistema debe asegurar que el orden producido es equivalente al orden seriado donde T_i ocurre antes que T_j

Protocolo Timestamp (Implementación)

- El dato Q obtiene dos tiempos (timestamps)
  - W-timestamp(Q) – tiempo mayor de latransacción que ejecutó write(Q) exitosamente
  - R-timestamp(Q) – tiempo mayor de latransacción que ejecutó read(Q) exitosamente
  - Se actualiza cuando se ejecutan read(Q) o write(Q)
- Protocolo de ordenamiento de Timestamps asegura que los conflictos de read y write se ejecutan en el orden definido por la asignación de tiempos (timestamps)
- Si Ti ejecuta read(Q)
  - If $TS(T_i) < W\text{-timestamp}(Q)$, Ti necesita leer el valor de Q que ya ha sido sobre-escritowas already overwritten
    - Se rechaza la operación read y se hace un roll back en T_i
  - If $TS(T_i) \geq W\text{-timestamp}(Q)$
    - Se ejecuta el read, $R\text{-timestamp}(Q) = \max(R\text{-timestamp}(Q), TS(T_i))$

Protocolo Timestamp

- Si Ti ejecuta write(Q)
  - If $TS(T_i)$ < R-timestamp(Q), el valor de Q producido por T_i se necesitaba previamente, T_i supuso que ya existía y nunca se produjo
    - Se rechaza la operaciónn Write, se hace roll back en T_i
  - If $TS(T_i)$ < W-timestamp(Q), T_i trata de escribir un valor obsoleto en Q
    - Se rechaza la operación Write y se hace roll back en T_i
  - Si no, se ejecuta el write
- A toda transacción T_i a la que se le aplica un roll back, se le asigna un nuevo tiempo y se vuelve a ejecutar
- El algorithm asegura la serialización de conflictos y la evasión de deadlocks

Orden: Protocolo Timestamp

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)