Monitores

Programación concurrente

Hasta ahora vimos:

- Los programas concurrentes pueden hace un uso eficiente de los recursos computacionales.

Hasta ahora vimos:

- Los programas concurrentes pueden hace un uso eficiente de los recursos computacionales.
- Sin embargo, en un modelo con memoria compartida *interleavings* indeseables pueden generar resultados erróneos.

Hasta ahora vimos:

- Los programas concurrentes pueden hace un uso eficiente de los recursos computacionales.
- Sin embargo, en un modelo con memoria compartida *interleavings* indeseables pueden generar resultados erróneos.
- Los *Semáforos* son una herramienta eficiente para resolver los problemas de sincronización.

Hasta ahora vimos:

- Los programas concurrentes pueden hace un uso eficiente de los recursos computacionales.
- Sin embargo, en un modelo con memoria compartida *interleavings* indeseables pueden generar resultados erróneos.
- Los *Semáforos* son una herramienta eficiente para resolver los problemas de sincronización.
    1. Son muy bajo nivel (es fácil olvidarse un acquire o un release).

Hasta ahora vimos:

- Los programas concurrentes pueden hace un uso eficiente de los recursos computacionales.
- Sin embargo, en un modelo con memoria compartida *interleavings* indeseables pueden generar resultados erróneos.
- Los *Semáforos* son una herramienta eficiente para resolver los problemas de sincronización.
  1. Son muy bajo nivel (es fácil olvidarse un acquire o un release).
  2. No están vinculados a datos (pueden aparecer en cualquier parte del código).

Monitores

Monitores

- Combina tipos de datos abstractos y exclusión mutua
  - Propuesto por Tony Hoare [1974]

Monitores

- Combina tipos de datos abstractos y exclusión mutua
  - Propuesto por Tony Hoare [1974]
- Incorporados en lenguajes de programación modernos
  - Java
  - C#

Elementos principales

Elementos principales

- Conjunto de operaciones encapsuladas en módulos.

Elementos principales

- Conjunto de operaciones encapsuladas en módulos.
- Un único *lock* que asegura exclusión mutua para todas las operaciones del monitor.

Elementos principales

- Conjunto de operaciones encapsuladas en módulos.
- Un único *lock* que asegura exclusión mutua para todas las operaciones del monitor.
- Variables especiales llamadas *condition variables*, utilizadas para programar sincronización condicional. En este curso trabajaremos utilizando una única variable de condición por Monitor.

Ejemplo: Vector

```
monitor Vector {

  int[] valores = [0,0,0,0,0,0]

  public void escribir(int nuevoValor) {
    int i = 0;
    while (i<6) {
      valores[i] = nuevoValor;
      i++;
    }
  }

  public void imprimir() {
    int i = 0;
    while (i<6) {
      print(valores[i]);
      i++;
    }
  }

}
```

Ejemplo: Vector

- ¿Pueden dos threads estar seteando el vector de manera concurrente?

Ejemplo: Vector

- ¿Pueden dos threads estar seteando el vector de manera concurrente? No

Ejemplo: Vector

- ¿Pueden dos threads estar seteando el vector de manera concurrente? No
- ¿Puede un thread estar seteando el vector mientras otro lo imprime?

Ejemplo: Vector

- ¿Pueden dos threads estar seteando el vector de manera concurrente? No
- ¿Puede un thread estar seteando el vector mientras otro lo imprime? No

Ejemplo: Vector

- ¿Pueden dos threads estar seteando el vector de manera concurrente? No
- ¿Puede un thread estar seteando el vector mientras otro lo imprime? No
- Esto siempre hablando de *la misma instancia de Vector*. El *Lock* es único para cada *instancia.*

Condition variables

- Están ligadas al monitor.

Condition variables

- Están ligadas al monitor.
- Poseen dos operaciones:
  - `wait()`
  - `notify()`/`notifyAll()`

Condition variables

- Están ligadas al monitor.
- Poseen dos operaciones:
  - `wait()`
  - `notify()`/`notifyAll()`
- Como vamos a utilizar una única variable de condición, desde el cuerpo del monitor podemos escribir directamente las invocaciones a estas funciones

Condition variables

- Están ligadas al monitor.
- Poseen dos operaciones:
  - `wait()`
  - `notify()`/`notifyAll()`
- Como vamos a utilizar una única variable de condición, desde el cuerpo del monitor podemos escribir directamente las invocaciones a estas funciones
- Al igual que los semáforos fuertes, las variables de condición tienen asociadas una **cola** de procesos bloqueados.

Condition variables

- Están ligadas al monitor.
- Poseen dos operaciones:
  - `wait()`
  - `notify()`/`notifyAll()`
- Como vamos a utilizar una única variable de condición, desde el cuerpo del monitor podemos escribir directamente las invocaciones a estas funciones
- Al igual que los semáforos fuertes, las variables de condición tienen asociadas una **cola** de procesos bloqueados.
- Por otro lado, el *lock* también tiene un **conjunto** de procesos bloqueados asociados.

En el Ejemplo Vector

Supongamos que no queremos que se puedan hacer dos impresiones seguidas (debe haber una escritura en el medio).

En el Ejemplo Vector

Supongamos que no queremos que se puedan hacer dos impresiones seguidas (debe haber una escritura en el medio).

```
bool huboEscritura = False;

public void escribir(int nuevoValor) {
  int i = 0;
  while (i<6) {
    valores[i] = nuevoValor;
    i++;
  }
  huboEscritura = True;
  notify();
}

public void imprimir() {
  while (!huboEscritura) {
      wait();
  }
  int i = 0;
  while (i<6) {
    print(valores[i]);
    i++;
  }
  huboEscritura = False;
}
```

Wait

- Bloquea al proceso en ejecución y lo asocia a la variable.

Wait

- Bloquea al proceso en ejecución y lo asocia a la variable.
- Al bloquearse libera el *lock* permitiendo la entrada de otro.

Notify/NotifyAll

- Desbloquea al primer proceso de la variable de condición (`NotifyAll` desbloquea a todos).

Notify/NotifyAll

- Desbloquea al primer proceso de la variable de condición (`NotifyAll` desbloquea a todos).
- La ejecución de estos procesos se da desde la instrucción siguiente al `wait` que lo bloqueó.

Notify/NotifyAll

- Desbloquea al primer proceso de la variable de condición (`NotifyAll` desbloquea a todos).
- La ejecución de estos procesos se da desde la instrucción siguiente al `wait` que lo bloqueó.
- ¿Qué sucede con el *lock*?

Notify/NotifyAll

- Desbloquea al primer proceso de la variable de condición (`NotifyAll` desbloquea a todos).
- La ejecución de estos procesos se da desde la instrucción siguiente al `wait` que lo bloqueó.
- ¿Qué sucede con el *lock*? El proceso desbloqueado debe competir por él.

Notify/NotifyAll

- Desbloquea al primer proceso de la variable de condición (`NotifyAll` desbloquea a todos).
- La ejecución de estos procesos se da desde la instrucción siguiente al `wait` que lo bloqueó.
- ¿Qué sucede con el *lock*? El proceso desbloqueado debe competir por él.
- Por ello, *no tenemos garantía de que la condición siga valiendo cuando le vuelva a tocar*. Por eso volvemos a preguntar por la condición (Ponemos `while` en vez de `if`)

Notify/NotifyAll

- Desbloquea al primer proceso de la variable de condición (`NotifyAll` desbloquea a todos).
- La ejecución de estos procesos se da desde la instrucción siguiente al `wait` que lo bloqueó.
- ¿Qué sucede con el *lock*? El proceso desbloqueado debe competir por él.
- Por ello, *no tenemos garantía de que la condición siga valiendo cuando le vuelva a tocar*. Por eso volvemos a preguntar por la condición (Ponemos `while` en vez de `if`)
- ¿En qué se diferencia de un `release` de un semáforo?

Notify/NotifyAll

- Desbloquea al primer proceso de la variable de condición (`NotifyAll` desbloquea a todos).
- La ejecución de estos procesos se da desde la instrucción siguiente al `wait` que lo bloqueó.
- ¿Qué sucede con el *lock*? El proceso desbloqueado debe competir por él.
- Por ello, *no tenemos garantía de que la condición siga valiendo cuando le vuelva a tocar*. Por eso volvemos a preguntar por la condición (Ponemos `while` en vez de `if`)
- ¿En qué se diferencia de un `release` de un semáforo? No se acumulan permisos, no hay "pasaje en mano".

Ejemplo: Buffer

Ejemplo: Buffer

```
monitor Buffer {

  private Object dato = null;    // el dato compartido

  public Object read() {
    while (dato == null)
      wait();
    aux = dato;
    dato = null;
    notifyAll();
    return aux;
  }

  public void write(Object o) {
    while (dato != null)
      wait();
    dato = o;
    notifyAll();
  }

}
```

Ejemplo: Buffer

¿Por qué son `notifyAll()`?

Ejemplo: Buffer

¿Por qué son `notifyAll()`?
En este caso, usar `notify()` puede traer problemas. Consideremos el siguiente escenario:

Ejemplo: Buffer

¿Por qué son `notifyAll()`?
En este caso, usar `notify()` puede traer problemas. Consideremos el siguiente escenario:

- El buffer comienza vacío.

Ejemplo: Buffer

¿Por qué son `notifyAll()`?
En este caso, usar `notify()` puede traer problemas. Consideremos el siguiente escenario:

- El buffer comienza vacío.
- Llega un consumidor **C1**. No puede leer, se pone a esperar.

Ejemplo: Buffer

¿Por qué son `notifyAll()`?
En este caso, usar `notify()` puede traer problemas. Consideremos el siguiente escenario:

- El buffer comienza vacío.
- Llega un consumidor **C1**. No puede leer, se pone a esperar.
- Llega un consumidor **C2**. Tampoco puede leer, se pone a esperar.

Ejemplo: Buffer

¿Por qué son `notifyAll()`?
En este caso, usar `notify()` puede traer problemas. Consideremos el siguiente escenario:

- El buffer comienza vacío.
- Llega un consumidor **C1**. No puede leer, se pone a esperar.
- Llega un consumidor **C2**. Tampoco puede leer, se pone a esperar.
- Llega un productor **P1**. Escribe, lanza un `notify()` que despierta a **C1**.

Ejemplo: Buffer

¿Por qué son `notifyAll()`?
En este caso, usar `notify()` puede traer problemas. Consideremos el siguiente escenario:

- El buffer comienza vacío.
- Llega un consumidor **C1**. No puede leer, se pone a esperar.
- Llega un consumidor **C2**. Tampoco puede leer, se pone a esperar.
- Llega un productor **P1**. Escribe, lanza un `notify()` que despierta a **C1**.
- Llega un productor **P2** y obtiene el lock antes que **C1**. No puede escribir, se pone a esperar.

Ejemplo: Buffer

¿Por qué son `notifyAll()`?
En este caso, usar `notify()` puede traer problemas. Consideremos el siguiente escenario:

- El buffer comienza vacío.
- Llega un consumidor **C1**. No puede leer, se pone a esperar.
- Llega un consumidor **C2**. Tampoco puede leer, se pone a esperar.
- Llega un productor **P1**. Escribe, lanza un `notify()` que despierta a **C1**.
- Llega un productor **P2** y obtiene el lock antes que **C1**. No puede escribir, se pone a esperar.
- **C1** lee el dato que escribió **P1**. Lanza un `notify()` que *debería despertar a* **P2**, sin embargo, despierta a **C2**, que sigue en la cola.

Ejemplo: Buffer

¿Por qué son `notifyAll()`?
En este caso, usar `notify()` puede traer problemas. Consideremos el siguiente escenario:

- El buffer comienza vacío.
- Llega un consumidor **C1**. No puede leer, se pone a esperar.
- Llega un consumidor **C2**. Tampoco puede leer, se pone a esperar.
- Llega un productor **P1**. Escribe, lanza un `notify()` que despierta a **C1**.
- Llega un productor **P2** y obtiene el lock antes que **C1**. No puede escribir, se pone a esperar.
- **C1** lee el dato que escribió **P1**. Lanza un `notify()` que *debería despertar a* **P2**, sin embargo, despierta a **C2**, que sigue en la cola.
- El sistema se queda esperando, aunque hay un productor que podría aprovechar el buffer.

Ejercicio: Buffer de dimensión N

Ejercicio: Buffer de dimensión N

```
monitor Buffer {
  private Object[] datos = new Object[N+1];
  private int begin = 0, end = 0;

  public write(Object o) {
    while (next(begin) == end) wait();
    datos[begin] = o;
    begin = next(begin);
    notifyAll();
  }

  public read() {
    while (begin == end) wait();
    result = datos[end];
    end = next(end);
    notifyAll();
    return result;
  }

  private int next(int i) {
    return (i+1)%(N+1);
  }

}
```

Monitor que define a un semáforo

Monitor que define a un semáforo

```
monitor Semaphore {
  private int permisos;

  public Semaphore(int n) {
    this.permisos = n;
  }

  public void acquire() {
    while (permisos == 0)
      wait();
    permisos--;
  }

  public void release() {
    permisos++;
    notify();
  }

}
```

- ¿Tiene algún problema esta solución?

Monitor que define a un semáforo

```
monitor Semaphore {
  private int permisos;

  public Semaphore(int n) {
    this.permisos = n;
  }

  public void acquire() {
    while (permisos == 0)
      wait();
    permisos--;
  }

  public void release() {
    permisos++;
    notify();
  }

}
```

- ¿Tiene algún problema esta solución? Tiene *starvation*.

Monitor que define un semáforo II

```
public void acquire() {
  if (permisos == 0) {
    esperando++;
    wait();
  } else {
    permisos--;
  }
}

public void release() {
  if (esperando == 0) {
    permisos++;
  } else {
    notify();
    esperando--;
  }
}
```

Monitor que define un semáforo II

```
public void acquire() {
  if (permisos == 0) {
    esperando++;
    wait();
  } else {
    permisos--;
  }
}

public void release() {
  if (esperando == 0) {
    permisos++;
  } else {
    notify();
    esperando--;
  }
}
```

- Mejor, pero hay que tener cuidado con el `if` y el `wait` (En Java debe ser evitado debido a la existencia de *wakeups espíureos*)

Monitor que define un semáforo II

```
public void acquire() {
  if (permisos == 0) {
    esperando++;
    wait();
  } else {
    permisos--;
  }
}

public void release() {
  if (esperando == 0) {
    permisos++;
  } else {
    notify();
    esperando--;
  }
}
```

- Mejor, pero hay que tener cuidado con el `if` y el `wait` (En Java debe ser evitado debido a la existencia de *wakeups espíureos*)
- Se puede hacer una implementación de semáforos fuertes, pero es más sofisticada.

Monitores en Java

Monitores en Java

- Toda clase tiene un lock y una única variable de condición.
  - Ventaja: Queda bien encapsulado (no se puede manipular el lock ni la variable de condición)

Monitores en Java

- Toda clase tiene un lock y una única variable de condición.
  - Ventaja: Queda bien encapsulado (no se puede manipular el lock ni la variable de condición)
  - Desventaja: Usar más de una variable de condición puede mejorar la eficiencia
- Los métodos `wait`, `notify` y `notifyAll` pertenecen a la interfaz de la clase `Object`

Monitores en Java

- Toda clase tiene un lock y una única variable de condición.
  - Ventaja: Queda bien encapsulado (no se puede manipular el lock ni la variable de condición)
  - Desventaja: Usar más de una variable de condición puede mejorar la eficiencia
- Los métodos `wait`, `notify` y `notifyAll` pertenecen a la interfaz de la clase `Object`
- Es necesario usar el keyword `synchronized` en cada método del monitor
  - Garantiza exclusión mutua
  - Permite invocar las operaciones sobre la variable de condición

IllegalMonitorStateException

- Sólo se pueden invocar los métodos wait, notify y notifyAll desde métodos synchronized.
- En caso contrario se emite una excepción.

```
public void m1() {
  this.wait(); // IllegalMonitorStateException
}

public void m1() {
  this.notify(); // IllegalMonitorStateException
}
```

Buffer de dimensión N en Java

```
class Buffer {
  private Object[] data = new Object[N+1];
  private int begin = 0, end = 0;

  public synchronized void write(Object o) {
    while (isFull()) try{wait();} catch(Exception e) { return; }
    data[begin] = o;
    begin = next(begin);
    notifyAll();
  }

  public synchronized Object read() {
    while (isEmpty()) try{wait();} catch(Exception e) { return; }
    Object result = data[end];
    end = next(end);
    notifyAll();
    return result;
  }

  private boolean isEmpty() { return begin == end; }
  private boolean isFull()  { return next(begin) == end; }
  private int next(int i)   { return (i+1)%(N+1); }

}
```

Threads en Java (Productor)

```
class Productor extends Thread {

  private final Buffer buffer;

  public Productor(Buffer buffer) {
    this.buffer = buffer;
  }

  public void run() {
    int i = 0;
    while (true) {
      buffer.write(i);
      i++;
    }
  }

}
```

Threads en Java (Consumidor)

```
class Consumidor implements Runnable {

  private final Buffer buffer;

  public Consumidor(Buffer buffer) {
    this.buffer = buffer;
  }

  public void run() {
    while (true) {
      Object o = buffer.read();
      System.out.println("Leido " + o.toString());
    }
  }

}
```

Threads en Java (thread principal)

```
public static void main(String[] args) {
  Buffer buffer = new Buffer();
  Productor p   = new Productor(buffer);
  Consumidor c = new Consumidor(buffer);
  Thread ct     = new Thread(c);

  p.start();
  ct.start();
}
```

El método `start` inicia un nuevo thread que ejcutará de forma concurrente con el original. El nuevo thread ejecutará internamente el método `run`.

InterruptedException

- El método wait puede arrojar una excepción.
- Esto ocurre cuando se interrumpe al thread en espera usando el método interrupt.
- Aunque no lo usemos es una buena práctica considerar eventuales interrupciones.

Lectores Escritores con Monitores

¿Cuáles deberían ser los métodos para un monitor que representa una base de datos que puede ser leída o escrita?

Lectores Escritores con Monitores

¿Cuáles deberían ser los métodos para un monitor que representa una base de datos que puede ser leída o escrita?

```
class Database {

  synchronized void beginWrite();

  synchronized void endWrite();

  synchronized void beginRead();

  synchronized void endRead();

}
```

Lectores Escritores Solución

```
class Database {

  private int writers = 0;
  private int readers = 0;

  private boolean canRead() {
    return writers == 0;
  }

  private boolean canWrite() {
    return writers == 0 && readers == 0;
  }

  ...

}
```

Lectores Escritores Solución

```
syncrhonized void beginRead() {
  while (!canRead()) {
    try { wait(); }
    catch (InterruptedException e) { return; }
  }
  readers++;
}

syncrhonized void endRead() {
  readers--;
  if (readers == 0)
    notify();
}
```

Lectores Escritores Solución

```
syncrhonized void beginWrite() {
  while (!canWrite()) {
    try { wait(); }
    catch (InterruptedException e) { return; }
  }
  writers = 1;
}

syncrhonized void endWrite() {
  writers = 0;
  notifyAll();
}
```

Lectores Escritores (Prioridad Escritores)

```
private int waitingWriters = 0;

private boolean canRead() {
  return writers == 0 && waitingWriters == 0;
}

syncrhonized void beginWrite() {
  while (!canWrite()) {
    waitingWriters++;
    try { wait(); } catch (InterruptedException e) { return; }
    waitingWriters--;
  }
  writers = 1;
}

syncrhonized void endWrite() {
  writers = 0;
  notifyAll();
}
```

Resumen

- Con Monitores podemos resolver problemas de sincronización.

Resumen

- Con Monitores podemos resolver problemas de sincronización.
- Tienen un mayor nivel de abstracción que los semáforos.

Resumen

- Con Monitores podemos resolver problemas de sincronización.
- Tienen un mayor nivel de abstracción que los semáforos.
- En Java todo objeto tiene lock y una (única) variable de condición asociada.

Resumen

- Con Monitores podemos resolver problemas de sincronización.
- Tienen un mayor nivel de abstracción que los semáforos.
- En Java todo objeto tiene lock y una (única) variable de condición asociada.
- La "condición" no está en la variable, está en el `while` que envuelve la espera.