UD 3.- Primitives de sincronització

Concurrència i Sistemes Distribuïts

Objectius de la Unitat Didàctica

- Explicar el model de programació concurrent proporcionat pels monitors.
- Solucionar el problema de la sincronització condicional mitjançant la utilització de monitors.
- Construir monitors en llenguatges de programació concurrent, evitant els seus problemes potencials.
- Avaluar les variants de monitor existents.

Contingut

- Llenguatges de programació concurrent
- Concepte de monitor
- Variants de monitor
- Invocacions niades

Llenguatges de Programació Concurrent

- La Programació Concurrent ha de resoldre les necessitats de comunicació i sincronització entre fils
- Podem dissenyar estratègies de solució a diversos nivells:
  - Sense suport del Sistema Operatiu
    - Utilitzen espera activa (bucle), i per tant són ineficients
    - Inhabilitació d'interrupcions (només possible en mode supervisor)
    - Primitives testAndSet o Swap
  - Amb suport per part del SO (tots els SO moderns disposen del suport necessari)
    - Semàfors
      - Avantatges: eficient (s'espera de manera suspesa) i flexible
      - Inconvenient: ni el llenguatge ni el compilador reforcen l'encapsulació ni detecten ús incorrecte
  - Utilitzant Llenguatges de Programació Concurrent
    - Monitors

Llenguatges de Programació Concurrent

- Els Llenguatges de Programació Concurrent:
  - Inclouen construccions específiques per a representar la concurrència
    - Suport per a gestió de fils, primitives per a comunicació i sincronització, etc.
- Actualment els més populars són Java i C#
  - Tots dos utilitzen POO i usen un model de memòria compartida (objectes compartits) i el concepte de monitor
    - Nosaltres ens centrem en Java
  - Entre els llenguatges que utilitzen pas de missatges destaca Erlang, però hi ha uns altres (Occam, Go...)
- Existeixen biblioteques per a estendre llenguatges que no suporten concurrència
  - ex. pthreads sobre C

Exemple.- Formigues

- Les formigues comparteixen un territori
  - Matriu de cel·les, cadascuna amb valors lliure/ocupat
  - Restricció.- màxim una formiga per cel·la
  - Existeix un *lock* que protegeix aquest territori
- Cada formiga es modela mitjançant un fil
  - A partir de la cel·la actual, es desplaça a una altra veïna
  - Però si està ocupada, ha d'esperar
- Quan una formiga es desplaça des de (x,y) a (x',y') executa

```
tancar lock
// Si ocupada[x',y'] ha d'esperar fins que quede lliure
ocupada[x',i']=true; ocupada[x,i]=false //actualitza matriu
obrir lock
```

Contingut

- Llenguatges de programació concurrent
- Concepte de monitor
- Variants de monitor
- Invocacions niades

Monitor.- motivació

- Les primitives obrir i tancar *lock* garanteixen accés segur a variables compartides
- Però també necessitem altres primitives que permeten esperar de forma segura fins que es complisca determinada condició lògica (sincronització)
  - Un bucle d'espera (bucle buit que comprova repetidament la condició) no funciona. Per què?

```
tancar lock
while (ocupada[x',y']) {} //NO funciona
ocupada[x',y']=true; ocupada[x,y]=false
obrir lock
```

Monitor.- motivació

- La major part dels Llenguatges de Programació moderns
  - Són llenguatges orientats a objectes
    - El programador pot definir tipus de dades (classes)
    - Una classe permet definir variables (objectes)
    - Separació interfície/implementació
      - La interfície (part visible) correspon al seu comportament (conjunt de mètodes)
      - Implementació (part oculta).- conjunt d'atributs, codi dels mètodes
  - Requereixen concurrència
- Idea.- barrejar POO i Programació Concurrent
  - Els fils es coordinen mitjançant objectes compartits
  - Ocultem els detalls d'exclusió mútua i sincronització en les classes que representen els objectes compartits
- Avantatges
  - Simplifica el desenvolupament, manteniment, i comprensió del codi
  - Facilita la depuració (podem provar cada peça per separat)
  - Facilita la reutilització de codi
  - Millora la documentació i la llegibilitat

Monitor

- Monitor = classe per a definir objectes que podem compartir de forma segura entre diferents fils
  - Els seus mètodes s'executen en exclusió mútua
    - Disposa d'una cua d'entrada on esperen aquells fils que volen utilitzar el monitor quan l'està utilitzant un altre fil
    - No hi ha condicions de carrera dins del monitor
  - Resol la sincronització
    - Podem definir cues d'espera (variables 'condition') dins del monitor
      - Ex.- condition noPle, noBuit; //per a productor/consumidor
    - Si un fil que executa codi dins del monitor necessita esperar fins que es complisca determinada condició lògica (ex.- *buffer* no buit)
      - Executa noBuit.wait() -> deixa lliure el monitor i espera sobre la cua noBuit
    - Quan un altre fil modifica l'estat del monitor (ex. un productor genera un element)
      - Executa noBuit.notify() -> reactiva un fil que espera en la cua noBuit

Exemple de monitor.- Productor/Consumidor

- L'objecte compartit és el *buffer*.
  - Dissenyem interfície i implementació (atributs i codi dels mètodes) com en qualsevol altra classe.
- Reomplim una taula on decidim per a cada mètode:
  - La seua interfície (nom, arguments, tipus de retorn).
  - En quins casos (estats del *buffer*) no es pot aplicar aquesta operació.
  - Si modifica l'estat de l'objecte, a quins fils en espera s'ha d'avisar.

Mètode	Espera quan	Avisa a
int get()	*Buffer* buit	Qui espere per *buffer* ple
void put(int e)	*Buffer* ple	Qui espere per *buffer* buit
int numElems()	--	--

- Definim una cua d'espera (variable 'condition') per a cada cas d'espera de la taula.
  - condition noPle, noBuit;

Exemple de monitor.- Productor/Consumidor

```
Monitor Buffer {                        //IMPORTANT.- NO ÉS JAVA, sinó pseudollenguatge
  .... //atributs per a implantar el monitor
  condition noPle, noBuit;  //cues d'espera
  int elems = 0;

  public Buffer() {..} //inicialització dels atributs
  entry void put(int x) { // entry= mètode públic amb accés en exclusió mútua
    if (elems==N) {noPle.wait();}       // espera en la cua noPle
      ...  //codi per a inserir l'element
    elems++;
    noBuit.notify();                    // reactiva a algú de la cua noBuit
  }
  entry int get() {
    if (elems==0) {noBuit.wait();}      // espera en la cua noBuit
      ... // codi per a extraure un element
    elems--;
    noPle.notify(); return ... ;        // reactiva a algú de la cua noPle
  }
  entry int numItems() {return elems;}
}
...
Buffer b; //i des de qualsevol fil es pot invocar b.numItems(), b.get() o b.put(x)
```

Monitor.- resum

- Monitor = Classe + Exclusió Mútua + Sincronització
- Oculta els detalls d'Exclusió Mútua i sincronització
  - L'execució de procediments en el mateix monitor no se solapa (exclusió mútua)
  - Per a la coordinació s'usen cues d'espera (variables condition)
    - Primitives per a esperar wait() i avisar a qui espera notify()
      - Si en avisar no hi ha ningú esperant, l'avís es perd (no té efecte)
    - Únicament poden utilitzar-se dins del monitor
    - El programador és responsable d'esperar/avisar en els moments oportuns
- Proporciona abstracció
  - El programador que invoca operacions sobre el monitor ignora com s'implementen
  - El programador que implementa el monitor ignora com s'usa

Monitor.- Exemple formigues

- El territori es modela mitjançant un monitor
  - Atributs:
    - Una matriu de valors lògics, indicant per a cada cel·la si lliure/ocupada
      - `boolean[N][N] ocupada;  // NO és Java`
  - Mètodes:
    - `entry void desplaça(x,y,x',y')` La formiga es desplaça des de la cel·la (x,y) a (x',y')
- Dues alternatives de solució:
  1. Una cua d'espera per cel·la per a esperar que aqueixa cel·la quede *lliure*
     - `condition[N][N] lliure;  // NO és Java`
  2. Una cua d'espera única anomenada *lliure*
     - `condition lliure;  // NO és Java`

Monitor.- exemple Formigues

Alternativa 1

```
Monitor  Terreny {
   boolean[N][N] ocupada;
   condition[N][N] lliure;

   entry void desplaça(int x,y,x',y') {
      if (ocupada[x',y'])
          lliure[x',y'].wait();
      //actualitza matriu
      ocupada[x',y']=true;
      ocupada[x,y]=false;
      //per a avisar a qui vol anar a x,y
      lliure[x,y].notify();
   }
}
```

Alternativa 2

```
Monitor  Terreny {
   boolean[N][N] ocupada;
   condition lliure;

   entry void desplaça(int x,y,x',y') {
      while (ocupada[x',y'])
          lliure.wait();
      //actualitza matriu
      ocupada[x',y']=true;
      ocupada[x,y]=false;
      //per a avisar a qui vol anar a x,y
      lliure.notifyAll();
   }
}
```

Monitor.- Exemple formigues

- Ambdues són correctes
- L'opció 1
  - És molt més eficient
    - Només reactiva a una formiga si ha quedat lliure la cel·la per la qual espera
  - L'ús de **if** només és possible en algunes variants de monitor (discutides després). La resta requereixen **while**
- L'opció 2
  - No és eficient
    - Reactiva a totes les formigues després d'alliberar cada cel·la
  - Es pot aplicar en totes les variants de monitor
  - És l'única opció si no podem definir diverses cues d'espera (variables condition)
- Sempre que resulte possible, triem l'alternativa 1

Java.- Monitor

- Dos nivells possibles
  - Suport bàsic en el llenguatge
  - Suport estès (mitjançant la biblioteca java.util.concurrent)
- En aquesta unitat ens centrem en el suport bàsic
  - Java.util.concurrent es desenvolupa en la unitat 5

Java suporta el concepte de monitor

- Tot objecte posseeix de forma implícita (sense necessitat de declarar-los)
  - Un *lock*
    - En etiquetar un mètode amb **synchronized**, es garanteix execució en exclusió mútua
      - equival a *tancar lock* abans de la seua primera instrucció i *obrir lock* després de l'última
  - Una cua d'espera amb primitives
    - wait().- espera sobre la cua d'espera
    - notify().- reactiva a un dels fils que esperen en aquesta cua
    - notifyAll().- reactiva a tots els que esperen
- Però **no** podem declarar altres *locks* ni altres cues d'espera

Java.- Com definir un Monitor

- Una classe que definisca objectes a compartir entre fils, deuria:
  - Definir tots els seus atributs com a **privats**
  - Sincronitzar tots els seus mètodes no privats (paraula synchronized)
  - En la implementació de cada mètode, accedir només a atributs de la classe i variables locals (definides en el propi mètode)
  - Utilitzar wait(), notify(), notifyAll() dins de mètodes sincronitzats
- **Alerta!** El compilador no comprova absolutament res (és responsabilitat del programador)
  - No hi ha cap tipus d'avís ni error si hi ha atributs no privats, o mètodes públics sense la paraula synchronized, o s'utilitza wait(), notify(), notifyAll() en un mètode no sincronitzat

Java.- Com definir un Monitor (cont.)

- En un monitor ideal els fils que esperen per condicions lògiques diferents esperen en cues d'espera diferents
  - Ex.- en productor consumidor podem tenir cues noBuit, noPle.
    - Els productors que troben buffer ple esperen en noPle,
    - Els consumidors que esperen perquè el buffer està buit esperen en noBuit
- Però Java utilitza únicament una variable condició per monitor
  - Els fils que esperen per condicions lògiques diferents esperen en una única cua, no en cues diferents
  - En reactivar un fil no sabem si reactivem al que esperava per una condició o per una altra
    - Excepte en casos molt simples, es recomana despertar a tots i que cadascun torne a comprovar la seua condició
  - La biblioteca java.util.concurrent (veure unitat didàctica 5) resol aquesta limitació
- L'esquema típic d'un mètode en un objecte compartit en Java és:

```
while (condicióLògica) { // espera mentre condició certa
    wait(); // try { wait } catch (InterruptedException e) {..}
}
... // codi normal
notifyAll(); // si hem modificat l'estat de l'objecte, avisa a  els fils en espera
```

Java.- Exemple formigues

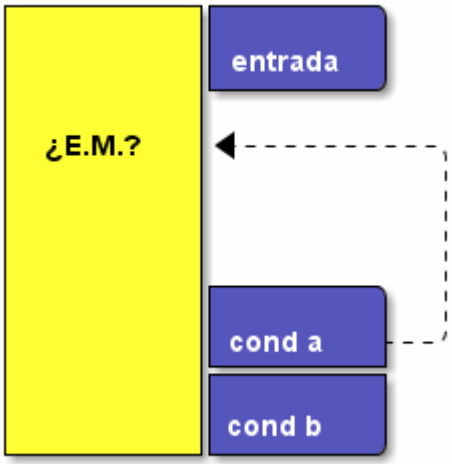
```
public class Territori {
    private boolean[][] ocupada;

    public Territori(int N) {
        ocupada=new boolean[N][N];
        for (int i=0; i<N; i++)
          for (int j=0; j<N; j++)
              ocupada[i][j]=false; // lliure
    }

    public synchronized void desplaça(int x0, int y0, int x, int y) {
        while (ocupada[x][y])
            try { wait(); } catch (InterruptedException e) {};
        ocupada[x0][y0]=false; ocupada[x][y]=true;
        notifyAll();
    }
}
```

Contingut

- Llenguatges de programació concurrent
- Concepte de monitor
- Variants de monitor
- Invocacions niades

Monitor.- Variants

- El monitor garanteix l'Exclusió **Mútua**
  - Només un fil executa codi del monitor en un instant donat
    - Si intenta executar codi i monitor ocupat, espera en l'entrada
    - Quan finalitza el mètode, el monitor queda lliure
  - Quan el fil actiu (W) en el monitor executa c.wait(), pansa a espera sobre c
    - **El monitor queda lliure** (espera fora de la SC)
    - Un altre fil (N) que espera en l'entrada passa a actiu en el monitor
- Problema: si el fil N executa c.notify()
  - Reactiva a W
  - Però només un (W o N) pot continuar actiu en el monitor. Quin?

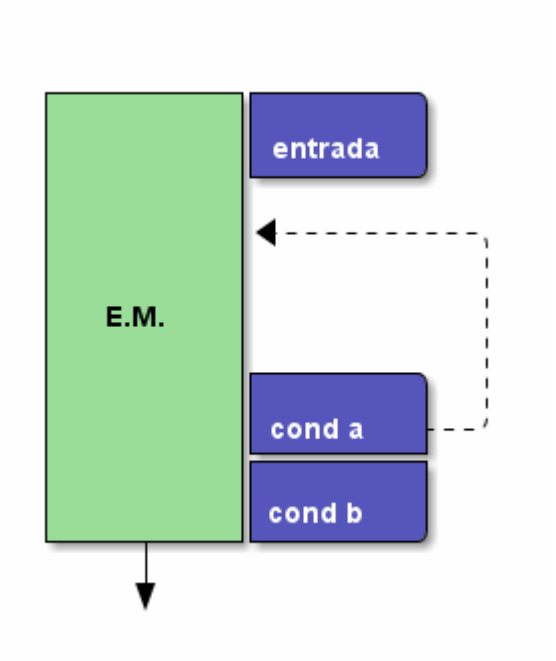


Monitor.- Variants

- Suposem que:
  - W espera en condició c
  - N executa c.notify() i reactiva W
- Alternatives de solució (variants de monitors)
  - El fil N abandona el monitor (model de Brinch Hansen)
  - El fil N espera en una cua especial (Hoare)
  - El fil W espera en l'entrada (Lampson-Redell)
- Quan parlem d'una cua especial, es tracta d'una cua per a esperar al fet que el monitor quede lliure
  - Però és prioritària sobre la cua d'entrada
  - Només s'extrau de l'entrada si la cua especial està buida
- Analitzem per separat cada variant

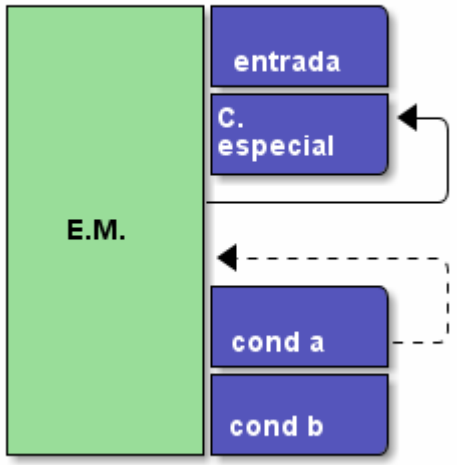
Monitor tipus Brinch Hansen

- Suposem que:
  - W espera en condició c
  - N executa c.notify() i reactiva W
- La sentencia *notify* és obligatòriament l'última sentència del mètode
  - N abandona el monitor i desperta al fil W
- Compleix exclusió mútua
  - N abandona el monitor
  - W queda actiu en el monitor
- No pot aplicar-se sempre
  - Alguns problemes complexos requereixen realitzar altres accions després de c.notify()



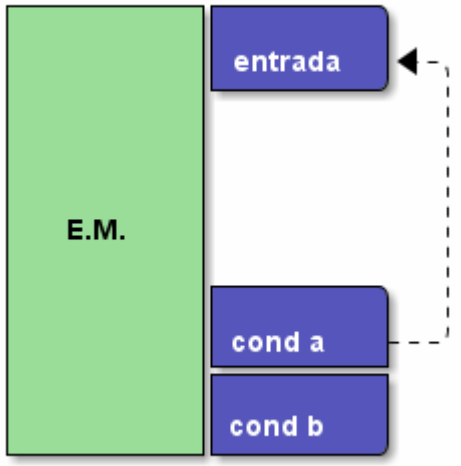
Monitor tipus Hoare

- Suposem que:
  - W espera en condició c
  - N executa c.notify() i reactiva W
- A més de l'entrada, hi ha una cua especial
- Quan N executa c.notify
  - N passa a la cua especial
  - W queda actiu en el monitor
- Compleix exclusió mútua
  - N espera fora del monitor
  - W queda actiu en el monitor



Monitor tipus Lampson-Redell

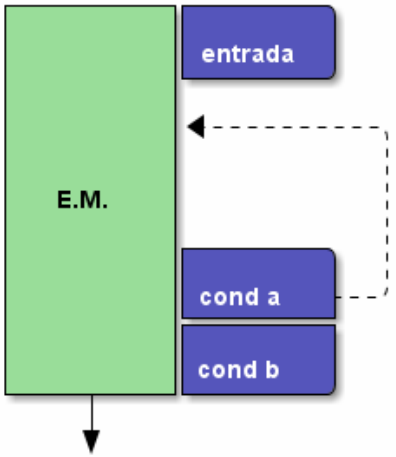
- Suposem que:
  - W espera en condició c
  - N executa c.notify() i reactiva W
- Quan N executa c.notify
  - W passa a la cua d'entrada
  - N queda actiu dins del monitor
- Compleix exclusió mútua
  - W espera fora del monitor (en l'entrada)
    - Quan aconseguisca entrar, l'estat pot haver canviat de nou: cal reavaluar la condició
  - N queda actiu en el monitor



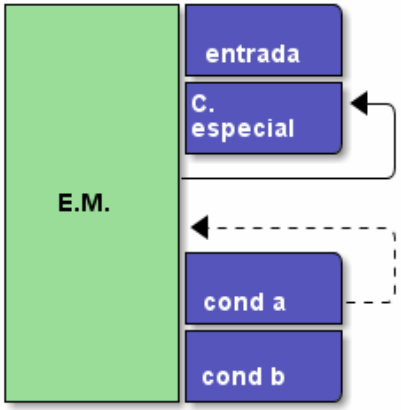
Variants de monitor.- Resum

Segueix W (troba condició OK)

Brinch Hansen

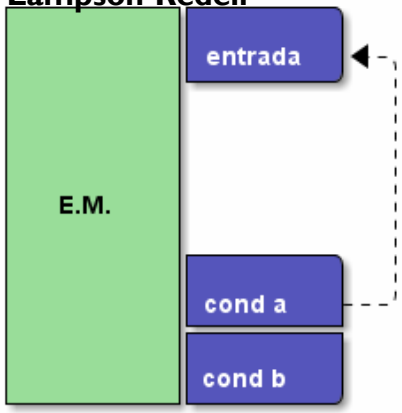


Hoare



Segueix N (W reavaluarà condició quan entre)

Lampson-Redell

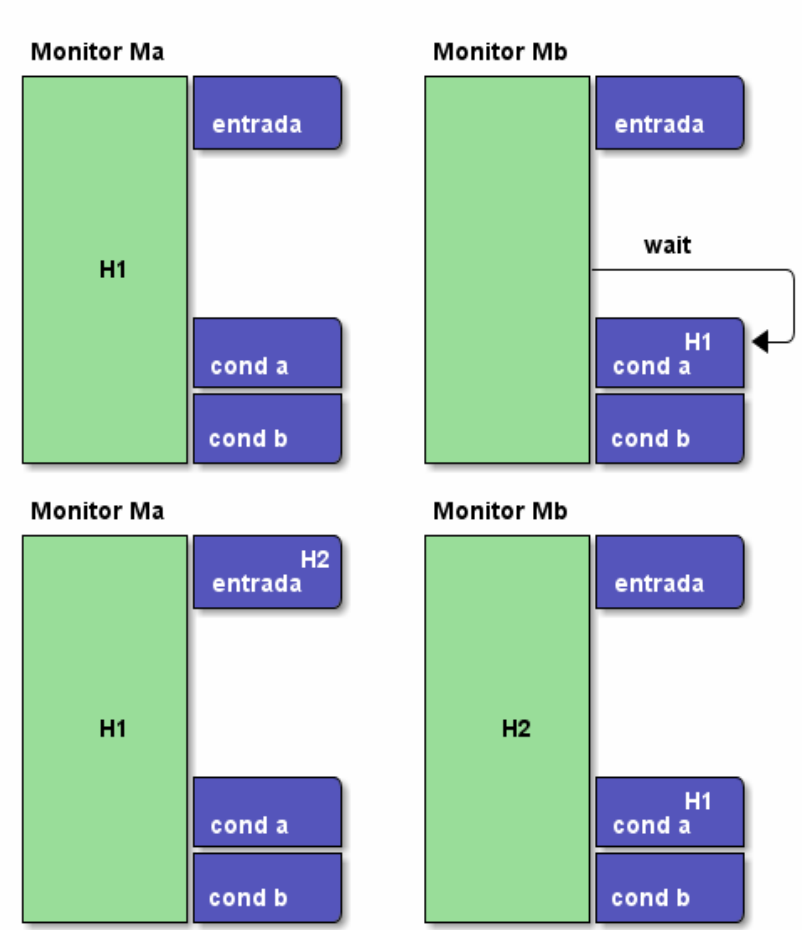


Contingut

- Llenguatges de programació concurrent
- Concepte de monitor
- Variants de monitor
- Invocacions niades

Monitor.- invocacions niades

- Invocar des d'un monitor a un mètode d'un altre monitor pot:
  - reduir la concurrència
  - i fins i tot provocar interbloquejos.
- Suposem
  - 2 monitors Ma i Mb: des d'un mètode de Ma s'invoca un mètode de Mb i viceversa
  - 2 fils H1 i H2
- H1 actiu en Ma, invoca un mètode de Mb, dins del qual s'executa a.wait()
  - Passa a la cua d'espera a del monitor Mb
  - Allibera el monitor Mb, però no Ma
    - Ningú pot usar Ma.- reduïm concurrència
- Si H2 entra en Mb (que estava lliure) i invoca un mètode del monitor Ma
  - Espera en la cua d'entrada de Ma (Ma està ocupat)
  - No deixa lliure el monitor Mb
  - Hem arribat **a un interbloqueig**



Invocacions niades.- Exemple d'interbloqueig

- Definim dos monitors (p,q) de tipus Bcell.
- Suposem 2 fils concurrents H1 i H2:
  - H1 invoca p.swap(q), obté accés al monitor p, i inicia l'execució de p.swap.
  - H2 invoca q.swap(p), obté accés al monitor q, i inicia l'execució de q.swap.
- Apareix un interbloqueig:
  - Dins de p.swap, H1 invoca q.getValue(), però ha d'esperar perquè el monitor q no està lliure.
  - Dins de q.swap, H2 invoca p.getValue(), però ha d'esperar perquè el monitor p no està lliure.
  - Tots dos s'esperen mútuament, i la situació no pot evolucionar.

```
class BCell {
  int value;
  public synchronized void getValue() {
    return value;
  }
  public synchronized void setValue(int i) {
    value=i;
  }
  public synchronized void swap(BCell x) {
    int temp= getValue();
      setValue(x.getValue());
      x.setValue(temp);
  }
}
```

Resultats d'aprenentatge de la Unitat Didàctica

- En finalitzar aquesta unitat, l'alumne ha de ser capaç de:
  - Programar solucions eficients al problema de la sincronització condicional, utilitzant monitors.
  - Dissenyar adequadament un nou monitor, segons les condicions que haja de gestionar.
  - Comparar les variants de monitor existents.
    - Classificar els llenguatges de programació concurrent d'acord amb la variant que suporten.