21720 Programació concurrent

6. Monitors

- **Semàfors**: Primitives de sincronització sense espera activa. Són de baix nivell, no estan estructurades i poden provocar errors crítics

- **Monitors**: Primitives estructurades que concentren en mòduls la responsabilitat de correctesa. Són una generalització del nucli dels SO on l'accés a la SC està centralitzat en un programa específic

- Són una generalització d'un objecte a **programació OO** que encapsula dades i operacions amb una classe

    - En temps d'execució es poden assignar objectes d'una classe i es poden invocar les operacions de la classe

    - Amb monitors: Només un procés pot executar una operació en un objecte en un determinat moment

- Formalització del monitors: Hoare 1973
- Monitors per estructurar els SOs amb llenguatges d'alt nivell:
  - El SO és un conjunt de mòduls, *schedulers*, que assigna recursos compartits entre processos
  - Monitor és el conjunt de procediments i dades que ha de gestionar cada *scheduler*
  - El monitor ha d'assegurar l'exclusió mútua en l'execució dels seus procediments, les variables del monitor només es poden accedir des de aquests procediments
- Primers llenguatges: Concurrent Pascal, Concurrent C, Mesa, ADA i Java

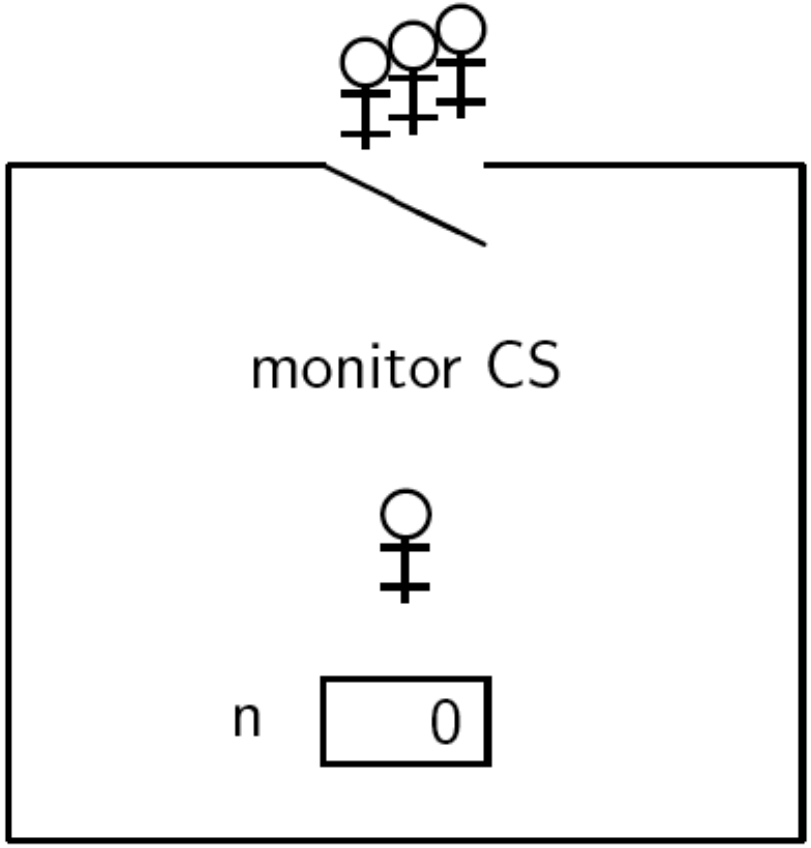
Algorithm 7.1: Atomicity of monitor operations

```
monitor CS
    integer n ← 0

    operation increment
        integer temp
        temp ← n
        n ← temp + 1
```

p	q
p1: CS.increment	q1: CS.increment

- Declaració i ús de monitors: Problema de l'increment del comptador encapsulat amb un monitor
  - El monitor CS conté una variable n i una operació d'increment
  - *n* no és accessible fora del monitor
  - *p* i *q* són dos processos que criden a l'operació del monitor i per definició només una es pot executar a la vegada assegurant l'exclusió mútua

- L’algorisme soluciona el problema de la SC
- La sincronització és implícita i no requereix que el programador col·loqui bé els *wait* i *signals*
- El monitor és una entitat estàtica no un procés dinàmic. El procés entra al monitor que queda tancat fins que el procés en surt
- Similar als semàfors si varis processos intenten entrar només un ho aconsegueix. No hi ha cap cua associada per tant la inanició és possible



Monitors i objectes en Java

- Java no té cap constructor especial per a monitor, cada objecte té un ***lock*** que pot ser usat per accedir al seus atributs
- Cada *lock* es pot usar per forçar l'exclusió mútua d'un bloc de codi indicant que està ***syncronized*** amb l'objecte. Java afegeix de forma automàtica el *lock* i *unlock* al principi i final del bloc
- També es poden declarar com a *syncronized* els mètodes que accedeixen a recursos compartits. Així el *lock* està associat a l'objecte instanciat
- Una classe amb tots els mètodes públics *syncronized* s'anomena **monitor Java**

CounterObject.java

```java
public class CounterObject implements Runnable {

    static final int THREADS = 4;
    static final int MAX_COUNT = 10000000;
    // El comptador no és static és un atribut d'un CounterObject
    public volatile int counter = 0;
    Object lock = new Object();

    @Override
    public void run() {
        int max = MAX_COUNT / THREADS;
        // L'ident. d'un fil és un long fixe durant la vida del fil
        System.out.printf("Thread %d\n", Thread.currentThread().getId());
        for (int i = 0; i < max; i++) {
            synchronized (lock) {
                counter += 1;
            } // unlock automàtic
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread[] threads = new Thread[THREADS];
        CounterObject c = new CounterObject();
        for (int i = 0; i < THREADS; i++) {
            threads[i] = new Thread(c);
            threads[i].start();
```

- Solució al problema del comptador compartit usant el bloqueig (*lock*) d'un objecte

https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

CounterMethod.java

```java
public class CounterMethod implements Runnable {
    static final int THREADS = 4;
    static final int MAX_COUNT = 10000000;
    public volatile int counter = 0;

    synchronized void add() {
        counter++;
    }

    @Override
    public void run() {
        int max = MAX_COUNT/THREADS;
        System.out.printf("Thread %d\n", Thread.currentThread().getId());
        for (int i =0; i < max; i++) {
            this.add();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread[] threads = new Thread[THREADS];
        int i;
        CounterMethod c = new CounterMethod();
        for (i=0; i< THREADS; i++) {
            threads[i] = new Thread(c);
```

- Solució al problema del comptador compartit amb el bloqueig d'un mètode

https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

SyncronizedCounter.java

```java
package pkg257_countermethod;

public class CounterMethod implements Runnable {

    static final int THREADS = 4;
    static final int MAX_COUNT = 10000000;
    static SynchronizedCounter sc;

    @Override
    public void run() {
        int max = MAX_COUNT / THREADS;
        System.out.printf("Thread %d\n", Thread.currentThread().getId());
        for (int i = 0; i < max; i++) {
            sc.increment();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread[] threads = new Thread[THREADS];
        int i;
        CounterMethod c = new CounterMethod();
        sc = new SynchronizedCounter();
        for (i = 0; i < THREADS; i++) {
            threads[i] = new Thread(c);
            threads[i].start();
        }

        for (i = 0; i < THREADS; i++) {
            threads[i].join();
        }
        System.out.printf("Counter value: %d Expected: %d\n", sc.value(), MAX_COUNT);
    }
}
```

```java
package pkg257_countermethod;

/**
 *
 * @author miquelmascarooliver
 */
public class SynchronizedCounter {

    private int c = 0;

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

    public synchronized int value() {
        return c;
    }
}
```

- Solució amb un monitor definit com un objecte encapsul·lat

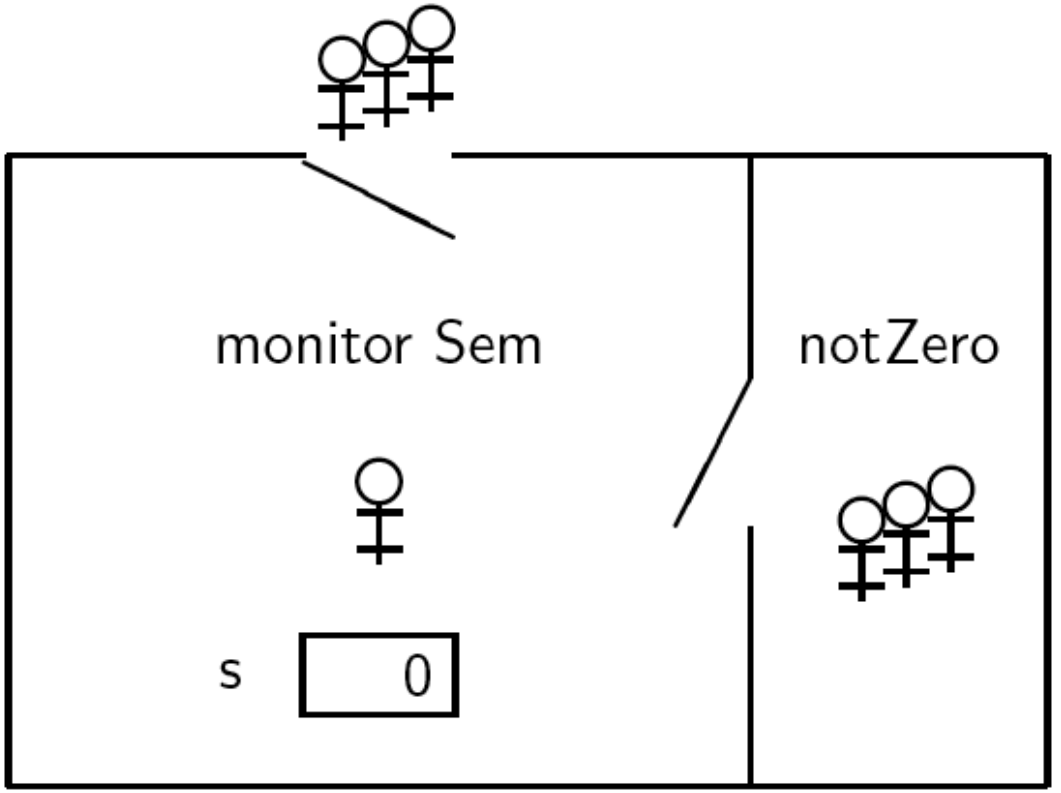
Variables de condició

- L'exclusió mútua no és suficient per a la sincronització general entre processos
- S'afegeixen dues operacions *waitC* i *signalC* que permeten bloquejar i desbloquejar processos quan es compleix una determinada condició. P.e. bloquejar els productors quan el *buffer* està ple
- Implementació de les variables de condició:
  - **Explícites**: Es declaren variables només per rebre *waitC* i *signalC*. Tenen una cua de processos bloquejats. El programa verifica les condicions i crida a les operacions. *SignalC* sobre una variable desbloqueja un procés en aquesta variable (Concurrent Pascal, C, Python, Ruby, Go…)
  - **Implícites**: Les operacions no estan lligades a cap variable. Hi ha una variable amb una única cua. Es requereixen variables d'estat (Java)
  - **Objectes protegits**: Bloqueig i desbloqueig automàtic depenent d'expressions lògiques *guards* (Ada)

- Simulació de semàfors amb var. de condició

Algorithm 7.2: Semaphore simulated with a monitor

```
monitor Sem
    integer s ← k
    condition notZero
    operation wait
        if s = 0
            waitC(notZero)
        s ← s − 1
    operation signal
        s ← s + 1
        signalC(notZero)
```



p	q
loop forever	loop forever
non-critical section	non-critical section
p1: Sem.wait	q1: Sem.wait
critical section	critical section
p2: Sem.signal	q2: Sem.signal

simulation_semaphore.c

```c
/* Simulation of semaphores with "monitors" */
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t notZero = PTHREAD_COND_INITIALIZER;
int sem_value = 1;

void p() {
    pthread_mutex_lock(&mutex);
    while (sem_value == 0) {
        pthread_cond_wait(&notZero, &mutex);
    }
    sem_value--;
    pthread_mutex_unlock(&mutex);
}

void v() {
    pthread_mutex_lock(&mutex);
    sem_value++;
    pthread_cond_signal(&notZero);
    pthread_mutex_unlock(&mutex);
}
```

```c
int counter = 0;

void *count(void *ptr) {
    long i, max = MAX_COUNT/NUM_THREADS;
    int tid = ((struct tdata *) ptr)->tid;

    for (i=0; i < max; i++) {
        p();
        counter += 1;
        v();
    }
    printf("End %d counter: %d\n", tid, counter);
}
```

- Implementació amb C: Simulació d'un semàfor usant les operacions *wait* i *signal* sobre la variable de condició

CounterMutex.java

```java
// Amb la classe Mutex es simula el semàfor
class Mutex {
    boolean lock = false;

    synchronized void lock() {
        while (lock) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        lock = true;
    }

    synchronized void unlock() {
        lock = false;
        this.notify();
    }
}
```

```java
volatile static int counter = 0;
static Mutex mutex = new Mutex();
int n, id;

public CounterMutex(int id, int n) {
    this.id = id;
    this.n = n;
}

@Override
public void run() {
    System.out.println("Thread " + Thread.currentThread().getId());
    for (int i = 0; i < this.n; i++) {
        mutex.lock();
        counter += 1;
        mutex.unlock();
    }
}
```

- Implementació en Java: Simulació d'un semàfor amb les operacions *wait* i *notify* sobre l'objecte que simula el semàfor, la variable d'estat *lock* controla l'execució de les instruccions

Objectes protegits de Ada

- Els objectes protegits d'Ada encapsulen dades i donen accés a elles només amb subprogrames protegits. Aquest accés es fa sota exclusió mútua

- Una unitat protegida es declara com un tipus, amb una especificació i un cos

- Un tipus protegit és una interfície que conté funcions, procediments i entrades (*entries*)

- Una entrada és similar a un procediment excepte que està protegit per una expressió boolean (anomenada barrera). Si l'expressió és false la tasca queda suspesa i cap altra tasca pot accedir a l'objecte protegit

def_semafors.ads

```
protected type Semafor(Inicial: Natural) is
   --Wait es la cua FIFO de processos bloquejats
   entry Wait;
   procedure Signal;
private
   -- Comptador del semafor. Var condicional
   Contador: Natural := Inicial;
end Semafor;
```

def_semafors.adb

```
protected body Semafor is
   entry Wait when Contador > 0 is
   -- Contador positiu assegura excucio de Wait
   begin
      Contador := Contador - 1;
   end Wait;

   procedure Signal is
   begin
      Contador := Contador + 1;
   end Signal;
end Semafor;
```

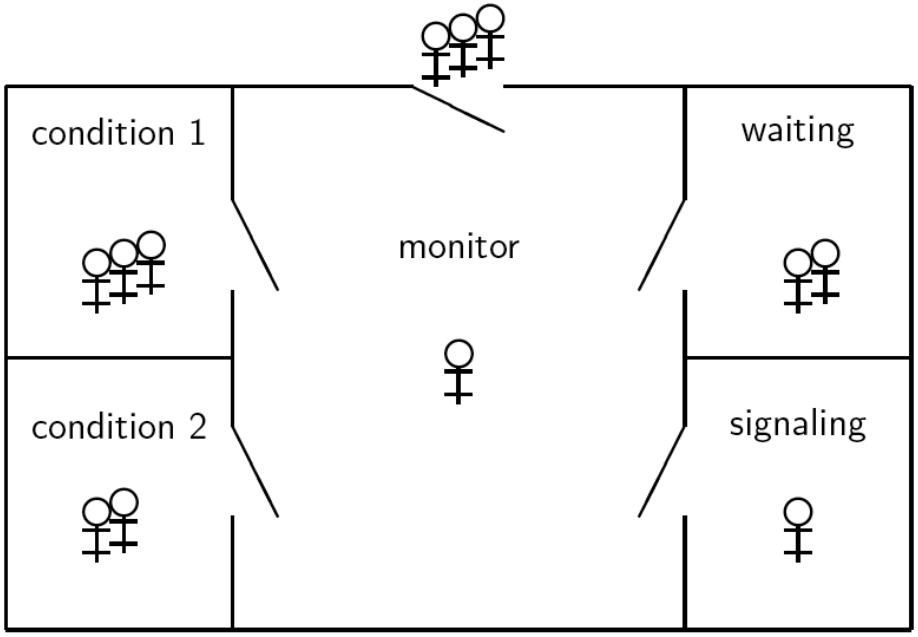
semaphore_simulation.adb

```
n: Integer := 0;
pragma Volatile(n);
S: Semafor(1);
--Sem: Counting_Semaphore(1,0);
-- El segon param. es prioritat

task type Tasca_Comptador;

task body Tasca_Comptador is
begin
   for i in 1..10000000 loop
      S.Wait;
      --Sem.Seize;
      n := n + 1;
      S.Signal;
      --Sem.Release;
   end loop;
end Tasca_Comptador;
```

https://www.adaic.org/resources/add_content/docs/95style/html/sec_6/6-1-1.html

Especificació de prioritat

- Els monitors han d'especificar les prioritats assignades als diferents processos
- *S* precedència del procés que fa el *signal*
- *W* precedència dels processos en espera
- *E* dels processos bloquejats a l'entrada



Esquema de monitor tradicional

- Hi ha tres alternatives:

1. Monitors tradicionals: El procés bloquejat a la variable de condició s'ha de reprendre immediatament (**IRR** *Immediate resumption requirement*). Els processos bloquejats a les variables de condició (*W*) són els de major prioritat, el procés que fa el signal (*S*) es bloqueja i cedeix el monitor, els que esperen a l'entrada (*E*) són els de menor prioritat

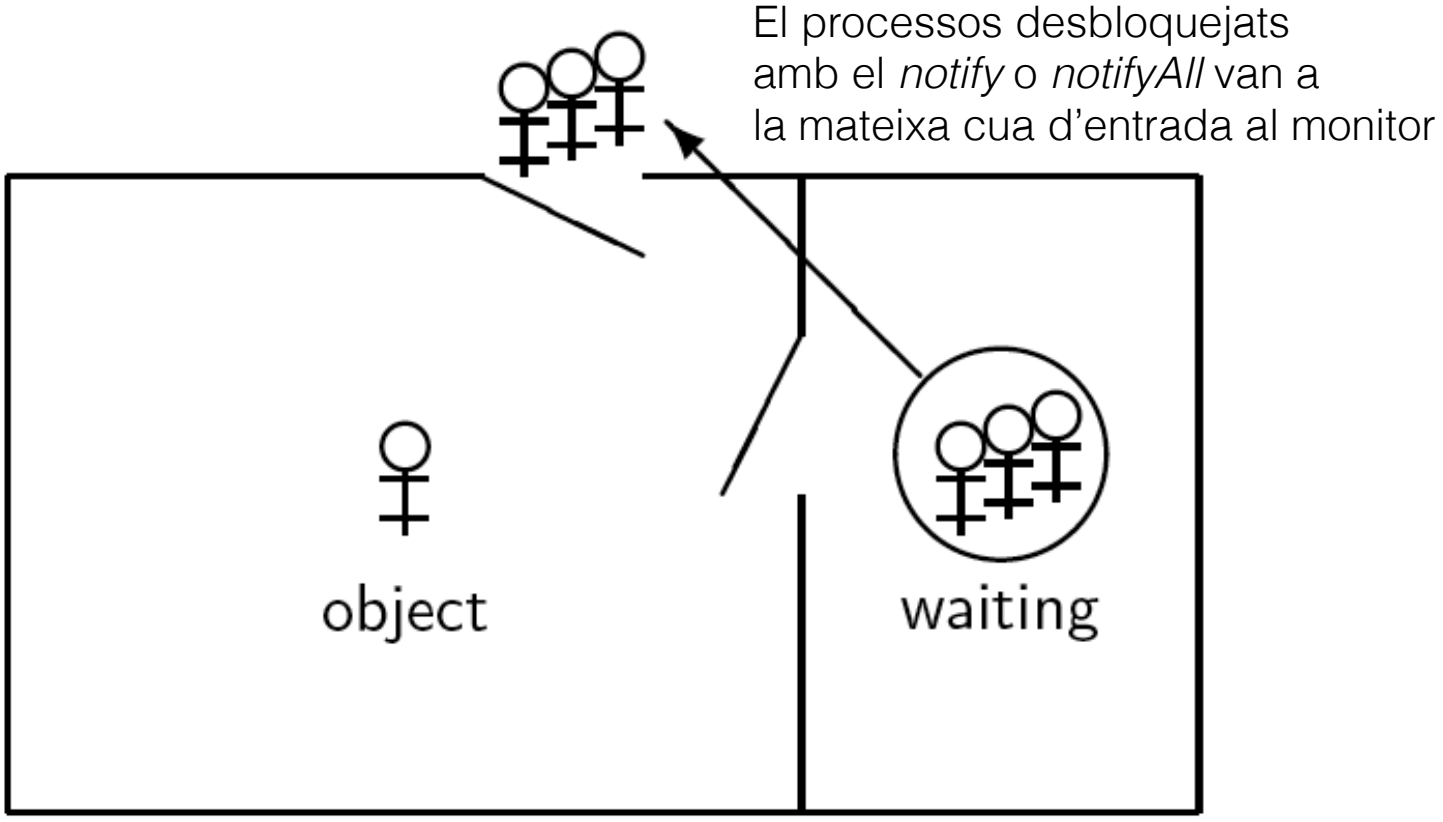
$$E < S < W$$

2. El procés que fa el *signal* surt del monitor, després s'executen els que estaven bloquejats a la variable de condició senyalitzada i finalment els que esperen a l'entrada

$$E < W < S$$

3. Monitors Java: Els processos que esperen per entrar tenen la mateixa prioritat que els bloquejats per la variable de condició

$$E = W < S$$



Esquema de monitor en Java

- Per a la correcta simulació de semàfors amb monitors es requereix la represa immediata IRR

- Quan s'executa el *signalC* el procés desbloquejat ha de ser executat immediatament per evitar que el valor sigui modificat per un altre

- Si el monitor no assegura E < S < W (cas de Java, Python i C) ha de tornar verificar la condició de despertar del *wait* (canviar l'*if* de l'algorisme per un *while*)

Semafor.java

```
class Semafor {
    int value;

    public Semafor(int v) {
        value = v;
    }

    synchronized void p() {
        while (value == 0) {
        //if (value == 0) { //Funcionament erroni IRR
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        value--;
    }

    synchronized void v() {
        value++;
        notify();
    }
}
```

SemaphoreSimulation.java

```
public class SemaphoreSimulation implements Runnable {
    static final int THREADS = 4;
    static final int MAX_COUNT = 10000000;
    volatile static int counter = 0;
    static Semafor sem = new Semafor(1);
    int n, id;

    public SemaphoreSimulation (int id, int n) {
        this.id = id;
        this.n = n;
    }

    @Override
    public void run() {
        for (int i = 0; i < this.n; i++) {
            sem.p();
            counter += 1;
            sem.v();
        }
    }
```

- Implementació en Java: Classe semàfor amb IRR

Problema de productors - consumidors

- Solució al problema dels productors - consumidors amb un *buffer* finit usant un monitor amb dues variables de condició

- *notFull*: Si el *buffer* està ple es bloqueja als productors

- *notEmpty*: Si el *buffer* està buit es bloquegen els consumidors

- Els procés del *buffer* està encapsulat a l'estructura de dades i no és visible als processos del productor i del consumidor

Algorithm 7.3: Producer-consumer (finite buffer, monitor)

```
monitor PC
    bufferType buffer ← empty
    condition notEmpty
    condition notFull
    operation append(datatype V)
        if buffer is full
            waitC(notFull)
        append(V, buffer)
        signalC(notEmpty)
    operation take()
        datatype W
        if buffer is empty
            waitC(notEmpty)
        W ← head(buffer)
        signalC(notFull)
        return W
```

Algorithm 7.3: Producer-consumer (finite buffer, monitor) (continued)

	producer		**consumer**
	datatype D		datatype D
	loop forever		loop forever
p1:	D ← produce	q1:	D ← PC.take
p2:	PC.append(D)	q2:	consume(D)

ProducerConsumerMonitor.java

```
static final int BUFFER_SIZE = 10;
static final int PRODUCERS = 2;
static final int CONSUMERS = 2;
static final int TO_CONSUME = 1000;
static final int TO_PRODUCE = 1000;

public static void main(String[] args) {
    Thread[] threads = new Thread[PRODUCERS+CONSUMERS];
    int t = 0, i;
    PCMonitor monitor = new PCMonitor(BUFFER_SIZE);
    for (i = 0; i < CONSUMERS; i++) {
        threads[t] = new Thread(new Consumer(monitor, TO_CONSUME));
        threads[t].start();
        t++;
    }
    for (i = 0; i < PRODUCERS; i++) {
        threads[t] = new Thread(new Producer(monitor, TO_PRODUCE));
        threads[t].start();
        t++;
    }
```

PCMonitor.java

```
class PCMonitor {

    int size;
    Deque<Integer> buffer = new LinkedList<>();

    public PCMonitor(int size) {
        this.size = size;
    }

    synchronized public int take() {
        Integer data;
        while (buffer.isEmpty()) {
            try {
                this.wait();
            } catch (InterruptedException e) {
            }
        }
        data = buffer.remove();
        notifyAll();
        return data;
    }

    synchronized public void append(Integer data) {
        while (buffer.size() == size) {
            try {
                this.wait();
            } catch (InterruptedException e) {
            }
        }
        buffer.add(data);
        this.notifyAll();
    }

}
```

- Implementació en Java: Productors - consumidors amb monitors

Producer.java

```
class Producer implements Runnable {
    PCMonitor monitor;
    int operations;

    public Producer(PCMonitor mon, int ops) {
        monitor = mon;
        operations = ops;
    }

    @Override
    public void run() {
        long id = Thread.currentThread().getId();
        System.out.println("Productor " + id);
        for (int i = 0; i < operations; i++ ) {
            monitor.append(i);
            System.out.println(id + " produeix: " + i);
        }
        System.out.println("Productor " + id + " ha acabat");
    }
}
```

Consumer.java

```
class Consumer implements Runnable {

    PCMonitor monitor;
    int operations;

    public Consumer(PCMonitor mon, int ops) {
        monitor = mon;
        operations = ops;
    }

    @Override
    public void run() {
        long id = Thread.currentThread().getId();
        Integer data;
        System.out.println("Consumidor " + id);
        for (int i = 0; i < operations; i++ ) {
            data = monitor.take();
            System.out.println("        " + id + " consumeix " + data);
        }
        System.out.println("          Consumidor " + id + " ha acabat");
    }

}
```

- Implementació en Java: Productors - consumidors amb monitors

- A la solució Java no es tenen variables independents i es comparteix una única cua per productors i consumidors

- *notifyAll*: Productors i consumidors verifiquen si poden continuar

- Quan un productor o consumidor executa un *notifyAll* es desperten tots els productors i consumidors malgrat només un pugui entrar!

- A la solució en Python es tenen dues variables una per bloquejar productors i l'altre consumidors

https://stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again

producerconsumermonitor.py

```python
class ProducerConsumer(object):
    def __init__(self, size):
        self.buffer = collections.deque([], size)
        self.mutex = threading.Lock()
        self.notFull = threading.Condition(self.mutex)
        self.notEmpty = threading.Condition(self.mutex)

    def append(self, data):
        with self.mutex:
            while len(self.buffer) == self.buffer.maxlen:
                self.notFull.wait()
            self.buffer.append(data)
            self.notEmpty.notify()

    def take(self):
        with self.mutex:
            while not self.buffer:
                self.notEmpty.wait()
            data = self.buffer.popleft()
            self.notFull.notify()
            return data
```

```python
def producer(buffer):
    id = threading.current_thread().name
    print("Producer {}".format(id))

    for i in range(TO_PRODUCE):
        data = "{} i: {}".format(id, i)
        buffer.append(data)
        print("        {} PRODUEIX: {}".format(id, data))

def consumer(buffer):
    id = threading.current_thread().name
    print("Consumer {}".format(id))

    for i in range(TO_PRODUCE):
        data = buffer.take()
        print("{} CONSUMEIX: {}".format(id, data))
```

- Implementació amb Python: Productors - consumidors amb monitors

https://docs.python.org/2/library/threading.html

Problema de lectors i escriptors

- Problema similar al d'exclusió mútua: diferents processos competint per accedir a la SC
- **Lectors**: Processos requerits per excloure als escriptors però no a altres lectors
- **Escriptors**: Processos requerits per excloure tant a lectors com a escriptors
- El problema és una abstracció de l'accés a BBDD: No hi ha perill per llegir concurrentment però l'escriptura s'ha de fer baix exclusió mútua per garantir la consistència

Algorithm 7.4: Readers and writers with a monitor

```
monitor RW
    integer readers ← 0
    integer writers ← 0
    condition OKtoRead, OKtoWrite
    operation StartRead
        if writers ≠ 0 or not empty(OKtoWrite)   // El lector es suspèn si un procés
            waitC(OKtoRead)                      // escriu o si espera per escriure
        readers ← readers + 1
        signalC(OKtoRead)
    operation EndRead
        readers ← readers − 1
        if readers = 0
            signalC(OKtoWrite)
```

Algorithm 7.4: Readers and writers with a monitor (continued)

```
operation StartWrite
    if writers ≠ 0 or readers ≠ 0   // L'escriptor es bloqueja si hi ha processos
        waitC(OKtoWrite)            // llegint o escrivint
    writers ← writers + 1

operation EndWrite
    writers ← writers − 1
    if empty(OKtoRead)              // Dona precedència al primer lector bloquejat
        then signalC(OKtoWrite)
        else signalC(OKtoRead)
```

reader	**writer**
p1: RW.StartRead	q1: RW.StartWrite
p2: read the database	q2: write to the database
p3: RW.EndRead	q3: RW.EndWrite

- El monitor usa 4 variables:
  - *readers*: nombre de lectors llegint la BD, després d'executar *StartRead* i abans de *EndRead*
  - *writers*: nombre d'escriptors a la BD, després d'executar *StartWrite* i abans de *EndWrite*
  - *OKtoRead*: Variable de condició per bloquejar lectors fins que sigui ok
  - *OKtoWrite*: Variable de condició per bloquejar escriptors fins que sigui ok
- Les variables *readers* i *writers* s'incrementen a *Start* i decrementen a End. En aquests les booleanes es comproven per bloquejar o desbloquejar processos

- Un lector es bloqueja si hi ha un escriptor o hi ha algun escriptor esperant (*OKtoWrite* no buit)

- Un escriptor es bloqueja només si hi ha processos llegint o escrivint

- *EndRead* executa un *signalC(OKtoWrite)* si no hi ha més lectors. Si hi ha escriptors bloquejats un s'allibera i executa *StartWrite*

- *SignalC(OKtoRead)* a *StartRead* provoca un desbloqueig en cascada als lectors bloquejats: Quan acaba un escriptor dona precedència a desbloquejar un lector sobre els escriptors

readerswriters.py

- Implementació amb Python: Lectors llegeixen un comptador i els escriptors l'incrementen

```python
counter = 0
def thread(rw):
    global counter
    id = threading.current_thread().name
    print("Thread {}".format(id))
    # El lector no modifica el comptador l'escriptor si
    for i in range(MAX_COUNT/THREADS):
        if i % 10:
            rw.reader_lock()
            c = counter
            rw.reader_unlock()
            print("        {} Llegeix: {}".format(id, counter))
        else:
            rw.writer_lock()
            counter += 1
            rw.writer_unlock()
            print("{} Incrementa: {}".format(id, counter))
```

```python
class ReaderWriter(object):
    def __init__(self):
        # Nombre de lectors a la SC
        self.readers = 0
        # Si hi ha un escriptor a la SC
        self.writing = False
        self.mutex = threading.Lock()
        # Variables de cond. (només poden wait i notify)
        self.canRead = threading.Condition(self.mutex)
        self.canWrite = threading.Condition(self.mutex)


    def reader_lock(self):
        with self.mutex:
            # Espera si hi ha escriptor
            while self.writing:
                self.canRead.wait()
            self.readers += 1
            # Per poder entrar més lectors
            self.canRead.notify()
```

```python
def reader_unlock(self):
    with self.mutex:
        self.readers -= 1
        # Si és el darrer lector desbloqueja escriptors
        if not self.readers:
            self.canWrite.notify()

def writer_lock(self):
    with self.mutex:
        # Esperia si hi ha lectors o escriptors
        while self.writing or self.readers:
            self.canWrite.wait()
        self.writing = True

def writer_unlock(self):
    with self.mutex:
        self.writing = False
        # we don't give priority to readers or writers
        # Fa el signal per lectors i escriptors
        self.canRead.notify()
        self.canWrite.notify()
```

```java
class RWMonitor {
    volatile int readers = 0;
    volatile boolean writing = false;

    synchronized void readerLock() {
        while (writing) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        readers++;
        notifyAll();
    }

    synchronized void readerUnlock() {
        readers--;
        if (readers == 0) {
            notifyAll();
        }
    }

    synchronized void writerLock() {
        while (writing || readers != 0) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        writing = true;
    }

    synchronized void writerUnlock() {
        writing = false;
        notifyAll();

    }
}
```

- Implementació amb Java: No es poden usar dues variables de condició. *NotifyAll* per desbloquejat tota cua de lectors i escriptors
- Els lectors es bloquegen quan hi ha un escriptor amb el seu *notifyAll* permeten que d'altres entrin, els escriptors també es desperten i tornen a quedar bloquejats
- Els escriptors es bloquegen si n'hi ha un altre o lectors
- No es pot saber a priori si entrarà un escriptor o lectors

def_monitor.ads

```
package def_monitor is

    protected type RWMonitor is
      entry readerLock;
      procedure readerUnlock;
      entry writerLock;
      procedure writerUnlock;
    private
      readers : integer := 0;
      writing : boolean := false;
    end RWMonitor;

end def_monitor;
```

def_monitor.adb

```
package body def_monitor is

    protected body RWMonitor is
     entry readerLock when not writing is
     begin
       readers := readers + 1;
     end readerLock;

     procedure readerUnlock is
     begin
       readers := readers - 1;
     end readerUnlock;

     entry writerLock when (readers = 0) and (not writing) is
     begin
       writing := true;
     end writerLock;

     procedure writerUnlock is
     begin
       writing := false;
     end writerUnlock;

   end RWMonitor;

end def_monitor;
```

- Implementació en Ada amb Objectes protegits

Problema del sopar dels filòsofs

- La solució amb monitors és més simple i menys propensa als errors
- Degut a l'exclusió mútua entre mètodes la verificació i manipulació de variables compartides és més senzilla
- El monitor manté un array *fork* que conté el nombre de bastonets lliures de cada filòsof
- L'operació *takeFork* espera fins que els dos bastonets estan disponibles. Abans de deixar el monitor decrementa el nombre de bastonets als veïns
- Després de menjar *releaseForks* actualitza l'array *fork* i els allibera
- *OKtoEat* és un array de variables de condició per bloquejar els filòsofs que no tenen els dos bastonets disponibles

Algorithm 7.5: Dining philosophers with a monitor

```
monitor ForkMonitor
    integer array[0..4] fork ← [2, . . . , 2]
    condition array[0..4] OKtoEat
    operation takeForks(integer i)
        if fork[i] ≠ 2
            waitC(OKtoEat[i])
        fork[i+1] ← fork[i+1] − 1
        fork[i−1] ← fork[i−1] − 1

    operation releaseForks(integer i)
        fork[i+1] ← fork[i+1] + 1
        fork[i−1] ← fork[i−1] + 1
        if fork[i+1] = 2
            signalC(OKtoEat[i+1])
        if fork[i−1] = 2
            signalC(OKtoEat[i−1])
```

Algorithm 7.5: Dining philosophers with a monitor (continued)

philosopher i

```
      loop forever
p1:       think
p2:       takeForks(i)
p3:       eat
p4:       releaseForks(i)
```

philosophers_monitor.py

```python
import threading
import time

PHILOSOPHERS = 5
EAT_COUNT = 10

class Philosopher(threading.Thread):
    mutex = threading.Lock()
    forks = [] #forks available for each philosopher
    canEat = []
    count = 0

    def __init__(self):
        super(Philosopher, self).__init__()
        self.id = Philosopher.count
        # Per tractar el darrer com els altres
        self.right = (self.id - 1) % PHILOSOPHERS
        self.left = (self.id + 1) % PHILOSOPHERS
        Philosopher.count += 1
        # forks Disponibles de esquerra i dretra
        Philosopher.forks.append(2)
        Philosopher.canEat.append(threading.Condition(Philosopher.mutex))

    def pick(self):
        with Philosopher.mutex:
            while Philosopher.forks[self.id] != 2:
                Philosopher.canEat[self.id].wait()
            Philosopher.forks[self.left] -= 1
            Philosopher.forks[self.right] -= 1

    def release(self):
        with Philosopher.mutex:
            Philosopher.forks[self.left] += 1
            Philosopher.forks[self.right] += 1
            if Philosopher.forks[self.left] == 2:
                Philosopher.canEat[self.left].notify()
            if Philosopher.forks[self.right] == 2:
                Philosopher.canEat[self.right].notify()

    def think(self):
        time.sleep(0.05)

    def eat(self):
        print("{} start eat".format(self.id))
        time.sleep(0.1)
        print("{} end eat".format(self.id))

    def run(self):
        for i in range(EAT_COUNT):
            self.think()
            self.pick()
            self.eat()
            self.release()
```

- **Interfici Condition de Java** (java.util.concurrent.locks)
  - Permeten configurar múltiples esperes per objecte
  - S'usen en combinació del *Locks*, de manera que aquests substitueixen els mètodes *syncronized* i les condicions els mètodes d'objecte monitor
  - Els mètodes *await* i *signal* s'apliquen a les condicions i els *lock* i *unlock* als *locks*

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Condition.html

PhilosopherConditions.java

```
class PhilosopherMonitor {
    final Lock lock = new ReentrantLock();
    Integer n;
    Integer forks[];
    ArrayList<Condition> canEat = new ArrayList<Condition>();

    public PhilosopherMonitor(int n) {
        this.n = n;
        forks = new Integer[n];
        Arrays.fill(forks, 2);
        for (int i = 0; i < n; i++) {
            canEat.add(lock.newCondition());
        }
    }

    int left(int i) {
        return (i + n - 1) % n;
    }

    int right(int i) {
        return (i + 1) % n;
    }
```

```
void pick(int i) {
    lock.lock();
    try {
        while (forks[i] != 2) {
            canEat.get(i).await();
        }
        forks[left(i)]--;
        forks[right(i)]--;
    } catch (InterruptedException e) {}
    finally {
        lock.unlock();
    }
}

void release(int i) {
    lock.lock();
    try {
        forks[left(i)]++;
        forks[right(i)]++;
        if (forks[left(i)] == 2) {
            canEat.get(left(i)).signal();
        }
        if (forks[right(i)] == 2) {
            canEat.get(right(i)).signal();
        }
    } finally {
        lock.unlock();
    }
}
```

- Implementació amb Java: Amb *ReentrantLock* (funcionament similar al monitor accedit amb mètodes *synchronized*)

L'ós, el pot de mel i les abelles (Andrews 2000)

- Hi ha un ós que bàsicament menja mel d'un pot i dorm. Hi ha N abelles que carregen 1 porció de mel i la duen al pot. Al pot hi caben H porcions de mel. Les abelles duen mel al pot fins que està ple. Quan el pot està ple, la darrera abella desperta l'ós. L'ós es menja el pot sencer i mentre ho fa les abelles no el molesten. Quan acaba de menjar se'n va a dormir i les abelles comencen de bell nou.