ALGORITMI CONCORRENTI PER PSC

ALGORITMO DI PETERSON PER 2 THREAD

Risorse condivise:

```
var turn : int;
var interested0 , interested1 : bool;
```

Processo P0:

```
while(TRUE) do
  interested0 := true;
  turn := 0;
  while(turn == 0 && interested1) do skip endwhile;
  #critical section
  interested0 := false;
  #non-critical section
endwhile
```

Processo P1:

```
while(TRUE) do
  interested1 := true;
  turn := 1;
  while(turn == 1 && interested0) do skip endwhile;
  #critical section
  interested1 := false;
  #non-critical section
endwhile
```

ALGORITMO DI LAMPORT PER N THREAD

```
var choosing: array[1,. . . ,N] of boolean;
var number: array[1,. . . ,N] of integer;
#Process P(i):
while (TRUE) do
  choosing [i] - TRUE ;
  number [i] = max{number [1], . . . , number [N]} + 1;
  choosing [i] = FALSE ;
  for k : 1 to N do
    while choosing [k] do skip endwhile;
    #Mi basta una sola iterazione, perché scelgo biglietti sempre più grandi
    while (number [k] != 0 and ( ⟨number [k], k⟩ << ⟨number [i], i⟩ ))
      do skip
    endwhile;
  endfor;
  critical section
  number [i] = 0;
endwhile
```

Dove ⟨a, b⟩ << ⟨c, d⟩ se e solo se a < c oppure a=c e b < d

ALGORITMI LOCK-FREE

TEST AND SET [TS] PER MUTUA ESCLUSIONE

$$TS(x,oldx) := atomico(\ oldx := x\ ;\ x := 1\ )$$

```
shared var lock = 0;

#Processo P
  bool old;
  while (true) do
    repeat
      TS(lock, old);
    until (old == 0);
  critical section
  lock := 0;
  #non-critical section
  endwhile
```

COMPARE AND SWAP [CAS] PER MUTUA ESCLUSIONE

$$CAS(x,y) := atomico(\ aux:=x;\ x:=y;\ y:=aux;\ )$$

```
shared var lock=0;
process P:
   bool old;
   while (true) do
     repeat
       old=1;
       CAS(lock, old);
     until (old==0);
     critical section
     lock:=0;
     non-critical section
   endwhile
```

SEMAFORI

SEZIONE CRITICA PER N PROCESSI

Variabili condivise:

- **var** *mutex* : *semaphore*
- inizialmente *mutex* = 1

C'è ancora starvation

Processo P_i

```
while (TRUE) {
    down(mutex);
        sezione critica
    up(mutex);
        sezione non critica
}
```

PRODUTTORE – CONSUMATORE

Variabili condivise

var mutex : *semaphore*
var empty : *semaphore*
var full : *semaphore*

inizializzate a

mutex = 1
empty=N (numero di slot)
full=0

Produttore

```
int item;
while(TRUE) do
  item=produce_item();
  down(empty);
  down(mutex);
  insert_item(item);
  up(mutex);
  up(full);
endwhile
```

Consumatore

```
int item;
while(TRUE) do
  down(full);
  down(mutex);
  item=remove_item();
  up(mutex);
  up(empty);
  consume_item(item); endwhile
```

- Mutex: protegge le strutture condivise
- empty e full: utilizzati come contatori lato produttore e consumatore
- empty è bloccante solo quando finiscono gli elementi

IMPLEMENTAZIONE SEMAFORI A LIVELLO DI KERNEL

```
procedure down(S):
  while (S≤0) skip endwhile;
  S:=S-1
```

```
procedure V(S):
  S:=S+1
```

MONITOR

PRODUTTORE – CONSUMATORE

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count – 1;
                if count = N – 1 then signal(full)
        end;
        count := 0;
end monitor;
```

Produttore:

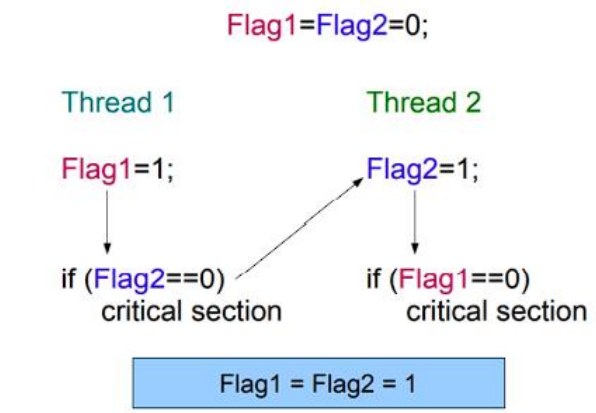
```
while (true)
        item := ...
        ProducerConsumer.insert(item)
```

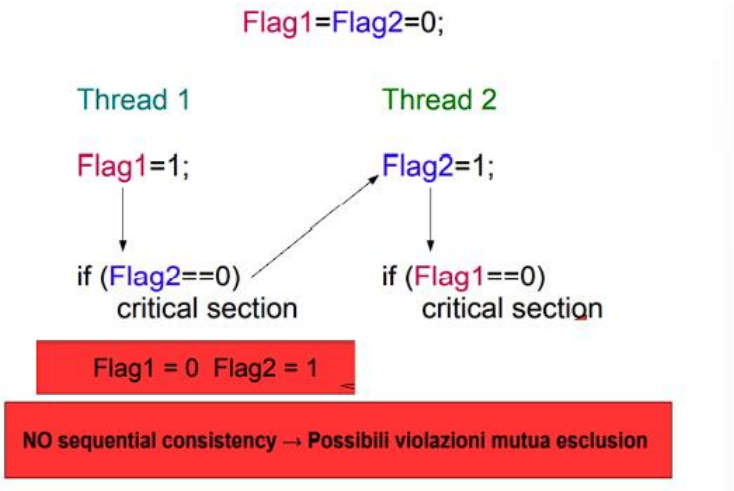
Consumatore:

```
while (true)
        item := ProducerConsumer.remove();
        ...
```

MEMORY MODEL

SEQUENTIAL CONSISTENCY [SC] – ALGORITMO DI DEKKER





THREAD IN C

PTHREAD

Creazione

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void * arg);
```

Attesa di terminazione

```
int pthread_join(pthread_t th, void *thread_return);
```

“Staccare” il thread dagli altri

```
int pthread_detach(pthread_t th);
```

Terminazione

```
void pthread_exit(void *retval);
```

PRODUTTORE – CONSUMATORE

```
struct node {
  int info;
  struct node * next;
};

/* testa lista inizialmente vuota*/
struct node * head=NULL;

/* mutex e cond var*/
pthread_mutex_t mtx=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;
```

```
static void* consumer (void*arg) {
struct node * p;
 while(true) {
     pthread_mutex_lock(&mtx);
     while (head == NULL){
     pthread_cond_wait(&cond, &mtx);
     printf(”Waken up!\n”); fflush(stdout);
 }
 p=estrai();
 pthread_mutex_unlock(&mtx);
 /* elaborazione p … … */
  }
}
```

```
static void* producer (void*arg) {
struct node * p;

for (i=0; i<N; i++) {
     p=produci();
     pthread_mutex_lock(&mtx);
     inserisci(p);
     pthread_cond_signal(&cond);
     pthread_mutex_unlock(&mtx);
 }

}
```

BARRIERE DI SINCRONIZZAZIONE

Le barriere sono un meccanismo di sincronizzazione utilizzato ad esempio nel calcolo parallelo.

Una barriera rappresenta un punto di sincronizzazione per N thread

Creazione barriera per *count* thread

```
pthread_barrier_t barrier = PTHREAD_BARRIER_INITIALIZER(count);
```

Attesa

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

BARRIERE DI MEMORIA

Forzano la write atomicity ed eliminano un possibile delay tra scrittura di una variabile e lettura successiva.

```
X=1;   // write(X,1) potrebbe essere nel buffer di scrittura
memory_fence();
if (Y==0) …          // X ha adesso valore 1 in memoria
```

THREAD IN JAVA

INTERFACCIA RUNNABLE

```
// Lanciato su più thread
if (separateThread) {
  t=new MyRunnable("thread 1");
  (new Thread(t)).start();
  // lancio un nuovo thread
}
//Funziona su un solo thread
else {
  o=new MyRunnable("only main thread");
  o.run();
  // invoco metodo run dell'oggetto t1
};
```

- **Oggetto Runnable** = Task da eseguire
- **Oggetto Thread** = Thread che esegue il task

PRODUTTORE – CONSUMATORE

```
class Produttore extends Thread {
  public void run() {
    for (int i = 1; i <= 10; ++i) {
      synchronized (cella) {
        //Funziona anche se si risveglia senza notify
        while (!cella.scrivibile) {
          try { cella.wait();
          } catch (InterruptedException e) { return;}
        }
        ++(cella.valore);
        cella.scrivibile = false; // cede il turno
        cella.notify(); // risveglia il consumatore
      }
    }
  }
}
```

```
class Consumatore extends Thread {
  public void run() {
    int valore letto;
    for (int i = 0; i < 10; ++i) {
      synchronized (cella) {
        while (cella.scrivibile) {
          try { cella.wait();
          }catch (InterruptedException e) { return; }
        }
        valore letto = cella.valore;
        cella.scrivibile = true; // cede il turno
        cella.notify(); // notifica il produttore
      }
    }
  }
}
```

EXECUTORS

- **Executors.newSingleThreadExecutor():** (single background thread)
- **Executors.newFixedThreadPool(int):** (fixed size thread pool)
- **Executors.newCachedThreadPool():** (unbounded thread pool, with automatic thread reclamation)

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class RunnableTester {
  public static void main(String[] args) {
    Task task1 = new Task("thread1");
    Task task2 = new Task("thread2");
    System.out.println("Starting threads");

    ExecutorService threadExecutor=Executors.newFixedThreadPool(2);
    threadExecutor.execute(task1);
    threadExecutor.execute(task2);
    threadExecutor.shutdown();
    System.out.println("Threads started, main ends\n");
  }
}
```

I task vengono eseguiti usando un pool con N threads.

Le richieste di esecuzione vengono gestite con una coda.

Shutdown(): interrompe la mia thread pool

FUNZIONI ASINCRONE

CALLABLE

```
// al posto di runnable
import java.util.concurrent.*;
import java.util.concurrent.Callable;

public class MyCallable implements Callable<Integer> {
        public Integer call() throws Exception {
                int result; ... return result;
        }
}
```

FUTURETASK CON CALLABLE SINGOLO

```
FutureTask task = new FutureTask (new MyCallable ());
Thread t =new Thread(task);
t.start();

try{
  int res = task.get(1000L, TimeUnit.MILLISECONDS);
}
catch(ExecutionException e) { ...}
catch(TimeoutException e){ System.out.println("tempo scaduto"); }
catch(InterruptedException e){ ... }
```

FUTURETASK CON CALLABLE ED EXECUTOR

```
FutureTask task = new FutureTask (new MyCallable ());
//FutureTask = cancellable asynchronous computation.
//A FutureTask can be used to wrap a Callable or Runnable object.
//Because FutureTask implements Runnable, a FutureTask can be submitted
//to an Executor for execution.

ExecutorService es =Executors.newSingleThreadExecutor();
es.submit (task);

try {
  int result = task.get ();
  System.out.println ("Result from task.get () = " +result);
}
catch (Exception e) {System.err.println (e);}

es.shutdown ();
```

PRODUTTORE – CONSUMATORE CON BLOCKINGQUEUE

```
//main
public class BlockingQueueExample {
        public static void main(String[] args) throws Exception {
                BlockingQueue queue = new ArrayBlockingQueue(1024);
                Producer producer = new Producer(queue);
                Consumer consumer = new Consumer(queue);
                new Thread(producer).start();
                new Thread(consumer).start();
                Thread.sleep(4000);
        }
}
```

Producer

```
        public class Producer implements Runnable{
                protected BlockingQueue queue = null;
                public Producer(BlockingQueue queue) { this.queue = queue; }
                public void run() {
                        try {
                                queue.put("1");
                                queue.put("2");
                                queue.put("3");
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
```

Consumatore

```
        public class Consumer implements Runnable {
                protected BlockingQueue queue = null;
                public Consumer(BlockingQueue queue) { this.queue = queue; }

                public void run() {
                        try {
                                var item = queue.take();
                                // consuma item
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
```

READWRITE LOCK

Variabili condivise

```
int Reader = 0;
M mutex
R mutex
```

Reader

```
//Entry
M.lock(); //Lock per reader, eventuale coda dei reader
Reader++;

//per dire che siamo il primo thread di read
if Reader == 1
  R.lock(); //di conseguenza questa sarà una coda per i writer

M.unlock();

//Exit
M.lock();
Reader--;

//Se entro vuol dire che sono l'ultimo ed ho il lock
if Reader == 0
  R.unlock();

M.unlock();
```

Writer

```
R.lock()
//----
Write(R)
//----
R.unlock()
```

SINGLE-THREAD SERVER

```
public class SingleThreadedServer implements Runnable{

    protected int          serverPort   = 8080;
    protected ServerSocket serverSocket = null;
    protected boolean      isStopped    = false;
    protected Thread       runningThread= null;

    public SingleThreadedServer(int port){
        this.serverPort = port;
    }

    public void run(){
        synchronized(this){
            this.runningThread = Thread.currentThread();
        }
        try {
            this.serverSocket = new ServerSocket(this.serverPort);
        } catch (IOException e) {
            throw new RuntimeException("Cannot open port 8080", e);
        }

        while(! isStopped()){
            Socket clientSocket = null;
            try {
                clientSocket = this.serverSocket.accept();
            } catch (IOException e) {
                if(isStopped()) {
                    System.out.println("Server Stopped.") ;
                    return;
                }
                throw new RuntimeException(
                    "Error accepting client connection", e);
            }
            try {
                processClientRequest(clientSocket);
            } catch (Exception e) {
                //log exception and go on to next request.
            }
        }

        System.out.println("Server Stopped.");
    }
```

```
    private void processClientRequest(Socket clientSocket)
    throws Exception {
        InputStream  input  = clientSocket.getInputStream();
        OutputStream output = clientSocket.getOutputStream();
        long time = System.currentTimeMillis();
        byte[] responseDocument = …
        byte[] responseHeader = …
        output.write(responseHeader);
        output.write(responseDocument);
        output.close();
        input.close();
        System.out.println("Request processed: " + time);
    }
```

```
    private synchronized boolean isStopped() {
        return this.isStopped;
    }

    public synchronized void stop(){
        this.isStopped = true;
        try {
            this.serverSocket.close();
        } catch (IOException e) {
            throw new RuntimeException("Error closing server", e);
        }
    }
}
```

MULI-THREAD SERVER

```
while(! isStopped()){
    Socket clientSocket = null;
    try {
        clientSocket = this.serverSocket.accept();
    } catch (IOException e) {
        if(isStopped()) {
            System.out.println("Server Stopped.") ;
            return;
        }
        throw new RuntimeException(
            "Error accepting client connection", e);
    }

        new Thread(
        new WorkerRunnable(
        clientSocket, "Multithreaded Server")
        ).start();

}
```

```
package servers;

import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;
import java.net.Socket;

public class WorkerRunnable implements Runnable{
    protected Socket clientSocket = null;
    protected String serverText   = null;

    public WorkerRunnable(Socket clientSocket, String serverText) {
        this.clientSocket = clientSocket;
        this.serverText   = serverText;
    }

    public void run() {
        try {
            InputStream input  = clientSocket.getInputStream();
            OutputStream output = clientSocket.getOutputStream();
            output.write("…").getBytes());
            output.close();
            input.close();
            System.out.println("Request processed: " + time);
        } catch (IOException e) {
            //report exception somewhere.
            e.printStackTrace();
        }
    }
}
```

THREADPOOL SERVER

```
while(! isStopped()){
      Socket clientSocket = null;
      try {
          clientSocket = this.serverSocket.accept();
      } catch (IOException e) {
          if(isStopped()) {
              System.out.println("Server Stopped.") ;
              break;
          }
          throw new RuntimeException(
             "Error accepting client connection", e);
      }

          this.threadPool.execute(
          new WorkerRunnable(clientSocket, "Thread Pooled Server"));

  }
```

```
public class WorkerRunnable implements Runnable{
    protected Socket clientSocket = null;
    protected String serverText   = null;

    public WorkerRunnable(Socket clientSocket, String serverText) {
        this.clientSocket = clientSocket;
        this.serverText   = serverText;
    }

    public void run() {
        try {
            InputStream input  = clientSocket.getInputStream();
            OutputStream output = clientSocket.getOutputStream();
            output.write(("…").getBytes());
            output.close();
            input.close();
            System.out.println("Request processed: " + time);
        } catch (IOException e) {
            //report exception somewhere.
            e.printStackTrace();
        }
    }
}
```

```
ThreadPooledServer server = new ThreadPooledServer(9000);
new Thread(server).start();

try {
    Thread.sleep(20 * 1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
System.out.println("Stopping Server");
server.stop();
```

ALGORITMI VECCHI ESAMI

PRODUTTORE – CONSUMATORE [RICERCA KEYWORD]

Partendo dallo schema del Produttore e Consumatore scrivere un algoritmo concorrente per gestire in maniera asincrona la ricerca di una keyword K in un insieme di pagine web con indirizzi memorizzati in un array URLs di 100 posizioni. Assumere di avere a disposizione una funzione String URLConnection(String A) per scaricare la pagina web all'indirizzo A.

```
// Variabili condivise
URL item = ... ;
var buffer: URL [100] of urls;
in, out, counter := 0;
String Key;
```

```
// Produttore
while (true) {
    Item = takeUrl();
    while (counter = n) { skip }
    URL[in] := nextp;
    in := in + 1 mod n;
    counter := counter + 1;
}
```

```
// Consumatore
while (true) {
    while (counter = 0) { skip }
    Url url := URL[out];
    Page page = URLConnection(url)
    findKey(Key, page)
    out := out + 1 mod n;
    counter := counter - 1;
}
```

SERVER MULTI-THREAD

Un sistema integrato è in grado di attivare simultaneamente la produzione di al più 5 prototipi. Un singolo prototipo viene creato tramite l'operazione crea_prototipo(file) dove file contiene il progetto dell'oggetto. Per un errore nella costruzione degli apparati, quando si superano le 5 richieste simultanee l'esecuzione di crea_prototipo(file) manda in errore il sistema con conseguente possibili guasti alle macchine. Scrivere lo schema di un server multithreaded in grado di gestire richieste multiple al sistema integrato in maniera che non vada mai in errore:

Con semaforo:

```
// Variabili condivise
var sem = 5: semaphore

// Server
while(serverIsOn()){
    Socket clientSocket = this.serverSocket.accept();//l'accept della richiesta
    // se non hai errori
    down(sem)
    new Thread(new WorkerRunnable(clientSocket)).start();
    up(sem)
}
```

Con ThreadPool:

```
public class threadPoolServer implements Runnable{
    ExecutorService executor = Executors.newFixedThreadPool(5);
    ...
    while(!isStopped()){
        Socket clientS = null;
        try { clientS = this.serverSocket.accept()} catch(Exception e) {...}
        executor.execute(new WorkerRunnable(clientS)
    }
}

public class WorkerRunnable implements Runnable{
    private Prototipo p;
    public void run() { crea_prototipo(p.file))}
}

public class Prototipo{
    public synchronized void crea_prototipo(File f){...}
}
```

PRODUTTORE – CONSUMATORE [CON BUFFER UNBOUNDED]

Scrivere una possibile soluzione (cioè un programma concorrente nel linguaggio che preferite) al problema del produttore e consumatore con buffer potenzialmente unbounded.

Strutturda dati thread safe

```
// Global variable:
var list: queue //considerando una struttura thread safe

// Produttore
while(TRUE):
    item = produce_item()
    list.append(item)
endwhile

// Consumatore
while(TRUE):
    while(list is empty) do skip endwhile
        item = list.remove()
        consume_item(item)
    endwhile
endwhile
```

Mutex

```
// Variabili condivise
type item = ... ;
var buffer: queue;
Var mutex = 1 : semaphore
Var full = 0: semaphore

// Produttore
while (true) {
    Item = produceOfItem()
    down(mutex)
    queue.add(item)
    up(full)
    up(mutex)
}

// Consumatore
while (true) {
    down(full)
    down(mutex)
    item = queue.remove();
    up(mutex)
    consumeItem(item)
}
```

ALGORITMO DI PARALELLIZZAZIONE

Sia A una matrice di dimensione MxN contenente numeri. Scrivere un algoritmo concorrente per parallelizzare, attraverso K=<N thread, il calcolo della somma dei numeri di ogni colonna calcolare quindi il valore massimo tra i valori calcolati al passo (1)

```
//SHARED DATA STRUCTURES:
int array A[M,N] = {.....}; // inizializzato
DIM=N/K //dimensione delle partizioni assumiamo N sia divisibile per K e K<=N
int array R[N]; // Somme colonne
COUNTER=0; //assumiamo sia intero con operazioni atomiche


// THREAD DEFINITIONS:
THREAD MONITOR {
    var MAX=-infinity;
    while (COUNTER < N) {};
    for J=0 to N if R[J]<MAX then MAX=R[J];
    print(MAX);
}

THREAD T[INDEX] {
    // con INDEX: 0,...,M
    var AUX=DIM*INDEX;
    var LIMIT=DIM*(INDEX+1)
    for I=0 to M do
        for J=AUX to LIMIT
            R[J]=R[J ]+A[I,J];
            COUNTER++;
}
```

NETWORKING

Supponiamo di avere a disposizione una funzione int HTTPconnect(String Address) che restituisce il tempo in millisec richiesto per aprire una connessione HTTP all'URL Address passato come parametro; -1 in caso di fallimento della richiesta. Sia A un array di N posizioni inizializzato con stringhe che rappresentano URL. Scrivere un algoritmo concorrente per parallelizzare, attraverso chiamate asincrone, la connessione ad ogni URL in A calcolare quindi il tempo medio di connessione scartando le connessioni non riuscite.

```
// Dati condivisi
int array URL[N] = {.....}; // inizializzato
DIM=N/K //dimenzione delle partizioni assumiamo N sia divisibile per K e K<=N
int array TIME[N]; tempi connessioni//
COUNTER=0; //assumiamo sia intero con operazioni atomiche

THREAD MONITOR {
    while (COUNTER < N) {};
    var AUX=0,C=0;
    for i=0 to N if TIME[i]>=0 then AUX=AUX+TIME[i]; C++;
    if (AUX>0) print(AUX/C); else print(0);
}

THREAD T[INDEX] { // con INDEX: 0,...,K
    // ogni thread esegue sequenzialmente N/K connessioni
    var AUX=DIM*INDEX;
    var LIMIT=DIM*(INDEX+1)
    for i=AUX to LIMIT do TIME[i]=HTTPconnect(A[i]); // bloccante
    COUNTER++;
}
```

THREADPOOL SENZA EXECUTORS

Descrivere una possibile implementazione in Java di un Thread Pool senza usare Executors etc.

```
public class ThreadPool {
    private BlockingQueue taskQueue = null;
    private List<PoolThread> threads = new ArrayList<PoolThread>();
    private boolean isStopped = false;

    public ThreadPool(int noOfThreads, int maxNoOfTasks){
        taskQueue = new BlockingQueue(maxNoOfTasks);
        for(int i=0; i<noOfThreads; i++)
            threads.add(new PoolThread(taskQueue));

        for(PoolThread thread : threads)
            thread.start();
}

    public void synchronized execute(Runnable task) {
        if(this.isStopped) throw new IllegalStateException("ThreadPool is stopped");
        this.taskQueue.enqueue(task);
    }

    public synchronized void stop() {
        this.isStopped = true;
        for(PoolThread thread : threads){
            thread.stop();
        }
    }
}
```

BARRIERA DI SINCRONIZZAZIONE TRAMITE SEMAFORI E CONDITION VARIABLE

Descrivere una possibile implementazione di una barriera di sincronizzazione tramite semafori e variabili Condizione.

```
// Variabili condivise
Semaphore mutex;
ConditionVariable cond,
Int N ;// number of process
Int current = 0;

sync_barrier(){
    lock(mutex)
    current++;
    if(N == current){
        Current = 0;
        signal_all(cond)
    else {
        wait(cond,mutex);
    }
    unlock(mutex)
}
```

CLOCK LOGICI – DA SCALARI A VETTORIALI

Spiegare a cosa servono i clock logici e quali vantaggi si ottengono passando da clock scalari a vettoriali.

I **clock logici** servono a definire un timestamp che consenta a tutti i nodi della rete di aver un tempo sincronizzato con gli altri nodi, questo permette inoltre di ottenere un etichetta per le azioni da svolgere e un numero di ordine che rappresenta la sequenza delle operazioni , I principali algoritmi di sincronizzazione sono infatti basati sui clock logici possono essere scalari, vettoriali e matriciali.

Nell'**algoritmo di Lamport** ogni nodo ha una variabile locale che rappresenta il suo clock logico corrente, che verrà incluso in ogni messaggio inviato. Tra due eventi il clock deve essere incrementato di almeno un tick. Quando un messaggio viene ricevuto, il clock del ricevente viene aggiornato a tempo di invio del messaggio + 1 tick se il clock locale è minore del timestamp del messaggio, altrimenti viene semplicemente incrementato di un tick. Il problema di questo algoritmo è che a -> b implica C(a) < C(b), ma C(a) < C(b) non necessariamente implica che a -> b. E' un problema perché non mi permette di capire se due eventi su nodi diversi sono concorrenti.

Clock vettoriali di Mattern e Fridge consiste In un sistema di N processi il clock logico vettoriale sarà un vettore V di dimensione N dove la posizione V[i] corrisponde al timestamp del processo i-esimo. Ogni processo manterrà una copia locale del vettore, in cui metterà tutti i valori di clock ricevuti dagli altri processi. Ogni nodo non conserva solo il clock locale ma un vettore di clock, che rappresenta uno snapshot del sistema: il timestamp degli altri nodi così com'era l'ultima volta che ne ho avuto notizia. Strategie simili sono adottate da molti algoritmi distribuiti: invio la mia conoscenza della rete agli altri nodi, in modo che ogni nodo abbia la conoscenza più vasta possibile, basata non solo su quello che può sapere direttamente ma anche su quello che sanno gli altri.

L’uso di clock vettoriali aumenta la conoscenza che i singoli processi hanno del sistema rispetto all’uso di clock scalari

LETTORI – SCRITTORI

Scrivere una possibile soluzione (cioè un programma concorrente nel linguaggio che preferite) al problema dei lettori e scrittori con priorità ai lettori rispetto agli scrittori (priorità nel caso un altro lettore sia già in sezione critica).

```
var writeMutex : mutex
var readMutex : mutex
var nLettori : 0


// Scrittore
wait(writeMutex)
writeItem()
signal(writeMutex)

// Lettore:
wait(readMutex)
nLettori += 1
if nLettori == 1:
    wait(writeMutex)
signal(readMutex)
readItem()
wait(readMutex)
nLettori -= 1
if nLettori == 0:
    signal(writeMutex)
signal(readMutex)
```

BARRIERA DI SINCRONIZZAZIONE – SEMAFORI

```
// memoria condivisa
Semaphore m = 1
Semaphore s = 0;
Mutex mutex
int N; //number of threads
int current = 0;

barrier()
    down(m)
    barrier_semaphore()
    down(m)

// Thread i
barrier_semaphore() {
    lock(mutex)
    ++current;
    if(current == N)
        current = 0
        for i to N up(s)
        up(m)
    else
        down(s)
    unlock(mutex)
}
```

```java
// Main
public GUI() {
    super("Swing");
    counter=0;
    step = new JButton("Step");
    statusLabel = new JLabel("0");
    MyListener step_handler  = new MyListener(this);
    step.addActionListener(step_handler);
    JPanel Panel = new JPanel();
    Panel.add(step);
    Panel.add(statusLabel);
    getContentPane().add(Panel);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
    pack();
}

public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        // per mantenere fin da principio l'esecuzione single thread
        // passando al EDT (Event Dispatch Thread) la gestione
        @Override
        public void run() {
            new GUI();
        }
    });
}
```

```java
// MyListener
private GUI gui;
private MyWorker worker;

public MyListener(GUI gui) {
    this.gui = gui;
}

@Override
public void actionPerformed(ActionEvent ev) {
    gui.step.setEnabled(false);
    worker = new MyWorker(gui);
    worker.execute();
}
```

```java
// MyWorker
// Lanciata da Execute()
// NON FARE UPDATE perche' crea conflitto con EDT
@Override
protected String doInBackground() throws Exception {
    Thread.sleep(1000);
    count++;
    return "Done!";
}

//delegata al EDT e thread safedone();
// Log dei messaggi intermedi
@Override
protected void process(List<Integer> chunks) {
}

//delegata al EDT e thread safe
// Da lanciare alla fine di doInBackground()
// Scrive risultato su GUI
@Override
protected void done() {
    // Possiamo anche rimuovere la riga count++ da doInBackground()
    // e aggiungere qui l'incremento, così da rimuovere la variabile locale count
    //gui.counter += 1
    gui.counter = count;
    gui.statusLabel.setText(Integer.toString(count));
    gui.step.setEnabled(true);
}
```

FUTURE

Esempio di uso di completableFuture

```
//Main {
    Runnable r1 = new myRunnable(7);
    Supplier s1 = new testSupplier(7);
    //runAsync() per task che non devono tornare valori quindi semolici runnable
    // supplyAsync() usati per ritornare valori da un task quindi si deve passare un tipo supplier
    //CompletableFuture<Void> future = CompletableFuture.runAsync(r1);
    CompletableFuture<Void> future = CompletableFuture.supplyAsync(s1);
    System.out.println(future.get());
}
```