Algoritmi esami scorsi
CAMBIATE I NOMI

1) Partendo dallo schema del Produttore e Consumatore scrivere una 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.

Svolgimento

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

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

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

2) 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.:

Sia p il numero di posti disponibili, il server multithreaded potrebbe gestire una struttura dati del genere, eseguendo l'entrata per memorizzare la richiesta che si occuperà di chiamare il metodo crea_prototipo()

CON SEMAFORO:

```
Global variable:
        var sem = 5: semaphore
Server(file):
        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:

```
tp = fixedSizeThreadPool(5)
while true:
  clientSocket = serverSocket.accept()
  tp.execute(new gestisciRichiesta(clientSocket))
```

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

Svolgimento 1

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

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

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

Svolgimento 2

```
condivisi:
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)
}
```

4) Scrivere un esempio di algoritmo lock-free per risolvere il problema della sezione critica.

TS(x,oldx) := atomico(oldx := x ; x := 1)
TS per mutua esclusione :
condivisa
shared var lock=0;

process P:
```
bool old;
while (true){
        while(!old) TS(lock, old);
        //critical section
        lock:=0;
        //non-critical section
}
```

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

Svolgimento
```
//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++;
}
```

6) 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.

Svolgimento

```
SHARED DATA STRUCTURES
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++;
}
```

7) 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();
		}
	}
}
```

alternativa) Descrivere una possibile implementazione in Java di un Thread Pool con usare Executors etc.

```
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");
}
```

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

Svolgimento

```
//condivisa
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)
```

9) 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 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.

10) 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)

Svolgimento CAPITOLO25.pdf

```
procedure scrittore;
        while(true) {
                wait(wsem);
```

```
            SCRIVI_UNITÀ;
            signal(wsem);
        }

        procedure lettore;
        while(true) {
            wait(x);
            numlettori++;
            if numlettori=1 { wait(wsem)};
             signal(x);
            //leggi unita
            wait(x);
            numlettori--;
            if numlettori=0 {signal(wsem)}
            signal(x);
        }
```

```
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)
```

11) Descrivere una possibile implementazione tramite semafori di una Barriera di Sincronizzazione

```
 // 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)
}
```

12) Partendo dall'algoritmo di Peterson per 2 thread scrivere una sua estensione per risolvere il problema della sezione critica per 4 thread usando uno schema a torneo (semifinali e finale).

13) gestione interfaccia

Listener sul click di un bottone

Worker i
        Result

        worker():
                Inizializzo i campi dell'interfaccia utilizzati

doThings():
    return qualcosa

doInBackground():
    Result = doThings()

done():
    Codice dopo la doInBackground che andrà a sfruttare l'edt, per esempio andando a mettere il valore di result in una text

15) possibile esercizio sui completable future

14) Possibile esercizio su gestione interfaccia EDT

```
//main{
        SwingUtilities.invokeLater(new MyRunnable());
}


//myRunnable {
        //esempio di un bottone
         JButton button = new JButton("Premi");
        JTextField middleResult = new JTextField(4);
        middleResult.setEditable(false);

        @Override
        Public void run(){
                        //listener al bottone che effetua una determinata azione
                button.addActionListener(new ActionListener() {
                        @Override
                        public void actionPerformed(ActionEvent arg0) {
                                button.setEnabled(false);
                                //azione da performare
                                Worker worker = new Worker(middleResult, button);
                                worker.execute();
                        }
                });
        }
}



//Worker
public class Worker extends SwingWorker<Void, Integer> {
        private JTextField middleResult;
        private JButton button;
        String Result;

        public Worker( JTextField middleResult, JButton button) {
                this.middleResult = middleResult;
                this.button = button;

        }
```

```
	@Override
	protected Void doInBackground() throws Exception {
		result = “il bottone e stato cliccato”;
		return null;
	}

	@Override
	protected void done() {
		middleResult.setText(String.valueOf(result));
		button.setEnabled(true);
	}
}
```

15) 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());
}
```

```
public class myRunnable implements Runnable {
	int n;

	public myRunnable(int n) {
		this.n = n;
	}

	@Override
	public void run()  {
		// TODO Auto-generated method stub
		System.out.println(factorial(n));
	}

	private int factorial(int n) {
		if(n <= 1) return 1;
		return n * factorial(n-1);
	}
}


public class testSupplier implements Supplier<Integer> {
	private int n = 0;

	public  testSupplier(int n) {
		this.n = n;
	}

	@Override
	public Integer get() {
		// TODO Auto-generated method stub
		System.out.println(Thread.currentThread().getName());
		return factorial(n);
	}

	private int factorial(int n) {
		if(n <= 1) return 1;
		return n * factorial(n-1);
	}


}
```