

PCD2018 - 16

CONDIVISIONE DELLO STATO E SINCRONIZZAZIONE

NATURA DEL PROBLEMA

Abbiamo visto i vari metodi con cui gestire l'accesso agli stessi dati da parte di più Threads.

Tuttavia, non tutto è modellabile con una struttura dati: a volte quello di cui abbiamo bisogno è controllare come diversi `Threads` attraversano una sezione di codice.

pcd2018.sync.SimpleCounter

```
/**
 * A simple interface to a counter.
 */
interface SimpleCounter {

    public void add();

    public int getState();
}
```

pcd2018.sync.UnsyncCounter

```
class UnsyncCounter implements SimpleCounter {
    private int state = 0;
    public void add() {
        int current = state;
        try {
            TimeUnit.MILLISECONDS.sleep(
                Math.round(Math.random() * 100));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        state = current + 1;
    }
}
```

pcd2018.sync.Incrementer

```
class Incrementer implements Callable<Boolean> {  
    private SimpleCounter counter;  
    Incrementer(SimpleCounter counter) {  
        this.counter = counter;  
    }  
    @Override  
    public Boolean call() {  
        IntStream.range(0, 10).forEach((i) -> counter.add());  
        return true;  
    }  
}
```


pcd2018.sync.RunCounter

```
ExecutorService executor = Executors.newFixedThreadPool(1);

SimpleCounter counter = new UnsyncCounter();
List<Incrementer> incs = Arrays.asList(new Incrementer(counter),
    new Incrementer(counter), new Incrementer(counter));

executor.invokeAll(incs);

System.out.println("All done. Final state: " +
    counter.getState() + " (" + (end - time) + ")");
```

SYNCRONIZED

Definizione: si dice *sezione critica* la parte di codice in cui vengono acceduti i dati condivisi.

Permettere a più Thread di trovarsi contemporaneamente nella sezione critica porta ad errori.

La soluzione è impedire a più Thread di trovarsi insieme nella sezione critica.

`synchronized` è una parola chiave che applicata ad un blocco di istruzioni impedisce che sia percorso contemporaneamente da più di un Thread.

pcd2018.sync.SyncCounter

```
class SyncCounter implements SimpleCounter {  
    private int state = 0;  
    synchronized public void add() {  
        int current = state;  
        try {  
            TimeUnit.MILLISECONDS.sleep(  
                Math.round(Math.random() * 100));  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        state = current + 1;  
    }  
}
```

`synchronized` può decorare due tipi di raggruppamenti di istruzioni:

- un blocco di istruzioni semplice { ... }
- un metodo

Tutti i blocchi sincronizzati di un oggetto condividono lo stesso `monitor lock` o `intrinsic lock`.

Quando un Thread rilascia un `monitor` uscendo da un blocco `synchronized`, stabilisce una relazione di `happens-before` fra l'azione di rilascio del lock e ogni successiva acquisizione dello stesso.

Una forma alternativa della parola chiave `synchronize` permette di esplicitare l'oggetto su cui effettuare la sincronizzazione.

```
synchronize (that) {  
}
```

permette di rendere il blocco di codice sincronizzato sul `monitor` dell'oggetto ritornato dall'espressione `that`.

```
synchronize {  
}
```

è equivalente a

```
synchronize (this) {  
}
```

Tutti i `monitor` sono `reentrant`: un `Thread` può acquisire lo stesso `monitor lock` più volte senza temere di entrare in un *deadlock* con se stesso.

pcd2018.sync.SimpleFriend

```
class SimpleFriend {  
    private final String name;  
  
    public synchronized void bow(SimpleFriend bower) {  
        System.out.format("%s: %s" + " has bowed to me!\n",  
            this.name, bower.getName());  
        bower.bowBack(this);  
    }  
  
    public synchronized void bowBack(SimpleFriend bower) {  
        System.out.format("%s: %s" + " has bowed back to me!\n",  
            this.name, bower.getName());  
    }  
}
```

pcd2018.sync.SimpleFriend

```
public class SimpleFriends {  
  
    public static void main(String[] args) {  
        final SimpleFriend alphonse = new SimpleFriend("Alphonse")  
        final SimpleFriend gaston = new SimpleFriend("Gaston");  
        new Thread(() -> alphonse.bow(gaston)).start();  
        new Thread(() -> gaston.bow(alphonse)).start();  
  
    }  
}
```



Mario Fusco

@mariofusco

Following

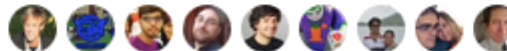


For each deadlock, race condition and multithreading problem in general there exists a simple and elegant solution. And that's the wrong one

Traduci dalla lingua originale: Inglese

08:59 - 1 set 2017

38 Retweet 81 Mi piace



4



38



81



WAIT

Un'alternativa a `synchronized` è la gestione esplicita del monitor di un oggetto.

```
/**  
 * Causes the current thread to wait until another  
 * thread invokes the notify() method or the notifyAll()  
 * method for this object.  
 */  
void wait() throws InterruptedException;
```

Per operare sul monitor dell'oggetto il Thread deve
"averlo a disposizione", cioè poter asserire la
"proprietà" dell'oggetto.

Un Thread può farlo:

- eseguendo un metodo `synchronized` dell'oggetto
- eseguendo un blocco `synchronized` all'interno dell'oggetto
- se l'oggetto una `Class`, eseguendone un metodo `synchronized static`

```
/**  
 * Wakes up a single thread that is waiting on this  
 * object's monitor.  
 */  
void notify();
```

```
/**  
 * Wakes up all threads that are waiting on this  
 * object's monitor.  
 */  
void notifyAll();
```

pcd2018.sync.Named

```
class Named {  
    public final String name;  
    private boolean red = false;  
  
    Named(String name) {  
        this.name = name;  
    }  
}
```

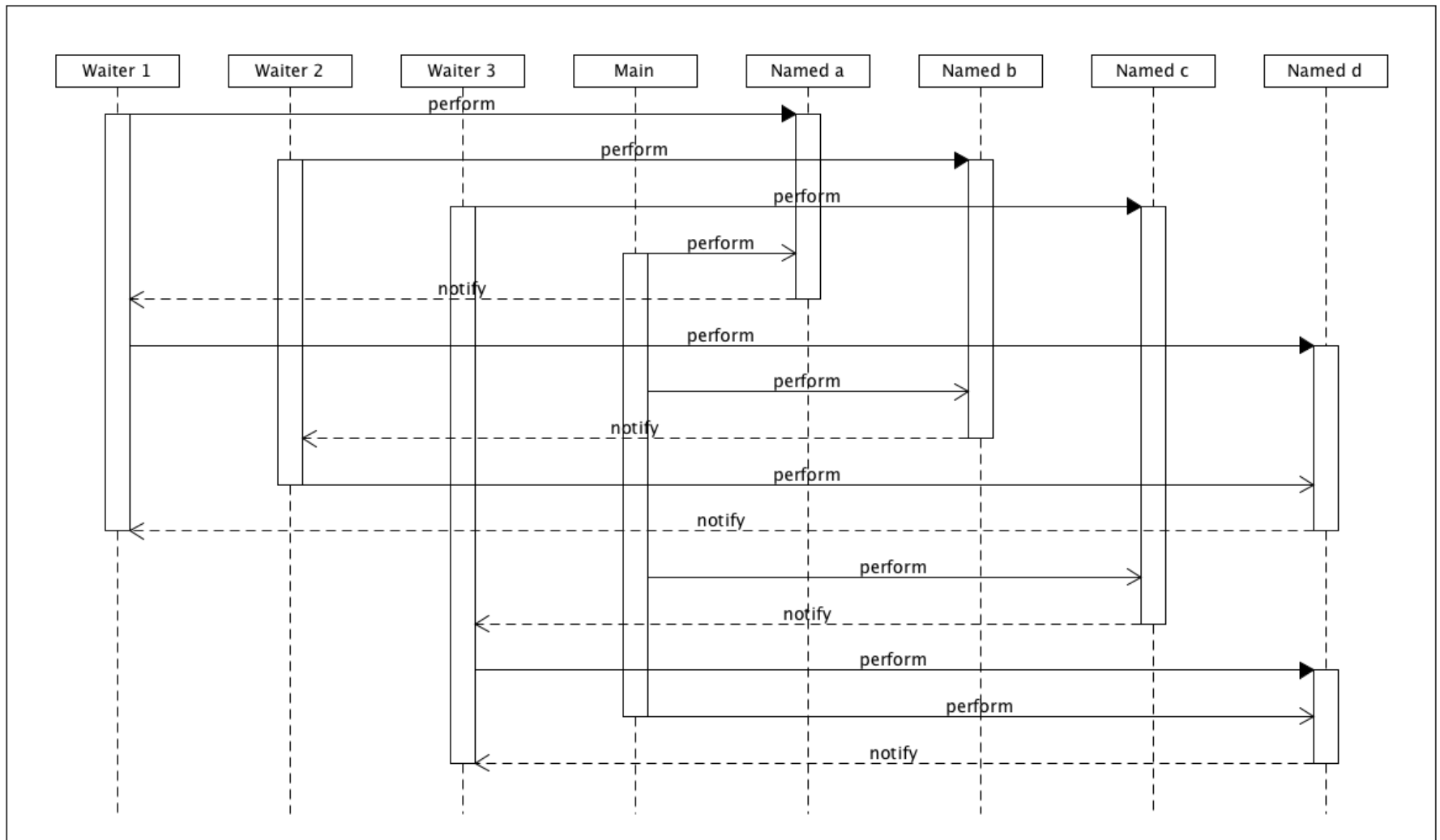
```
synchronized void perform() throws InterruptedException {  
    if (!red) {  
        red = true;  
        this.wait();  
    } else {  
        red = false;  
        this.notify();  
    }  
}
```


pcd2018.sync.Waiter

```
class Waiter implements Runnable {  
    private final Named first, second;  
  
    Waiter(Named first, Named second) {  
        this.first = first;  
        this.second = second;  
    }  
}
```

```
var thread = Thread.currentThread().getName();
System.out.println(thread + " waiting on " + first.name);
String doing = first.name;
try {
    first.perform();
    System.out.println(thread + " signalled on " + first.name);
    System.out.println(thread + " waiting on " + second.name);
    doing = second.name;
    second.perform();
} catch (InterruptedException e) {
    System.out.println(thread + " interrupted on " + doing);
}
System.out.println(thread + " signalled on " + second.name);
```

```
Named a = new Named("a"), b = new Named("b"),  
    c = new Named("c"), d = new Named("d");  
new Thread(new Waiter(a, d), "Waiter 1").start();  
new Thread(new Waiter(b, d), "Waiter 2").start();  
new Thread(new Waiter(c, d), "Waiter 3").start();  
try {  
    a.perform();  
    b.perform();  
    c.perform();  
    d.performAll();  
} catch (InterruptedException e) { e.printStackTrace(); }
```



Attenzione: la documentazione avvisa esplicitamente
che un thread può essere svegliato da un `wait()`
senza nessun `notify()`.

Viene detto "Spurious wakeup".

LOCKS

`synchronized` crea un blocco implicito.

`wait ()` ci costringe a gestire lo stato del blocco.

A volte abbiamo bisogno di controllare esplicitamente le condizioni di blocco e sblocco della sezione critica.

```
/**  
 * Lock implementations provide more extensive locking  
 * operations than can be obtained using synchronized  
 * methods and statements.  
 *  
 */  
public interface Lock;
```



```
/**  
 * Acquires the lock.  
 *  
 */  
void lock();
```

```
/**  
 * Releases the lock.  
 *  
 */  
void unlock();
```

```
/**  
 * Acquires the lock only if it is free at the time  
 * of invocation.  
 *  
 */  
boolean tryLock();
```

pcd2018.sync.LockedFriend

```
class LockedFriend {  
    private final String name;  
    private final Lock lock = new ReentrantLock();  
  
    public LockedFriend(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return this.name;  
    }  
}
```

```
public boolean impendingBow(LockedFriend bower) {  
    boolean myLock = false, yourLock = false;  
    try {  
        myLock = lock.tryLock();  
        yourLock = bower.lock.tryLock();  
    } finally {  
        if (!(myLock && yourLock)) {  
            if (myLock) { lock.unlock();}  
            if (yourLock) { bower.lock.unlock(); }  
        }  
    }  
    return myLock && yourLock;  
}
```

```
public void bow(LockedFriend bower) {  
    if (impendingBow(bower)) {  
        try {  
            System.out.format("%s: %s has" + " bowed to me!%n",  
                this.name, bower.getName());  
            bower.bowBack(this);  
        } finally { lock.unlock(); bower.lock.unlock(); }  
    } else {  
        System.out.format("%s: %s started" + " to bow to me,  
            but saw that" + " I was already bowing to" + " him.%n",  
            this.name, bower.getName());  
    }  
}
```

```
public static void main(String[] args) {  
    final LockedFriend alphonse = new LockedFriend("Alphonse");  
    final LockedFriend gaston = new LockedFriend("Gaston");  
    new Thread(new BowLoop(alphonse, gaston)).start();  
    new Thread(new BowLoop(gaston, alphonse)).start();  
}
```

PRODUCER AND CONSUMER

Producer/Consumer: pattern architetturale che modella un insieme di thread divisi in due gruppi

- *Producers*: threads che producono dati da elaborare
- *Consumers*: threads che elaborano i dati prodotti

Con un `Lock` possiamo controllare manualmente una sezione critica.

Il Consumatore ammette un solo `Thread` alla volta nella sezione critica.

Il `Lock`, implicitamente, gestisce la coda dei Produttori in attesa.

```
/**  
 * Creates an instance of ReentrantLock with the given  
 * fairness policy.  
 *  
 */  
public ReentrantLock(boolean fair)
```

Un `ReentrantLock` ha caratteristiche equivalenti ad un `implicit lock`, ma può essere controllato manualmente (con le responsabilità che ne derivano).

Se viene costruito come `fair` il `Thread` che riceve il lock è sempre quello che ha aspettato di più.

Si riduce il rischio di `starvation` a prezzo di una prestazione inferiore (dovuta al maggior costo di mantenere una gestire una lista ordinata).

```
public class PrintQueue implements Printer {  
    private final Lock queueLock = new ReentrantLock();  
  
    public void printJob(Object document) {  
        queueLock.lock();  
        try {  
            Long duration = (long) (Math.random() * 10000);  
            Thread.sleep(duration);  
        } catch (InterruptedException e) { e.printStackTrace();  
        } finally { queueLock.unlock(); }  
    }  
}
```

```
public static void main(String args[]) {  
    PrintQueue printQueue = new PrintQueue();  
    Thread thread[] = new Thread[10];  
    for (int i = 0; i < 10; i++) {  
        thread[i] = new Thread(new Job(printQueue),  
                                "Thread " + i);  
    }  
    for (int i = 0; i < 10; i++) { thread[i].start(); }  
}
```


CONDITIONS

A volte, non tutti i Thread che cercano di acquisire un lock sono uguali: possono avere semantiche diverse e necessitare di essere segnalati in condizioni differenti.

Una `Condition` permette di separare l'accodamento in attesa dal possesso del lock che controlla l'attesa.

Lo scopo è da poter gestire, su di un solo lock, di più condizioni di attesa distinte.

```
/**  
 * Returns a new Condition instance that is bound to this  
 * Lock instance.  
 *  
 */  
public Condition newCondition()
```

Ciascuna `Condition` di uno stesso lock consente di gestire un insieme distinto di Thread in attesa.

```
/**  
 * Causes the current thread to wait until it is signalled  
 * or interrupted.  
 *  
 */  
public void await()
```

```
/**  
 * Wakes up one waiting thread.  
 *  
 */  
public void signal()
```

```
/**  
 * Wakes up all waiting threads.  
 *  
 */  
public void signalAll()
```


pcd2018.sync.CharSource

```
/**
 * Bounded, non thread-safe random source of characters
 */
class CharSource {

    public boolean hasMoreLines()

    public Optional<String> getLine()
```

pcd2018.sync.Buffer

```
class Buffer {
    private LinkedList<String> buffer;
    private int maxSize;
    private ReentrantLock lock;
    private Condition lines, space;
    private boolean pendingLines;

    public Buffer(int maxSize) {
        this.maxSize = maxSize; pendingLines = true;
        buffer = new LinkedList<>();
        lock = new ReentrantLock();
        lines = lock.newCondition();
        space = lock.newCondition();
    }
}
```

```
public void insert(String line) {  
    lock.lock();  
    try {  
        while (buffer.size() == maxSize) { space.await(); }  
        buffer.offer(line);  
        System.out.printf("%s: Inserted Line: %d\n",  
            Thread.currentThread().getName(), buffer.size());  
        lines.signalAll();  
    } catch (InterruptedException e) { e.printStackTrace(); }  
    finally { lock.unlock(); }  
}
```

```
public Optional<String> get() {  
    Optional<String> line = Optional.empty();  
    lock.lock();  
    try {  
        while ((buffer.size() == 0) && (hasPendingLines())) {  
            lines.await(); }  
        if (hasPendingLines()) {  
            line = Optional.ofNullable(buffer.poll());  
            space.signalAll(); }  
    } catch (InterruptedException e) { e.printStackTrace(); }  
    finally { lock.unlock(); }  
    return line;  
}
```

```
public boolean hasPendingLines() {  
    return pendingLines || buffer.size() > 0;  
}  
  
public void setPendingLines(boolean pendingLines) {  
    this.pendingLines = pendingLines;  
}
```

pcd2018.sync.Producer

```
class Producer implements Runnable {  
    private CharSource source;  
    private Buffer buffer;  
  
    @Override  
    public void run() {  
        buffer.setPendingLines(true);  
        while (source.hasMoreLines())  
            source.getLine().ifPresent((line) -> {  
                buffer.insert(line); randomWait(50);  
            });  
        buffer.setPendingLines(false);  
    }  
}
```

pcd2018.sync.Consumer

```
class Consumer implements Runnable {  
    private Buffer buffer;  
  
    @Override  
    public void run() {  
        while (buffer.hasPendingLines())  
            buffer.get().ifPresent((line) -> process(line));  
    }  
  
    private void process(String line) {  
        LockedBuffer.randomWait(250);  
    }  
}
```

pcd2018.sync.LockedBuffer

```
public static void main(String[] args) {
    CharSource source = new CharSource(100, 100);
    Buffer buffer = new Buffer(20);
    Thread producer = new Thread(new Producer(source, buffer),
        "producer");
    Thread[] consumers = new Thread[] {
        new Thread(new Consumer(buffer)),
        new Thread(new Consumer(buffer)),
        new Thread(new Consumer(buffer)) };
    producer.start();
    for (Thread t : consumers) t.start();
}
```


Come sempre succede, da grandi poteri derivano
grandi responsabilità.

Maneggiando direttamente i Lock si chiede al sistema
di delegarci un notevole potere, ed insieme ne
riceviamo una corrispondente responsabilità.

SALTATE UN SIGNALO

DEADLOCK

UN AWAITO DI TROPPO

DEADLOCK

imgflip.com

ECCEZIONE NON GESTITA

DEADLOCK

imgflip.com

SEMAPHORES

Per controllare l'accesso ad un insieme omogeneo di risorse, si usa un *semaforo*.

Un semaforo è simile ad un lock, ma tiene un conteggio invece di un semplice stato libero/occupato.

```
/**  
 * Creates a Semaphore with the given number of permits and  
 * the given fairness setting.  
 *  
 */  
public Semaphore(int permits, boolean fair)
```

Se inizializzato come `fair`, l'ordinamento dei Thread in attesa è garantito FIFO. Altrimenti non è garantito.

Il costo è giustificato quando il semaforo regola l'accesso ad un insieme di risorse. In caso di un uso diverso, un semaforo non `fair` è molto più efficiente.

Ad ogni acquisizione il numero di "permessi"
disponibili diminuisce.

Ad ogni "rilascio" il numero viene aumentato.

```
/**  
 * Acquires a permit from this semaphore, blocking until one  
 * is available, or the thread is interrupted.  
 *  
 */  
public void acquire()
```

```
/**
 * Acquires the given number of permits from this semaphore,
 * blocking until all are available, or the thread is
 * interrupted.
 *
 * @param the number of permits to acquire
 */
public void acquire(int permits)
```

```
/**  
 * Releases a permit, returning it to the semaphore.  
 *  
 */  
public void release()
```

```
/**  
 * Releases the given number of permits, returning them to  
 * the semaphore.  
 *  
 * @param the number of permits to release  
 */  
public void release(int permits)
```

Il valore iniziale del semaforo non è un limite: può essere superato, e può essere anche negativo inizialmente.

Mantenere la coerenza semantica sta all'utilizzatore.

A differenza di un lock, un semaphore può essere rilasciato da un Thread diverso da quello che lo ha acquisito.

```
/**  
 * Shrinks the number of available permits by the  
 * indicated reduction.  
 *  
 */  
protected void reducePermits(int reduction)
```


La maggior parte dei metodi di Semaphore

- può lanciare `InterruptedException` se il thread viene interrotto durante l'attesa
- lancia `IllegalArgumentException` se il parametro è negativo

```
/**  
 * Acquires a permit from this semaphore, only if one is  
 * available at the time of invocation.  
 *  
 */  
public boolean tryAcquire()
```

```
/**  
 * Acquires the given number of permits from this semaphore,  
 * if all become available within the given waiting time and  
 * the current thread has not been interrupted.  
 *  
 */  
public boolean tryAcquire(int permits, long timeout,  
    TimeUnit unit)
```

`tryAcquire` ritorna immediatamente, con risultato falso se non ha ottenuto un permesso.

E' in grado di violare la *fairness* del semaforo

pcd2018.sync.MultiPrintQueue

```
class MultiPrintQueue implements Printer {  
    private Semaphore semaphore;  
    private boolean[] freePrinters;  
    private ReentrantLock lockPrinters;  
  
    public MultiPrintQueue() {  
        semaphore = new Semaphore(3);  
        freePrinters = new boolean[] { true, true, true };  
        lockPrinters = new ReentrantLock();  
    }  
}
```

```
public void printJob(Object document) {  
    try {  
        semaphore.acquire();  
        int assignedPrinter = getPrinter();  
        Long duration = (long) (Math.random() * 10000);  
        TimeUnit.MILLISECONDS.sleep(duration);  
        freePrinters[assignedPrinter] = true;  
    } catch (InterruptedException e) { e.printStackTrace(); }  
    finally { semaphore.release(); }  
}
```

```
int getPrinter() {  
    int res = -1;  
    try {  
        lockPrinters.lock();  
        for (int i = 0; i < freePrinters.length; i++) {  
            if (freePrinters[i]) {  
                res = i; freePrinters[i] = false;  
                break; }  
        }  
    } catch (Exception e) { e.printStackTrace();  
    } finally { lockPrinters.unlock();  
    }  
    return res;  
}
```