# Sistemi 10

Parliamo dell’implementazione dei lock.

Come si possono costruire e come possiamo confrontare implementazioni diverse? Dobbiamo considerare la mutua esclusione (un mutex che non ci garantisce quella non serve a nulla), poi la fairness/starvation; cioè se ci sono m thread che vogliono acquisire il mutex l’implementazione permetterà a tutti e tre di acquisire, prima o poi, il mutex (senza casi di starvation)?  
L’ultimo punto da considerare è la performance: l’introduzione di mutex produce un overhead, quindi è meglio che questo sia abbastanza piccolo.

Una possibile implementazione:

void lock() {

DisableInterrupts();

}

Void unlock() {

EnableInterrupts();

}

Non funzionerà bene, anche ammettendo che al processo venga permesso di disabilitare gli interrupt, questa soluzione è veramente brutale visto che il processo si prenderebbe tutta la cpu per quanto vuole. Inoltre non funziona neanche, perché sui sistemi multiprocessori non ha senso disabilitare gli interrupt su un core.   
Proviamo a usare solo codice utente:

typedef \_\_­lock\_t {int flags} lock\_t;

void init(lock\_t \*mutex) {

mutex->flag = 0;

}

Void lock(lock\_t \*mutex) {

While(mutex->flags == 1) ;

Mutex->flags = 1;

}

Void unlock(lock\_t \*mutex) {

Mutex->flag = 0;

}

Questo genera un problema di corse critiche. Tra quando ci si accorge che mutex->flags != 1 a quando lo si imposta a 1 passa del tempo: questa porzione del “lock” non è atomica. Può dunque capitare che un processo venga deschedulato tra la prima e la seconda istruzione, con conseguenza che quando verrà rischedulato ci saranno due processi col lock a 1.

Se non abbiamo un supporto hardware che permetta di effettuare atomicamente confronto e setto non se ne può uscire: diversi processori, per questo motivo, forniscono delle primitive atomiche che fanno più cose a tale scopo. Alcuni esempi sono:

* Test-and-Set (che fanno le due istruzioni del while(); e lock = 1; in un’unica istruzione)
* Scambi atomici

Su x86 abbiamo l’istruzione XCGH che corrisponde a:

int AtomicExcanghe(int \*ptr, int new) {

int old = \*ptr;

\*ptr = new;

return old;

} //questo permette l’implementazione di uno Spin-Lock.

Il nuovo lock sarà:

Void lock(lock\_t \*lock) {

Wjile(AtomicExchange (&lock->is\_locked, 1) == 1);

}

Void unlock(lock\_t \* lock) {

Lock->is\_locked = 0;

}

Questo codice funziona. Il problema di questa implementazione, però, è che se un thread ha già acquisito un lock e richiama la lock, allora blocca lì tutto il programma (e gli altri thread).

Questa implementazione garantisce la mutua esclusione. Ma non è fair, non c’è alcuna garanzia che un thread prima o poi riceva il lock.  
Per quanto riguarda la performance, si sta usando il busy waiting per aspettare che un lock venga rilasciato (se un thread sta per rilasciare il lock ma viene deschedulato per mettere sulla cpu un altro che sta aspettando è problematico) -> non ha senso su dispositivi a unico core, mentre può averlo su cpu multi-core (se si hanno più core è più conveniente il busy waiting che scomodare il kernel per fare un altro context-switch, specie se l’attesa che ci si aspetta è relativamente corta: ciò sarebbe lo standard nei programmi scritti bene).

Sono stati proposti, negli anni, anche approcci ibridi, in cui c’è una prima fase di busy waiting che porta però a un descheduling anticipato.

Supponiamo di avere uno scheduler a priorità e degli spin-lock (dove l’attesa è attiva). Abbiamo T1 che è un processo a bassa priorità e T2 è ad alta priorità. Entrambi hanno bisogno dello stesso lock. Supponiamo che T2 sia in attesa di qualcosa e quindi T1 vada in esecuzione prima, quindi T1 acquisisce il lock. T2 diventa ready, quindi lo scheduler deschedula T1 e manda in esecuzione T2. T2 inizia ad aspettare per il lock, aspetta e continua ad aspettare perché T1 non è in esecuzione e non rilascerà mai il lock. Game Over.

Ci sono modi per risolvere il problema, per esempio tramite “l’eredità di priorità” di un thread in attesa (se maggiore di quella corrente): ciò è stato usato per il NASA pathfinder.

Vediamo quali possono essere dei bug tipici, grazie al fatto di usare il multi-threading. Il primo è la violazione dell’atomicità:

struct foo \*p

void f() {

if(p != NULL) {

p->bar = 3;

}

}

Il problema qui è che il controllo e l’uso del puntatore non è atomico (p potrebbe essere reso null da un thread schedulato in mezzo all’istruzione dell’if e all’assegnazione).

Altro esempio:

glob\_thread = NULL;

void init() {

…

Glob\_thread = create\_thread(thread\_func, …);

…

}

Void thread\_func() {

Foo = glob\_thread-> bar;

…

}

Il problema qui è che non sappiamo quale sia l’ordine di scheduling del thread (rispetto al thread principale), la lettura di Foo = glob\_thread->bar potrebbe avvenire prima dell’assegnazione a glob\_thread nel thread “parent”, risultando in una lettura di NULL.

Infine, vediamo la situazione del DEADLOCK. Supponiamo che ci siano due risorse diverse, protette da due mutex diversi e supponiamo che ci siano due thread che hanno bisogno di entrambe:

Thread-A lock(m1); lock(m2);

Thread-B lock(m2); lock(m1);

Cosa può succedere? Entrambi eseguono il primo lock e poi si bloccano alla seconda istruzione, perché nessuno dei due lascerà mai l’altro lock: si ha un Deadlock.

Le condizioni per ottenere il deadlock sono:

* la mutua esclusione
* l’hold and wait: se un thread vuole acquisire nuovi lock, non deve rilasciare quelli che aveva già
* no preemption: le risorse non possono essere tolte (“da sotto i piedi” al processo)
* attesa circolare: ci deve essere un ciclo nelle attese

Come prevenirlo? Sicuramente non si possono togliere la mutua esclusione e la no preemption. Quindi le possibilità sono:

Imporre ai thread di prendere tutti i lock sulle risorse contemporaneamente e atomicamente. Dopo aver preso il lock su m1, lo si molla e poi lo si prende sia su m1 che su m2 (no hold-and-wait, è complicato da realizzare, perché serve mantenere tutte le risorse in uno stato consistente durante il mollo e la ripresa di un lock, inoltre è necessaria un idea almeno generale delle risorse che servono globalmente nel sistema in un dato momento).

Imporre un ordine sui lock e acquisirli in ordine (così non ci può essere circular wait). Questa seconda opzione è il metodo più semplice nei sistemi reali per evitare il dead lock.

Nota a parte: oltre all’address sanitizer esiste il thread sanitizer.

Passiamo al prossimo argomento. Parliamo di sicurezza.

Il sistema operativo costituisce le fondamenta di ogni applicativo: non possiamo costruire sistemi sicuri se quello che c’è sotto non lo è (e non possiamo realizzare sistemi operativi sicuri se l’hardware non lo è ecc.)

In generale quando si va a parlare di sicurezza c’è l’idea dell’anello debole della catena: se un attaccante non è stupido, andrà ad attaccare la componente più debole.

Cosa vuol dire poi, sicurezza? In generale, si parla di tre obiettivi (a livello di Computer Security):

Confidenzialità/segretezza, Integrità e Disponibilità. In generale, queste sono ottenute tramite la condivisione controllata: per esempio, potremmo voler fare leggere un nostro file ad alcuni utenti ma non ad altri.

Come avvengono questi controlli a livello di s.o? Quando un processo effettua una system call il kernel va a valutare se è sensata (cioè se il numero della call è valido) e se i parametri che utilizza sono validi (cioè sono nello spazio di indirizzamento del processo). Poi controlla che la richiesta rispetti la politica di sicurezza:

* l’entità che fa la richiesta è chiamato principal/subject (il nostro processo)
* le richieste sono relative a una risorsa, un object
* tipicamente la richiesta prevederà una modalità di accesso, per esempio in lettura o in scrittura.

Per valutare se eseguire o meno l’operazione va quindi valutato il contesto: in alcuni casi (come in Linux) tutti i processi avviati da un utente hanno gli stessi permessi, mentre in altri (in Android), c’è molto più isolamento tra un programma e un altro. In questo mondo, windows sta un po’ in mezzo (distingue tra processi di serie A e di serie B, i browser per esempio sono considerati più sfigati, essendo più spesso vittima di attacchi).