Meccanismi di sincronizzazione

Con la programmazione concorrente sorge il problema di accedere a risorse condivise o a sezioni critiche in modo consistente e senza perdita di dati. I meccanismi di sincronizzazione servono proprio a ciò, per evitare problemi quali:

* Race conditions: si presenta quando il risultato finale dell'esecuzione dei processi dipende dalla temporizzazione o dalla sequenza con cui vengono eseguiti.
* Deadlocks: indica una situazione in cui due o più processi si bloccano a vicenda, aspettando che uno esegua una certa azione che serve all'altro e viceversa.
* Resource starvation: si intende l'impossibilità perpetua, da parte di un processo, di ottenere le risorse di cui necessita per essere eseguito.

Meccanismi di sincronizzazione in OS161

Spinlock

OS161 usa gli spinlock come meccanismo di sincronizzazione. I lock permettono di eseguire la sezione critica impedendo al thread di essere interrotto, in questo modo si evita che altri thread vadano in esecuzione prima che il thread già in esecuzione abbia finito di eseguire la sezione critica, impedendo, così, la lettura o scrittura di dati inconsistenti. Per fare ciò si utilizza un’istruzione hardware atomica chiamata test and set, questa istruzione legge il valore di una variabile boolena, lo imposta a true e ritorna il valore vecchio. Questa istruzione è utile per l’acquisizione del lock: se il lock è già stato acquisito i thread che lo richiedono vengono messi in attesa, la test and set trova il valore della variabile impostato a true, lo reimposta a true e ritorna il vecchio valore che era sempre true, quindi i thread restano sempre in attesa finchè il lock non viene rilasciato e la variabile viene impostata a false. Quando il lock viene rilasciato il thread che vuole acquisirlo chiama la test and set sulla variabile che adesso sarà false, questa istruzione modifica il valore della variabile a true e ritorna il vecchio valore, cioè false, permettendo così al thread che l’ha chiamata di uscire dall’attesa, acquisire il lock e accedere alla sezione critica.

Di seguito è riportato il codice dell’implementazione degli spinlock in OS161. La funzione testandset di questo sistema non usa una variabile booleana, ma imposta la variabile a 1 e ritorna il vecchio valore.

La struct spinlock è la seguente:

struct spinlock {

volatile spinlock\_data\_t splk\_lock; // Contiene 0 se il lock è libero, 1 se è acquisito

struct cpu \*splk\_holder; // CPU che possiede il lock.

HANGMAN\_LOCKABLE(splk\_hangman); // Deadlock detector.

};

Acquisizione dello spinlock:

void spinlock\_acquire(struct spinlock \*splk)

{

//Vengono disabilitati gli interrupt per evitare deadlock

struct cpu \*mycpu;

splraise(IPL\_NONE, IPL\_HIGH);

/\* this must work before curcpu initialization \*/

if (CURCPU\_EXISTS()) {

mycpu = curcpu->c\_self;

if (splk->splk\_holder == mycpu) {

panic("Deadlock on spinlock %p\n", splk);

}

mycpu->c\_spinlocks++;

HANGMAN\_WAIT(&curcpu->c\_hangman, &splk->splk\_hangman);

}

else {

mycpu = NULL;

}

while (1) {

if (spinlock\_data\_get(&splk->splk\_lock) != 0) { //ottiene il valore del lock, se è

continue; // diverso da 0 resta in attesa

}

if (spinlock\_data\_testandset(&splk->splk\_lock) != 0) { //chiama la test and set,

continue; //se è diverso da 0 il lock è già

} // acquisito quindi resta in

break; // attesa, altrimenti lo acquisisce

}

membar\_store\_any();

splk->splk\_holder = mycpu; //Imposta la cpu corrente come holder del lock

if (CURCPU\_EXISTS()) {

HANGMAN\_ACQUIRE(&curcpu->c\_hangman, &splk->splk\_hangman);

}

}

Come si evince dal codice OS161 usa una verisone più performante degli spinlock, la test+test and set. L’istruzione test and set essendo atomica è molto onerosa in termini di performance, quindi è preferibile chiamarla il meno volte possibile. Per fare ciò, quando i thread sono in busy-wait anziché chiamare la test and set per verificare se il lock è stato rilasciato, si usa una get per ottenere il valore del lock, se è ancora aquisito si continua l’attesa, altrimenti viene chiamata la test and set e si acquisisce il lock. In questo modo si riduce il numero di chiamate di questa istruzione perché viene eseguita soltanto quando il lock è stato rilasciato.

Rilascio dello spinlock:

void spinlock\_release(struct spinlock \*splk)

{

/\* this must work before curcpu initialization \*/

if (CURCPU\_EXISTS()) {

KASSERT(splk->splk\_holder == curcpu->c\_self);

KASSERT(curcpu->c\_spinlocks > 0);

curcpu->c\_spinlocks--;

HANGMAN\_RELEASE(&curcpu->c\_hangman, &splk->splk\_hangman);

}

splk->splk\_holder = NULL; //imposta l’holder del lock a null

membar\_any\_store();

spinlock\_data\_set(&splk->splk\_lock, 0); //Setta il valore del lock a 0, quindi è rilasciato

spllower(IPL\_HIGH, IPL\_NONE);

}

Inoltre sono presenti le funzioni:

* void spinlock\_init(struct spinlock \*splk): inizializza lo spinlock;
* void spinlock\_cleanup(struct spinlock \*splk): ripulisce lo spinlock;
* bool spinlock\_do\_i\_hold(struct spinlock \*splk): ritorna true se la cpu corrente ha acquisito lo spinlock, false altrimenti.

Semafori

Un altro meccanismo di sincronizzazione implementato in OS161 è il semaforo. Un semaforo contiene una variabile intera usata come contatore e si può accedere ad esso solamente tramite due funzioni atomiche: wait() and signal(), anche chiamate, rispettivamente, P() e V().

La funzione P() verifica se il contatore del semaforo è minore o uguale a zero, se lo è mette il thread in busy wait. Non appena il contatore sarà maggiore di zero il thread viene risvegliato, avendo, cosi, la possibiltà di accedere alla sezione critica e il contatore viene decrementato, portandolo nuovamente a zero per impedire ad altri thread di poter accedere.

La funzione V() invece, viene chiamata all’uscita della sezione critica. La funzione semplicemente incrementa il contatore, permettendo a un altro thread di essere risvegliato.

In OS161 i semafori sono usati in combinazione con i wait channels e gli spinlock precedentemente descritti. I wait channels permettono di mettere in attesa e risvegliare i thread. Essi contengono una lista di thread in attesa e si può scegliere di risvegliarne uno casualmente o tutti. Un thread viene messo in attesa tramite la funzione:

* void wchan\_sleep( struct wchan \*wc, struct spinlock \*lk)

e risvegliato tramite le funzioni:

* void wchan\_wakeone( struct wchan \*wc, struct spinlock \*lk)
* void wchan\_wakeall( struct wchan \*wc, struct spinlock \*lk).

Sono presenti inoltre le funzioni:

* struct wchan \* wchan\_create(const char \*name): crea il wait channel
* void wchan\_destroy(struct wchan \*wc): distrugge il wait channel
* bool wchan\_isempty( struct wchan \*wc, struct spinlock \*lk): ritorna true se la lista di thread in attesa è vuota, false altrimenti.
* Inserire codice?

I semafori utilizzano la seguente struct:

struct semaphore {

char \*sem\_name;

struct wchan \*sem\_wchan;

struct spinlock sem\_lock;

volatile unsigned sem\_count;

};

Come si nota, oltre al nome e alla variabile contatore, essa contiene anche il riferimento al wait channel e uno spinlock.

Le funzioni principali del semaforo sono:

* La funzione P:

void P(struct semaphore \*sem)

{

KASSERT(sem != NULL);

KASSERT(curthread->t\_in\_interrupt == false);

//Viene acquisito lo spinlock contenuto nel semaforo per proteggere il wait channel

spinlock\_acquire(&sem->sem\_lock);

while (sem->sem\_count == 0) {

//Finchè il contatore è a zero i thread vengono messi in attesa e inseriti nella lista del

//wchan

wchan\_sleep(sem->sem\_wchan, &sem->sem\_lock);

}

//Se si esce dal while il contatore è maggiore di 0, ma ciò viene rincontrollato per evitare

//che un altro thread abbia preceduto quello corrente

KASSERT(sem->sem\_count > 0);

sem->sem\_count--; //Viene decrementato il contatore

spinlock\_release(&sem->sem\_lock); //Viene rilasciato lo spinlock

}

* La funzione V():

void V(struct semaphore \*sem)

{

KASSERT(sem != NULL);

//Acquisizione dello spinlock per proteggere le modifiche al semaforo

spinlock\_acquire(&sem->sem\_lock);

//Incremento del contatore e controllo che sia maggiore di zero

sem->sem\_count++;

KASSERT(sem->sem\_count > 0);

//Risveglio di uno dei thread in attesa nel wait channel

wchan\_wakeone(sem->sem\_wchan, &sem->sem\_lock);

//Rilascio del lock

spinlock\_release(&sem->sem\_lock);

}

Inoltre sono presenti le funzioni:

* struct semaphore \* sem\_create(const char \*name, unsigned initial\_count): crea il semaforo
* void sem\_destroy(struct semaphore \*sem): distrugge il semaforo;

Meccanismi di sincronizzazione in XV6

XV6 presenta due tipi di lock come meccanismi di sincronizzazione: spinlock e sleep-lock

Spinlock

Gli spinlock usano la seguente struttura dati:

struct spinlock {

uint locked; // Contiene 0 se il lock è libero, 1 se è acquisito

char \*name; // Nome del lock

struct cpu \*cpu; // La cpu che posside il lock

};

Come in OS161 anche in XV6 viene utlizzata una variabile intera binaria per verificare se il lock è libero o acquisito. Non ci sono differenze tra le struct dei due sistemi.

Lo spinlock viene acquisito tramite la seguente funzione:

void acquire(struct spinlock \*lk)

{

push\_off(); //disabilta le interruzioni per evitare deadlock

if(holding(lk))

panic("acquire");

//chiama la test and set, se è diverso da 0 il lock è già acquisito quindi resta in attesa, altrimenti lo //acquisisce

while(\_\_sync\_lock\_test\_and\_set(&lk->locked, 1) != 0)

;

\_\_sync\_synchronize();

lk->cpu = mycpu(); //Imposta la cpu corrente come holder del lock

}

Come notiamo dal codice riportato sopra xv6 non utilizza la verisone ottimizzata test+test and set degli spinlock, ma continua a richiamare sempre la test and set. Questo riduce le perfomance del sistema, in quanto, essendo un’operazione atomica il thread in esecuzione non può essere interrotto sprecando così cicli di clock senza, sostanzialmente, far nulla.

La funzione di rilascio, invece, presenta il seguente codice:

void release(struct spinlock \*lk)

{

if(!holding(lk))

panic("release");

lk->cpu = 0; //Viene azzerato l’holder del lock

\_\_sync\_synchronize();

//Operazione atomica che imposta il valore di locked a 0, rilasciando, così il lock

\_\_sync\_lock\_release(&lk->locked);

pop\_off(); //riabilita le interruzioni

}

A differenza della funzione acquire, la funzione release non presenta sostanziali differenze con la verisione di OS161.

Anche qui sono presenti le funzioni:

* void initlock(struct spinlock \*lk, char \*name): Inizializza lo spinlock
* int holding(struct spinlock \*lk): Restituisce 1 se la cpu corrente possiede il lock e il lock è acquisito, 0 altrimenti.

A differenza di OS161 in XV6 non è presente una funzione di cleanup.

Sleeplock

In XV6 è presente una versione differente di lock, gli sleeplock. A volte il sistema ha necessità di possedere il lock per molto tempo, ad esempio se sta scrivendo il contenuto di un file nel disco. Questa è un’operazione che richiede molti cicli di clock, per tanto proteggere il file con uno spinlock sarebbe molto oneroso, in quanto il busy waiting spreca cicli di clock inutilmente impedendo ad altri processi di usare la cpu per del lavoro utile. Gli sleeplock servono per evitare ciò. Utilizzando quest’ultimi, quando il processo è in attesa cede la cpu permettendo così ad altri processi di utilizzarla e di conseguenza di avere performance e efficienza migliori.

Uno sleeplock presenta una struttura di questo tipo:

struct sleeplock {

uint locked; // Contiene 0 se il lock è libero, 1 se è acquisito

struct spinlock lk; // spinlock che protegge lo sleeplock

char \*name; // Nome del lock

int pid; // Il processo che posside il lock

};

Dalla struct si nota che viene utilizzato uno spinlock per proteggere il campo locked dello sleeplock, questo risulta evidente nella funzione di acquisizione del lock:

void acquiresleep(struct sleeplock \*lk)

{

acquire(&lk->lk); //Acquisisce lo spinlock

while (lk->locked) { //Finchè lo sleeplock è aquisito si mette il processo in attesa

sleep(lk, &lk->lk); //La sleep mette in attesa il processo,, cede la cpu e lo spinlock

}

lk->locked = 1; //Appena lo sleeplock è libero lo si acquisisce impostando locked a 1

lk->pid = myproc()->pid; //Viene impostato il processo corrente come possessore del lock

release(&lk->lk); //Viene rilasciato lo spinlock

}

La funzione sleep usa i wait channel, essa per prima cosa acquisisce lo spinlock contenuto nella struct proc del processo, adesso può rilasciare in modo sicuro lo spinlock che si era acquisito all’inizio della funzione acquiresleep, dopo di ciò inserisce il processo corrente nella lista d’attesa del wait channel e imposta il suo stato a SLEEPING, richiama lo scheduler permettendo così a un altro processo di ottenere la cpu, infine rilascia lo spinlock del processo e riacquisisce lo spinlock originale (Quello contenuto nella struct sleeplock). Il rilascio e l’acquisizione del lock originale, permette ad altri processi di tentare di acquisire lo sleeplock ed essere inseriti nel wait channel, in questo modo si evita il busy waiting.

La funzione di rilascio, invece, si presenta nel seguente modo:

void releasesleep(struct sleeplock \*lk)

{

acquire(&lk->lk); //Acquisisce lo spinlock

lk->locked = 0; //Rilascia lo sleeplock impostanto questa variabile a 0

lk->pid = 0; //Viene azzerato il possessore dello sleeplock

wakeup(lk); //Viene risvegliato un processo in attesa

release(&lk->lk); //Viene rilasciato lo spinlock

}

La wakeup acquisisce lo spinlock contenuto nella struc proc, dopo recupera un processo casualmente dal wait channel e imposta il suo stato a RUNNABLE, permettendogli di essere scelto dallo scheduler per essere eseguito, infine rilascia lo spinlock.

Anche in questa versione di lock sono presenti le funzioni:

* void initsleeplock(struct sleeplock \*lk, char \*name): inizializza il lock;
* int holdingsleep(struct sleeplock \*lk): Restituisce 1 se il processo corrente possiede il lock e il lock è acquisito, 0 altrimenti.