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)

{

struct cpu \*mycpu;

// Vengono disabilitati gli interrupt per evitare deadlock

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) {

// Ottiene il valore del lock, se è diverso da 0 resta in attesa

if (spinlock\_data\_get(&splk->splk\_lock) != 0) {

continue;

}

// Chiama la test and set, se è diverso da 0 il lock è già acquisito quindi resta in attesa

// altrimenti lo acquisisce

if (spinlock\_data\_testandset(&splk->splk\_lock) != 0) {

continue;

}

break;

}

membar\_store\_any();

// Imposta la cpu corrente come holder del lock

splk->splk\_holder = mycpu;

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);

}

// Imposta l'holder del lock a NULL

splk->splk\_holder = NULL;

membar\_any\_store();

// Setta il valore del lock a 0, quindi è libero

spinlock\_data\_set(&splk->splk\_lock, 0);

spllower(IPL\_HIGH, IPL\_NONE);

}

Inoltre sono presenti le funzioni:

* void spinlock\_init(struct spinlock \*splk): inizializza lo spinlock;

void spinlock\_init(struct spinlock \*splk)

{

spinlock\_data\_set(&splk->splk\_lock, 0);

splk->splk\_holder = NULL;

HANGMAN\_LOCKABLEINIT(&splk->splk\_hangman, "spinlock");

}

* void spinlock\_cleanup(struct spinlock \*splk): ripulisce lo spinlock;

void spinlock\_cleanup(struct spinlock \*splk)

{

KASSERT(splk->splk\_holder == NULL);

KASSERT(spinlock\_data\_get(&splk->splk\_lock) == 0);

}

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

bool spinlock\_do\_i\_hold(struct spinlock \*splk)

{

if (!CURCPU\_EXISTS()) {

return true;

}

/\* Assume we can read splk\_holder atomically enough for this to work \*/

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

}

Wait channels

I wait channels permettono di mettere in attesa e risvegliare i thread. Essi contengono una lista di thread in attesa di entrare in una determinata sezione critica e non appena questa è libera, si può scegliere di risvegliarne uno, in modo casuale, o tutti.

La struct di un wait channel in OS161 è la seguente:

struct wchan {

const char \*wc\_name; // Nome del wait channel

struct threadlist wc\_threads; // Lista dei thread in attesa

};

Un thread viene messo in attesa tramite la funzione:

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

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

{

// Verifica che il thread non sia in un interrupt handler

KASSERT(!curthread->t\_in\_interrupt);

// Verifica che possegga lo spinlock

KASSERT(spinlock\_do\_i\_hold(lk));

// Verifica che non si posseggano altri spinlock

KASSERT(curcpu->c\_spinlocks == 1);

// Il thread viene messo in stato SLEEP e inserito nella lista d'attesa

thread\_switch(S\_SLEEP, wc, lk);

// Viene acquisito il lock

spinlock\_acquire(lk);

}

e risvegliato tramite le funzioni:

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

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

{

struct thread \*target;

// Verifica che si possegga il lock

KASSERT(spinlock\_do\_i\_hold(lk));

// Viene preso il primo thread dal wchan

target = threadlist\_remhead(&wc->wc\_threads);

// Se è nullo non c'è nessun thread in attesa, quindi la funzione termina

if (target == NULL) {

return;

}

// Il thread recuperato dalla lista viene messo in stato RUNNABLE

thread\_make\_runnable(target, false);

}

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

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

{

struct thread \*target;

struct threadlist list;

// Verifica che si possegga il lock

KASSERT(spinlock\_do\_i\_hold(lk));

// Viene creata una lista

threadlist\_init(&list);

// Vengono presi tutti i thread in attesa sul wchan e spostati nella lista appena creata

while ((target = threadlist\_remhead(&wc->wc\_threads)) != NULL) {

threadlist\_addtail(&list, target);

}

// Tutti i thread nella lista vengono messi in stato RUNNABLE

while ((target = threadlist\_remhead(&list)) != NULL) {

thread\_make\_runnable(target, false);

}

// Viene ripulita la lista

threadlist\_cleanup(&list);

}

Sono presenti inoltre le funzioni:

* struct wchan \* wchan\_create(const char \*name): crea il wait channel

struct wchan \* wchan\_create(const char \*name)

{

struct wchan \*wc;

// Alloca dinamicamente la memoria necessaria al wchan

wc = kmalloc(sizeof(\*wc));

if (wc == NULL) {

return NULL;

}

// Crea una lista per contenere i thread in attesa

threadlist\_init(&wc->wc\_threads);

wc->wc\_name = name;

return wc;

}

* void wchan\_destroy(struct wchan \*wc): distrugge il wait channel

void wchan\_destroy(struct wchan \*wc)

{

threadlist\_cleanup(&wc->wc\_threads);

kfree(wc);

}

* bool wchan\_isempty( struct wchan \*wc, struct spinlock \*lk): ritorna true se la lista di thread in attesa è vuota, false altrimenti.

bool wchan\_isempty(struct wchan \*wc, struct spinlock \*lk)

{

bool ret;

KASSERT(spinlock\_do\_i\_hold(lk));

ret = threadlist\_isempty(&wc->wc\_threads);

return ret;

}

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 semafori utilizzano la seguente struct:

struct semaphore {

char \*sem\_name; // Nome semaforo

struct wchan \*sem\_wchan; // Wait channel

struct spinlock sem\_lock; // Spinlock

volatile unsigned sem\_count; // Counter

};

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)

{

// Verifica che il semaforo non sia nullo

KASSERT(sem != NULL);

// Verifica che il thread non sia in un interrupt handler

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

// Viene acquisito lo spinlock contenuto nel semaforo per proteggere il wchan

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

// Finchè il contatore è 0, i thread vengono messi in attesa e inseriti nella lista del wchan

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

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

}

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

// per evitare che un altro thread abbia preceduto quello corrente

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

// Viene decrementato il contatore

sem->sem\_count--;

// Viene rilasciato lo spinlock

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

}

* La funzione V():

void V(struct semaphore \*sem)

{

// Verifica che il semaforo non sia nullo

KASSERT(sem != NULL);

// Acquisisce lo spinlock per proteggere le modifiche al semaforo

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

// Incrementa il contatore e verifica che sia maggiore di 0

sem->sem\_count++;

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

// Risveglia uno dei thread in attesa nel wchan

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

// Rilascia il lock

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

}

Inoltre sono presenti le funzioni:

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

struct semaphore \* sem\_create(const char \*name, unsigned initial\_count)

{

struct semaphore \*sem;

sem = kmalloc(sizeof(\*sem));

if (sem == NULL) {

return NULL;

}

sem->sem\_name = kstrdup(name);

if (sem->sem\_name == NULL) {

kfree(sem);

return NULL;

}

sem->sem\_wchan = wchan\_create(sem->sem\_name);

if (sem->sem\_wchan == NULL) {

kfree(sem->sem\_name);

kfree(sem);

return NULL;

}

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

sem->sem\_count = initial\_count;

return sem;

}

* void sem\_destroy(struct semaphore \*sem): distrugge il semaforo;

void sem\_destroy(struct semaphore \*sem)

{

KASSERT(sem != NULL);

/\* wchan\_cleanup will assert if anyone's waiting on it \*/

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

wchan\_destroy(sem->sem\_wchan);

kfree(sem->sem\_name);

kfree(sem);

}

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)

{

// Disabilta le interruzioni per evitare deadlock

push\_off();

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();

// Imposta la cpu corrente come holder del lock

lk->cpu = mycpu();

}

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");

// Viene azzerato l’holder del lock

lk->cpu = 0;

\_\_sync\_synchronize();

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

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

// Riabilita le interruzioni

pop\_off();

}

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

void

initlock(struct spinlock \*lk, char \*name)

{

lk->name = name;

lk->locked = 0;

lk->cpu = 0;

}

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

int

holding(struct spinlock \*lk)

{

int r;

r = (lk->locked && lk->cpu == mycpu());

return r;

}

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

Wait channels

A differenza di OS161 nei wait channels presenti in XV6 non esiste una lista in cui inserire i processi in attesa. La struct di un processo contiene un campo wchan.

struct proc {

struct spinlock lock;

// p->lock must be held when using these:

enum procstate state; // Process state

void \*chan; // If non-zero, sleeping on chan

int killed; // If non-zero, have been killed

int xstate; // Exit status to be returned to parent's wait

int pid; // Process ID

[…]

};

Quando un processo sarà in attesa, in questo campo viene inserito su che channel esso è in attesa, tramite la funzione sleep.

void sleep(void \*chan, struct spinlock \*lk)

{

struct proc \*p = myproc();

// Viene acquisito il lock del processo

acquire(&p->lock);

// E rilasciato quello della funzione che ha chiamato la sleep

release(lk);

// Viene inserito il channel su cui è in attesa il processo

p->chan = chan;

// Il processo è messo in SLEEPING

p->state = SLEEPING;

// Viene richiamato lo scheduler per mettere in stato RUNNABLE un altro processo

sched();

// Tidy up.

p->chan = 0;

// Si rilascia il lock del processo

release(&p->lock);

// Si riacquisisce il lock originale

acquire(lk);

}

La funzione di risveglio, di coseguenza, cercherà tra tutti i processi quelli che contengono nel campo chan il channel corretto e riseveglierà tutti i processi in attesa su quel channel.

void wakeup(void \*chan)

{

struct proc \*p;

// Itera su tutti i processi

for(p = proc; p < &proc[NPROC]; p++) {

// Se il processo è uno diverso da quello attuale

if(p != myproc()){

// Viene acquisito il lock del processo

acquire(&p->lock);

// Verifica che il processo sia in SLEEPING e nel channel corretto

if(p->state == SLEEPING && p->chan == chan) {

// Il processo viene messo in stato RUNNABLE

p->state = RUNNABLE;

}

// Viene rilasciato il lock

release(&p->lock);

}

}

}

Non esiste una funzione per risvegliare un solo processo.

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)

{

// Acquisisce lo spinlock

acquire(&lk->lk);

// Finchè lo sleeplock è aquisito si mette il processo in attesa

while (lk->locked) {

// La sleep mette in attesa il processo e cede la cpu a un altro processo

sleep(lk, &lk->lk);

}

// Appena lo sleeplock è libero lo si acquisisce impostando locked a 1

lk->locked = 1;

// Viene impostato il processo corrente come possessore del lock

lk->pid = myproc()->pid;

// Viene rilasciato lo spinlock

release(&lk->lk);

}

Il rilascio e l’acquisizione del lock originale, fatta dalla funzione sleep vista in precedenza 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)

{

// Acquisisce lo spinlock

acquire(&lk->lk);

// Rilascia lo sleeplock impostanto questa variabile a 0

lk->locked = 0;

// Viene azzerato il possessore dello sleeplock

lk->pid = 0;

// Viene risvegliato un processo in attesa

wakeup(lk);

// Viene rilasciato lo spinlock

release(&lk->lk);

}

Anche in questa versione di lock sono presenti le funzioni:

* void initsleeplock(struct sleeplock \*lk, char \*name): inizializza il lock;

void

initsleeplock(struct sleeplock \*lk, char \*name)

{

initlock(&lk->lk, "sleep lock");

lk->name = name;

lk->locked = 0;

lk->pid = 0;

}

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

int

holdingsleep(struct sleeplock \*lk)

{

int r;

acquire(&lk->lk);

r = lk->locked && (lk->pid == myproc()->pid);

release(&lk->lk);

return r;

}