Lezione 7

Sincronizzazione dei flussi d'esecuzione

Sistemi operativi

24 aprile 2012

Marco Cesati

System Programming Research Group
Università degli Studi di Roma Tor Vergata

Di cosa parliamo in questa lezione?

La sincronizzazione dei flussi d'esecuzione

1. Flussi di esecuzione e race condition
2. Le regioni critiche
3. Operazioni atomiche
4. Primitive di sincronizzazione
5. Il problema dello stallo
6. La sincronizzazione in Linux

Flussi d'esecuzione concorrenti

La fonte di maggior complicazione nei sistemi operativi è costituita dall'esistenza di *flussi di esecuzione concorrenti*:

- Il flusso d'esecuzione di un processo può sempre essere interrotto da un IRQ ($\Longrightarrow$ gestore di interruzione)
- In una applicazione multithread, ciascun thread è un flusso di esecuzione "in concorrenza" con quelli degli altri thread
- In un sistema multiprogrammato con prelazione, ciascun processo può essere interrotto da un altro processo
- In un sistema multiprocessore, diversi processi o gestori di interruzione sono in esecuzione contemporaneamente

La coerenza delle strutture di dati **private** di ciascun flusso di esecuzione è garantita dal meccanismo di cambio del flusso

La coerenza delle strutture di dati **condivise** tra i vari flussi di esecuzione concorrenti non è garantita da tale meccanismo

Race condition

Se due o più flussi di esecuzione hanno una struttura di dati in comune, la concorrenza dei flussi può determinare uno stato della struttura non coerente con la logica di ciascuno dei flussi

Race condition

Lo stato della memoria condivisa tra due o più flussi di esecuzione concorrenti dipende dall'ordine esatto (temporizzazione) degli accessi alla memoria stessa

Le race condition sono tra gli errori del software più insidiosi e difficili da determinare:

- non sono per loro natura deterministici
- possono non apparire mai sul sistema di sviluppo
- possono non essere facilmente replicabili
- sono di difficile comprensione

Esempio di race condition

Due flussi di esecuzione condividono una variabile `counter`:

Flusso #1:

```
for (;;) {
  crea_risorsa();
  counter = counter + 1;
}
```

Flusso #2:

```
while (counter > 0) {
  counter = counter - 1;
  consuma_risorsa();
}
```

Qual è l'origine della race condition?

L'incremento e decremento di `counter` non sono operazioni atomiche! Ciascuna consiste nel:

- Trasferimento del valore di `counter` in un registro (load)
- Operazione di incremento o decremento sul registro
- Trasferimento del valore del registro in `counter` (store)

Esempio di race condition (2)

Sia `counter` = 2: questa temporizzazione

Istante	Flusso #1	Flusso #2
1	$R_0 \leftarrow$ `counter` (2)	
2	$R_0 \leftarrow R_0 + 1$ (3)	
3	`counter` $\leftarrow R_0$ (3)	
4		$R_1 \leftarrow$ `counter` (3)
5		$R_1 \leftarrow R_1 - 1$ (2)
6		`counter` $\leftarrow R_1$ (2)

produce alla fine `counter` = 2 (corretto); ma quest'altra

Istante	Flusso #1	Flusso #2
1	$R_0 \leftarrow$ `counter` (2)	
2	$R_0 \leftarrow R_0 + 1$ (3)	
3		$R_1 \leftarrow$ `counter` (2)
4		$R_1 \leftarrow R_1 - 1$ (1)
5		`counter` $\leftarrow R_1$ (1)
6	`counter` $\leftarrow R_0$ (3)	

produce alla fine `counter` = 3 (non corretto)

Regioni critiche

Sezione o regione critica

Una sequenza di istruzioni di un flusso di esecuzione che accede ad una struttura di dati condivisa e che quindi non deve essere eseguita in modo concorrente ad un'altra regione critica

- Una regione critica è sempre relativa a se stessa
  - l'accesso alla regione è mutuamente esclusivo nel tempo
- Una regione critica può anche essere relativa ad un'altra regione critica (entrambe accedono agli stessi dati)
  - non si può consentire che due flussi eseguano nello stesso momento l'una o l'altra delle regioni critiche

È necessario disporre di regioni critiche che consentano

- La mutua esclusione dei flussi d'esecuzione
- Il progresso di tutti i flussi d'esecuzione
- L'attesa limita per l'ingresso in una regione critica

Regioni critiche nei sistemi monoprocessori

In un sistema monoprocessore un metodo facile per realizzare una regione critica tra processi User Mode consiste nel disabilitare temporaneamente il diritto di prelazione

Tuttavia questo metodo ha alcuni problemi:

- rende lo scheduler di tipo cooperativo all'interno di una regione critica, quindi tutti gli altri flussi di esecuzione possono essere bloccati per molto tempo
- da solo non è sufficiente per proteggere le regioni critiche poste all'interno del codice del kernel e che condividono dati con i gestori di interruzioni

In un sistema monoprocessore è sufficiente disabilitare le interruzioni hardware per realizzare una regione critica in Kernel Mode (viene disabilitata anche la prelazione)

Tuttavia anche questo metodo ha alcuni problemi:

- Non è possibile eseguire operazioni bloccanti che portino ad un cambio di processo entro la regione critica
- Nei sistemi multiprocessori disabilitare le interruzioni su tutti i processori è molto costoso e non risolve il problema

Soluzione di Peterson

La *soluzione di Peterson* è una tecnica software per implementare una regione critica tra due flussi di esecuzione

```
flag[p] = 1;
turno = 1-p;
while (flag[1-p] == 1 && turno == 1-p)
    /* aspetta */ ;
/*
   SEZIONE CRITICA
*/
flag[p] = 0;
```

`p`:	indice del flusso d'esecuzione (0 oppure 1)
`1-p`:	indice dell'altro flusso d'esecuzione
`flag[i]`:	1 se il flusso `i` vuole entrare, 0 altrimenti
`turno`:	l'indice del flusso autorizzato ad entrare

Perché la soluzione di Peterson è corretta?

Soluzione di Peterson (2)

```
flag[p] = 1;
turno = 1-p;
while (flag[1-p] && turno == 1-p)
    /* aspetta */ ;
/*
   SEZIONE CRITICA
*/
flag[p] = 0;
```

La mutua esclusione è assicurata perché il flusso `p` accede alla regione critica solo se l'altro flusso `1-p` non vuole entrare oppure concede la precedenza

In altri termini: se entrambi i flussi vogliono entrare, `flag[0]=flag[1]=1`, ma solo il flusso con indice uguale a `turno` può entrare, l'altro deve aspettare

È assicurato il progresso e l'attesa limitata: il flusso `p` è bloccato solo finché il flusso `1-p` è nella sezione critica; all'uscita imposterà `flag[1-p]` a 0 sbloccando il flusso `p`

Quali assunzioni si devono fare perché la soluzione funzioni?

Gli accessi a `flag` e `turno` devono essere atomici e coerenti

Supporto hardware per la sincronizzazione

Un accesso ad una cella di memoria è coerente se in ogni istante tutti i flussi di esecuzione “vedono” il medesimo valore

Nei sistemi moderni dotati di memorie statiche cache la coerenza deve essere garantita per mezzo di opportune tecniche realizzate in hardware oppure in software

In generale qualunque soluzione al problema della regione critica deve basarsi su garanzie offerte dall’hardware

Molte primitive di sincronizzazione tra i flussi di esecuzione possono essere ricondotte ad un unico concetto di base:

Lock (lucchetto)

Meccanismo di sincronizzazione costituito da una variabile il cui contenuto è accessibile in modo atomico e coerente

Supporto hardware per la sincronizzazione (2)

Tutte le architetture hardware possiedono qualche istruzione macchina atomica e coerente che consente di costruire un lock

- `FetchAndAdd`: istruzione macchina che legge ed incrementa il contenuto di una variabile (IA-32: `LOCK; XADD`)
- `TestAndSet`: istruzione macchina che consente di leggere il valore di una variabile ed impostare un nuovo valore (IA-32: `LOCK; BTS`)
- `Swap`: istruzione macchina che scambia tra loro il contenuto di una variabile e di un registro (o altra variabile) (IA-32: `LOCK; XCHG`)
- `TestAndSwap`: istruzione macchina che legge una variabile e se questa è uguale ad un dato valore scambia tra loro il contenuto della variabile e di un registro (o altra variabile) (IA-32: `LOCK; CMPXCHG`)

Esempi di lock realizzati con istruzioni atomiche

Lock realizzato con `TestAndSet`:

```
while (TestAndSet(&lock, LOCKED) == LOCKED)
  /* attendi */ ;
/* SEZIONE CRITICA */
lock = UNLOCKED;
```

Lock realizzato con `Swap`:

```
local = LOCKED;
while (local == LOCKED)
  Swap(&lock, &local);
/* SEZIONE CRITICA */
lock = UNLOCKED;
```

In cosa differiscono rispetto alla soluzione di Peterson?

- Funzionano con più di due flussi d'esecuzione
- L'attesa non è necessariamente limitata

Primitive di sincronizzazione con attesa attiva o sospensione

Quando un flusso di esecuzione tenta di acquisire una primitiva di sincronizzazione non disponibile, in alternativa:

- continua ad eseguire un piccolo ciclo di istruzioni (*tight loop*) attendendo che un lock cambi stato (*spinlock*)
- viene sospeso dall'esecuzione sulla CPU, entrando in uno stato di attesa (*primitiva bloccante*)

I SO offrono ed utilizzano primitive di entrambi i tipi:

Gli spinlock sono utili nei sistemi multiprocessore

- convenienti per proteggere regioni critiche di breve durata
- anche se durante l'attesa la CPU non compie lavoro utile

Viceversa le primitive bloccanti sono utili sia nei sistemi monoprocessori che in quelli multiprocessori

- hanno un costo (overhead) maggiore degli spinlock
- convenienti per regioni critiche di lunga durata
- possono essere usate solo nei flussi di esecuzioni schedulabili dal SO (thread e processi)

Implementazione di uno spinlock

Esempio di implementazione di uno spinlock Kernel Mode:

```
typedef spinlock_t int;

void spin_lock(spinlock_t *lock)
{
    local_interrupt_disable();
    while (TestAndSet(lock, LOCKED) == LOCKED) ;
}

void spin_unlock(spinlock_t *lock)
{
    *lock = UNLOCKED;
    local_interrupt_enable();
}
```

Perché vengono disabilitate le interruzioni locali?

- Se lo spinlock fosse utilizzato in qualche gestore di interruzione si potrebbe verificare uno stallo
- Lo stallo non può verificarsi tra flussi di esecuzione su CPU differenti!

Semaforo

Il *semaforo* è una primitiva di sincronizzazione inventata dall'olandese E. Dijkstra (1930–2002) basata su una variabile intera contatore:

- Inizializzata con valore $n > 0$
- Accessibile tramite due primitive atomiche:
  - Wait, Down, P (dall'olandese "Proberen", verificare): attendi che il contatore sia positivo, poi decrementalo
  - Signal, Up, V (dall'olandese "Verhogen", incrementare): incrementa il contatore

Un semaforo con $n > 1$ può essere acquisito da n flussi di esecuzione in modo concorrente (*semaforo contatore*)

Un semaforo con $n = 1$ può essere acquisito da un solo flusso alla volta (*semaforo binario* oppure *semaforo mutex*)

Per realizzare un semaforo è sempre necessario utilizzare un lock di livello più basso

Implementazione di un semaforo bloccante

```
struct semaphore {
  int value;            /* contatore */
  struct PCB *queue;    /* coda di processi in attesa */
  spinlock_t lock;      /* lock per accesso atomico  */
};

void wait(struct semaphore *s)
{
  spin_lock(&s->lock);      /* acquisisci il lock */
  s->value--;               /* decrementa il contatore */
  if (s->value >= 0) {      /* se contatore non è negativo */
    spin_unlock(&s->lock); /* rilascia il lock */
    return;                 /* il semaforo è acquisito */
  }
  enqueue_self(&s->queue); /* accoda il processo */
  set_self_state(WAIT);     /* imposta lo stato del processo */
  spin_unlock(&s->lock);    /* rilascia il lock */
  schedule();               /* rilascia il processore */
} /* quando torna in esecuzione il semaforo è acquisito */

void signal(struct semaphore *s)
{
  spin_lock(&s->lock);      /* acquisisci il lock */
  s->value++;               /* incrementa il contatore */
  if (s->value <= 0)        /* se contatore non è positivo */
    /* rimuovi il primo processo in coda e risveglialo */
    set_ready_state(dequeue_process(&s->queue));
  spin_unlock(&s->lock);    /* rilascia il lock */
}
```


Risorse ed istanze di risorse

Ciascuna primitiva di sincronizzazione arbitra l'accesso ad una *risorsa* (ad es., una regione critica oppure una stampante)

Una risorsa definisce il numero massimo di *istanze* assegnabili nello stesso momento ai flussi di esecuzione

Ad esempio, un semaforo inizializzato con il valore n definisce una risorsa con n istanze

Uno stesso flusso di esecuzione può richiedere più risorse, ed anche più istanze di ciascuna risorsa

Che differenza esiste tra due risorse ed una risorsa con due istanze?

Le risorse sono ben distinte tra loro, mentre le varie istanze di una stessa risorsa sono indistinguibili tra loro

Inversione di priorità

L'interazione tra scheduler a priorità e primitive di sincronizzazione dà luogo a fenomeni interessanti

Inversione di priorità

Un processo di priorità alta non può essere eseguito in quanto è bloccato in attesa di una risorsa (ad es., semaforo binario) acquisito da un altro processo di priorità bassa

Le inversioni di priorità sono praticamente inevitabili.
Tuttavia un meccanismo è particolarmente dannoso:

Inversione di priorità incontrollata

Un processo di priorità alta non è eseguito per un tempo arbitrariamente lungo in quanto in attesa di una risorsa acquisita da un processo di priorità bassa, e quest'ultimo non progredisce a causa di uno o più processi di priorità media

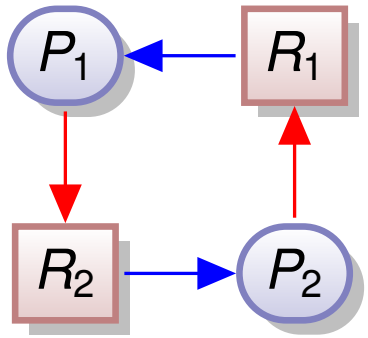
Stallo (deadlock)

Stallo o deadlock

Situazione per la quale due o più flussi di esecuzione non possono progredire in quanto ciascuno tenta di acquisire risorse possedute dagli altri

Il classico caso di deadlock si ha quando:

- il processo P_1 possiede la risorsa R_1 ed è bloccato sulla richiesta della risorsa R_2
- il processo P_2 possiede la risorsa R_2 ed è bloccato sulla richiesta della risorsa R_1



Ogni SO deve realizzare meccanismi per evitare la creazione di stalli durante l'esecuzione dei vari flussi di esecuzione

Condizioni necessarie per lo stallo

Una condizione di stallo tra processi non può verificarsi a meno che non si verifichino contemporaneamente quattro condizioni:

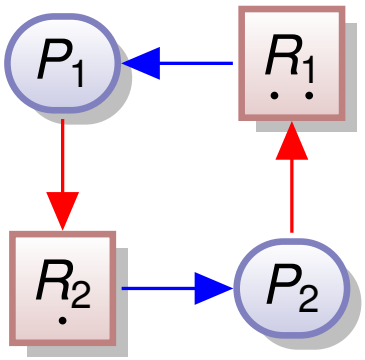
1. Mutua esclusione: almeno una richiesta ad una risorsa deve essere in mutua esclusione
2. Possesso e attesa: un processo in possesso di almeno una risorsa deve attendere di acquisire risorse non disponibili
3. Impossibilità di prelazione: non esiste diritto di prelazione sulle risorse (non si può togliere una risorsa ad un processo che l'ha già acquisita)
4. Attesa circolare: deve esistere una sequenza ordinata di processi e risorse in cui ciascun processo richiede una risorsa assegnata al processo successivo, e tale che l'ultima risorsa è assegnata al primo processo

(Le condizioni non sono del tutto indipendenti tra loro, ad esempio la condizione 4 implica la 2)

Grafo di assegnazione delle risorse

Per modellare assegnazioni e richieste si utilizza un grafo:

- I processi sono nodi circolari
- Le risorse sono nodi rettangolari
- All'interno di ciascun nodo alcuni punti rappresentano le istanze definite
- Una freccia da una (istanza di una) risorsa ad un processo denota una "acquisizione"
- Una freccia da processo a risorsa denota una "richiesta"



Cosa rappresenta un ciclo in un grafo di assegnazione?

Un possibile stallo!

Metodi per la gestione degli stalli

Essenzialmente esistono tre metodi per gestire gli stalli:

1. Utilizzo di un protocollo che prevenga gli stalli impedendo che le quattro condizioni necessarie siano verificate contemporaneamente
2. Utilizzo di un protocollo che eviti gli stalli negando le assegnazioni di risorse che potrebbero in futuro portare ad una situazione di stallo
3. Utilizzo di un algoritmo che determini l'esistenza di uno stallo e lo risolva cancellando una o più assegnazioni

In effetti esiste anche un'altra possibilità: un SO potrebbe ignorare di fatto gli stalli, ad esempio perché

- gli stalli sono eventi eccezionalmente rari, oppure
- perché possiede meccanismi di controllo che evitano che un flusso di esecuzione possa rimanere "congelato" per sempre

Prevenzione degli stalli

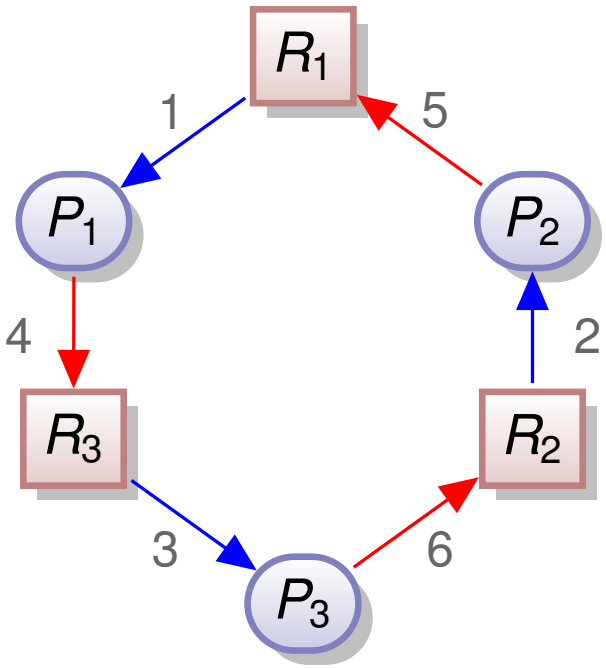
Per prevenire gli stalli è necessario che le quattro condizioni necessarie non siano mai tutte vere allo stesso tempo

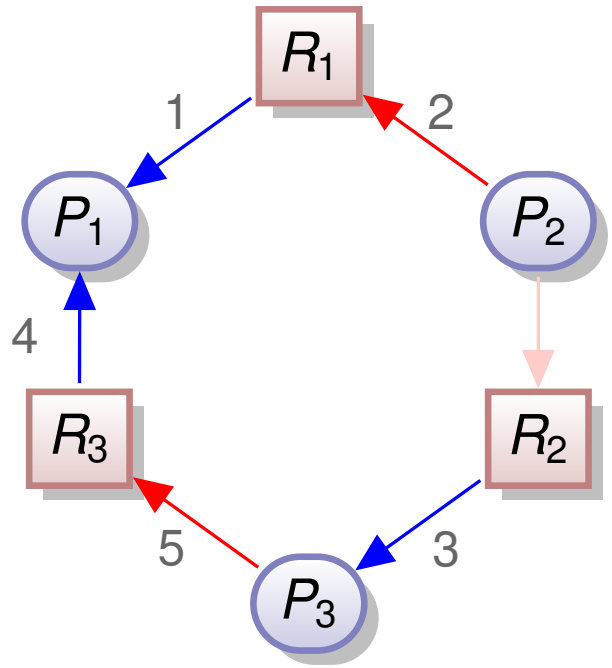
- Mutua esclusione: in generale questa caratteristica dipende dalla risorsa e non può essere sfruttata dal SO per prevenire gli stalli
- Possesso e attesa: per la prevenzione è sufficiente che un processo che richiede una risorsa non ne possieda altre
  - Un protocollo può assegnare a ciascun processo che inizia l'esecuzione tutte le risorse di cui avrà bisogno
  - Un altro protocollo consiste nel richiedere che non si possa ottenere più di una risorsa alla volta
- Impossibilità di prelazione: è necessario forzare il rilascio delle risorse in caso di necessità
  - se un processo richiede una risorsa che non si può assegnare, allora rilascia immediatamente tutte le risorse che già possiede, e attende che tutte le risorse necessarie siano disponibili
  - in alternativa, se un processo richiede una risorsa non disponibile, se questa appartiene ad un processo a sua volta in attesa gli viene sottratta e assegnata al primo

Prevenzione degli stalli (2)

Attesa circolare: il metodo più semplice per prevenire questa condizione consiste nel

- definire un ordinamento per le risorse: $R_1, R_2, \ldots R_n$
- imporre che ciascun processo richieda le risorse nell'ordine predefinito





Esempio: la sincronizzazione in Linux

Il kernel Linux implementa molti tipi di primitive di sincronizzazione, ad esempio:

- spinlock: spinlock tradizionale
- semaphore: semaforo contatore bloccante
- mutex: semaforo binario adattivo: si comporta come uno spinlock se la risorsa è occupata da un processo in esecuzione su un'altra CPU e se la coda d'attesa è vuota; altrimenti è bloccante
- rt-mutex: mutex con meccanismo per evitare l'inversione di priorità incontrollata
- rwsem: semaforo bloccante specializzato per consentire l'accesso concorrente dei lettori
- rwlock: spinlock specializzato per consentire l'accesso concorrente dei lettori
- RCU (Read-Copy-Update): primitiva di mutua esclusione che non fa uso di lock
- SRCU (Sleeping RCU): RCU bloccante
- futex (fast userspace mutex): meccanismo e API per realizzare primitive di sincronizzazione in User Mode