1 Programmazione concorrente

Si parla di programmazione "concorrente" quando si realizzano programmi che eseguono contemporaneamente ed interagiscono durante la loro esecuzione.
In questo capitolo illustreremo i problemi che deve affrontare chi si occupa della programmazioni di Sistemi Operativi, o di altri programmi che debbano stare ad un livello basso nella stratificazione dei S.O.
Ciò verrà fatto in generale, andando in dettaglio solo episodicamente, senza la presunzione di poter dare gli elementi sufficienti per scrivere effettivamente programmi di sistema.
La programmazione concorrente è una necessità inderogabile per i programmatori del Sistema Operativo; al contrario, i programmatori di applicazioni d'utente non sono normalmente interessati ai problemi di concorrenza, che vengono risolti in modo trasparente[1] dal S.O..
Quando però è necessario scrivere programmi che abbiano il massimo dell'efficienza, soprattutto quando destinati a sistemi multiprocessore, allora anche il programmatore di applicazioni potrà far ricorso alla programmazione concorrente.
L'aspetto più difficile della programmazione concorrente è il debugging. Infatti i programmi che hanno accesso contemporaneo alle risorse possono funzionare bene per la gran parte del tempo e bloccarsi improvvisamente solo quando accade una specifica, unica, sequenza di eventi, magari molto improbabile a verificarsi.
Questo significa trovarsi di fronte a crash improvvisi e "misteriosi" delle applicazioni, molto difficili a riprodursi[2].
Gli errori dipendono dall'ordine, casuale, con cui si presentano gli eventi che determinano i cambiamenti nelle risorse condivise.
Se i programmi concorrenti non sono scritti in modo accorto anche un piano di test molto accurato può risultare inutile ed i sistemi possono presentare errori software dovuti ad una particolare sequenza di eventi, che non si è mai presentata durante tutta la fase di test.

Concorrenza

Definiamo il concetto di concorrenza come:
Concorrenza: presenza di più attività simultanee.

In un sistema multiprogrammato i processi che devono fare un uso comune di alcune delle risorse del sistema, si dicono "processi **concorrenti**" (da "concurrent", che significa "simultaneo").

Nel caso, tutto sommato raro, in cui processi concorrenti non interagiscano, non ci sono differenze sostanziali nella loro programmazione, rispetto ai programmi ordinari; invece tutto cambia quando i programmi concorrenti si influenzano reciprocamente durante la loro esecuzione.
L'influenza reciproca fra due programmi si realizza attraverso la competizione o la cooperazione nell'uso delle risorse.
Come già definito nel Capitolo "processi", due processi in competizione si disputano l'uso della risorsa, mentre due processi cooperanti la usano "insieme".
La prosecuzione dell'esecuzione di entrambi i tipi di programmi concorrenti è influenzata dalle interazioni fra di essi, perciò se qualcosa non funziona è in pericolo la possibilità stessa che i programmi funzionino.

Nel far funzionare più "attività" contemporaneamente, sorgono tre tipi di problemi, che la programmazione concorrente deve risolvere:
- passaggio da un processo all'altro
- protezione di un processo dagli altri processi
- sincronizzazione delle attività dei processi, per farli comunicare e per evitare errori

i primi due problemi sono quasi sempre "assorbiti" dal Sistema Operativo, che si fa carico dello scheduling, eventualmente della preemption e che realizza spazi di memoria separati e protetti per i processi (non per i thread!).
L'ultimo problema deve essere lasciato al programmatore, perché ogni programma concorrente richiede di trattare a suo modo la sincronizzazione dei processi.

Determinismo
Un S.O. ha un comportamento non deterministico perché deve far fronte agli utenti ed ai loro programmi, che si succedono in sequenze non note a priori. Peraltro i programmi devono sempre funzionare nello stesso modo, per cui, visti da un utente, i programmi utente devono essere deterministici.

1 in Informatica, "trasparente" è detto di qualcosa che svolge le sue funzioni "di nascosto", senza avere alcun effetto visibile
2 per poter "rivedere" la sequenza di eventi che ha scatenato un errore di programmazione concorrente potrebbe dover essere necessario ricorrere ad un emulatore in circuito (ICE, vedi Volume 1).

1.0.1 Corse

Nel capitolo sugli interrupt abbiamo illustrato con un esempio la ragione per cui la CPU disabilita tutti gli interrupt prima di eseguire una procedura di risposta all'interruzione. Se non facesse così le interruzioni alle ISR rischierebbero di causare problemi di **corse** (races).
Nella programmazione del Sistema Operativo il problema delle corse si presenta ancora più pesantemente, dato che esistono due condizioni che peggiorano la situazione:

1. in un sistema operativo multiprogrammato le ISR tendono ad essere più lunghe, dato che, come abbiamo già illustrato, spesso devono fare delle chiamate al S.O. . Questo fatto può obbligare ad abilitare il sistema d'interruzione durante l'esecuzione delle ISR, per non lasciare il computer senza interrupt per troppo tempo.
2. durante il corso dell'esecuzione dei processi accadono eventi che, pur non essendo direttamente collegati con interrupt hardware, hanno delle caratteristiche funzionali del tutto simili ad esse (es. la "sospensione" di un processo, il suo "risveglio"). Questi eventi avvengono sempre ad interrupt abilitati per cui non vi è la "protezione" dalle corse assicurata dall'esecuzione delle ISR a interrupt disabilitasti.

Vediamo, con un esempio simile al precedente che fa uso del concetto di processo, come possono presentarsi problemi di corse.

1.0.2 Corse e consistenza delle variabili

Una prima cosa da notare è che durante l'esecuzione di un processo le variabili del programma assumono temporaneamente dei valori non coerenti.
Nella gran parte dei casi questo fatto non causa assolutamente alcun problema, ma esistono rare occasioni nelle quali un programma apparentemente inoffensivo nasconde errori subdoli.
Scriviamo per esempio una linea di programma in C:

```
int contatore;
..
contatore = contatore / 3; /* (1) */
..
```

Questo ci dice che il valore di contatore successivo all'istruzione deve essere un terzo del suo valore di precedente.
A prima vista sembra che l'istruzione C venga eseguita tutta "insieme", in realtà quando verrà compilata si trasformerà in diverse istruzioni di macchina.
In una piattaforma 80386, la traduzione della linea (1) sarà qualcosa di simile al seguente:

```
..
    XOR EDX, EDX         ; <1> cancellazione della parte alta del dividendo (per IDIV)
    MOV EAX, [contatore] ; <2> lettura della parte bassa
    IDIV WORD PTR 3      ; <3> divisione
    MOV [contatore], EAX ; <4> scrittura del risultato
..
```

Nel programma in Assembly il contenuto della variabile "contatore" non viene aggiornato fino a <4>, mentre è già calcolato in <3>. Dunque fra <3> e <4> il contenuto di EAX e quello di "contatore" non sono coerenti.
Se "contatore" è una variabile "locale", definita nel processo e che solo dal processo può essere modificata, questa incoerenza non darà nessun problema. Infatti, anche se dovesse intervenire una preemption fra <3> e <4>, quando non c' è coerenza fra "contatore" ed EAX, EAX non verrà modificato da altri e prima o poi verrà ricopiato esatto in "contatore".
Se invece "contatore" è una variabile condivisa da un altro processo simultaneo possono sorgere dei problemi, in un unico caso "sfortunato".
Il caso è quello in cui capita una preemption proprio fra <3> e <4>, nel punto in cui EAX e "contatore" non sono coerenti.
Supponiamo ora che arrivi un interrupt hardware mentre il nostro processo sta eseguendo l'istruzione <3> . Supponiamo che il S.O. decida che il nostro processo non deve proseguire e dia la CPU ad un secondo processo, lasciando il primo in attesa. Supponiamo infine che il secondo processo aggiorni durante la sua esecuzione il valore di "contatore".
Al suo successivo risveglio il nostro processo eseguirà la <4> e scriverà in "contatore" il valore vecchio, che si era calcolato prima che l'altro processo aggiornasse "contatore" (il valore vecchio di EAX viene ripristinato dal descrittore di processo, insieme a tutti gli altri registri della CPU, nel momento in cui il nostro processo viene nuovamente eseguito).
Il risultato è che l'aggiornamento del contatore fatto dal secondo processo è completamente perduto, è come se non fosse mai stato fatto. Siamo di fronte ad una corsa.
Si usa dire che la variabile "contatore" non è più "**consistente**", con un termine mutuato direttamente dall'Inglese ("consistent"), che si può tradurre, in un migliore Italiano, come "coerente".
Si può notare che perché si verifichi questo errore debbono succedere molte cose, tutto sommato abbastanza improbabili:

1. il S.O. deve spodestare il nostro processo proprio fra <3> e <4>.
2. prima che il nostro processo esegua di nuovo deve eseguire un processo che può modificare "contatore"
3. quel processo deve modificare "contatore" proprio ora.

Cionondimeno non possiamo accettare che ci sia una probabilità, sia pur bassa, che il programma non funzioni.
Questo vuol dire che i programmi concorrenti dovranno funzionare "sempre", cioè in presenza di qualsiasi sequenza di esecuzione dei processi.
Dunque:
Nella programmazione concorrente non si può fare alcuna ipotesi sull'ordine di esecuzione dei processi. Essi possono essere eseguiti in un ordine qualunque, ed è ciò che accade effettivamente.

Ci chiediamo ora come si può aggiustare il programma precedente. L'errore è dovuto al fatto che una singola istruzione scritta in C non viene realizzata con una sola istruzione di macchina, ed è quindi suscettibile di essere interrotta in momenti inattesi.
Il programma in Assembly sarebbe stato facilmente curabile in questo modo (almeno in un sistema a monoprocessore):

```
..
    MOV EAX, [contatore] ; <2> lettura della parte bassa
    CLI                  ; evito l'interruzione e quindi anche la preemption
    DIV WORD PTR 3       ; <3> divisione
    MOV [contatore], EAX ; <4> scrittura del risultato
    STI                  ; ora che la modifica è fatta lascio ancora interrompere
..
```

Disabilitando il sistema delle interruzioni abbiamo temporaneamente evitato che altri processi interrompessero il nostro, proprio nel momento critico in cui esso fa l'elaborazione ed il salvataggio del valore della variabile comune. In questo modo in quella fase critica dell'accesso a "contatore" i processi dovranno procedere "uno alla volta".
In qualche modo abbiamo reso indivisibile l'elaborazione e la modifica della variabile "contatore".
Si può dire che abbiamo "**sincronizzato**" i processi perché abbiamo impedito agli altri processi di intervenire al tempo sbagliato.
Naturalmente l'intervento sugli interrupt, che risolve il problema in Assembly, non è possibile programmando in C, per cui sono state inventate tecniche più ad alto livello per difendere le risorse condivise dall'intervento inopportuno di processi contemporanei.

Oltre al fatto che non è realizzabile in linguaggi ad alto livello, c'è anche da notare che la soluzione di disabilitare gli interrupt è un po' troppo drastica, perché impedisce la preemption non solo ai processi che tentano di modificare "contatore" ma anche a tutti gli altri, ed anche alle ISR dei dispositivi!
Una buona soluzione al problema della consistenza della variabili dovrà prevedere che venga impedita la prosecuzione solo a quei processi che provano a modificare la variabile da "difendere" e non a tutti.

1.0.3 Operazioni atomiche

L'esempio precedente ci ha mostrato come il rendere "indivisibile" un'istruzione ha curato un errore di concorrenza.
Le tecniche che si usano per proteggere l'uso delle variabili comuni fanno sempre uso di istruzioni o di funzioni "indivisibili".

Chiamiamo "**istruzioni atomiche**"[3] o "**primitive**" quelle istruzioni che non possono essere interrotte durante la loro esecuzione.

Sui sistemi con una sola CPU tutte le istruzioni di macchina sono per loro natura primitive, mentre le istruzioni in linguaggio ad alto livello non lo sono mai.
Utilizzando piccoli programmi in Assembly che disabilitano le interruzioni è possibile costruire delle **funzioni primitive** (procedure primitive), sulle quali possono contare i linguaggi ad alto livello per raggiungere l'obbiettivo di sincronizzare i processi. Oltre all'uso di procedure nelle CPU moderne si può contare su istruzioni di macchina specifiche, inventate proprio per la sincronizzazione dei processi, che permettono di realizzare primitive di sincronizzazione senza disabilitare gli interrupt.
Le funzioni primitive per la sincronizzazione, permettono di rendere ordinato l'acceso ad una risorsa, che non verrà usata contemporaneamente da tutti i processi ma dal numero voluto di processi simultanei.

In sistemi multiprocessore le istruzioni di macchina non sono atomiche perché mentre una CPU esegue un'istruzione che prevede l'uso della memoria ci può essere un'altra CPU che sta lavorando sulla stessa locazione. Per rendere atomiche le istruzioni di macchina bisogna che nessun'altra CPU abbia accesso simultaneo alla locazione che si sta modificando.
Questo è possibile aggiungendo una nuova linea al control "bus" del computer, che impedisca alle altre CPU di accedere alla memoria durante l'execute di istruzione atomiche.
Nell'architettura X86 esistono la linea LOCK# ed il prefisso Assembly LOCK ("blocca"), che può essere anteposto a certe istruzioni (es. LOCK DEC [Locazione]).

3 Letteralmente "atomico" (dal greco) significa "indivisibile"

Quando un'istruzione è preceduta dal prefisso LOCK durante tutta la sua fase di execute il segnale LOCK# viene tenuto alto e le altre CPU non possono accedere alla memoria.
La presenza di una linea LOCK# alta può bloccare, almeno in linea di principio: l'intera memoria, il solo circuito integrato interessato dalla istruzione atomica o addirittura la sola, singola, locazione interessata. Ciò dipende da come è fatto l'hardware del computer.

1.0.4 Sezioni critiche

Nel corso della storia dell'informatica il problema della realizzazione dell'accesso a risorse in mutua esclusione ha trovato molte soluzioni interessanti, anche se valide solo parzialmente, sorvoliamo su di esse e passiamo ad illustrare i concetti più fecondi di: "sezione critica" e "semaforo".

Si chiamano "**sezioni critiche**" le parti di codice di un programma che accedono alla stessa risorsa da processi diversi.

La "protezione" di una sezione critica deve obbligare i processi ad accedere in modo ordinato alle risorse, salvaguardando la consistenza delle informazioni.
Nel caso dell'esempio precedente tutta l'istruzione in C del punto (1) deve essere inserita in una sezione critica, per proteggere la risorsa "contatore". Fino a che tutta l'istruzione (1) non è stata eseguita deve essere impedita l'esecuzione di tutti gli altri processi che vogliono modificare "contatore".
Peraltro, per non abbassare l'efficienza del computer, se un processo non tenta di modificare "contatore" deve poter eseguire tranquillamente. Quindi un processo che entra una sezione critica può benissimo perdere la CPU, basta che non lo faccia a favore di processi che tentano di modificare "contatore".

Per entrare ed uscire da una sezione critica sono state inventate specifiche operazioni primitive. Di solito si tratta di chiamate al Sistema Operativo.
Quando un processo vuole entrare in una sezione critica chiede il permesso al S.O. con una chiamata di sistema; alla fine dell'esecuzione della sezione critica il processo esegue una diversa chiamata per comunicare al S.O. la conclusione.
Se qualche altro processo sta eseguendo la sezione critica, il processo che chiede di entrare viene sospeso e rimane in questa condizione fino a che l'altro processo non comunica che ha liberato la sezione critica.
Se invece la sezione critica è libera il processo può proseguire ed eseguirla tutta, poi comunicare al S.O. la conclusione con la primitiva di uscita, in modo che altri possano entrare.

Figura 1:

Per evitare questa evenienza basta che tutti i processi che usano "contatore" definiscano una sezione critica nei punti dove intervengono sulla variabile.

1.0.5 Semafori

Il semaforo è uno strumento per la realizzazione delle sezioni critiche. E' una variabile intera non negativa, condivisa dai processi, che essi modificano con funzioni primitive, in modo da segnalare la loro presenza all'interno della sezione critica.
Il nome "semaforo" proviene dalla segnalazione ferroviaria. Supponiamo di avere un ponte a binario unico in una linea a due binari, come illustrato in Figura 2.
Per non causare disastri ferroviari l'accesso al ponte dovrà essere in mutua esclusione. Immaginiamo una situazione in cui due treni, A e B stiano approcciando l'accesso al ponte. Mentre si avvicinano il ponte è libero da altri treni ed il semaforo è spento. La sequenza degli eventi che accadono è la seguente

1. A giunge all'imbocco del ponte e guarda il semaforo; è spento
    1.1. A accende il semaforo
    1.2. A entra nel ponte
2. B giunge all'imbocco del ponte e guarda il semaforo; è acceso
    2.1. B si blocca
    2.2. B continua a guardare il semaforo e rimane fermo fino a che esso non si spegne
3. A percorre il ponte (sezione critica)
4. A esce dalla zona del ponte, fino ad una posizione non pericolosa
    4.1 A spegne il semaforo
5. B vede il semaforo spento
    5.1. B accende il semaforo
    5.2. B entra nel ponte

..

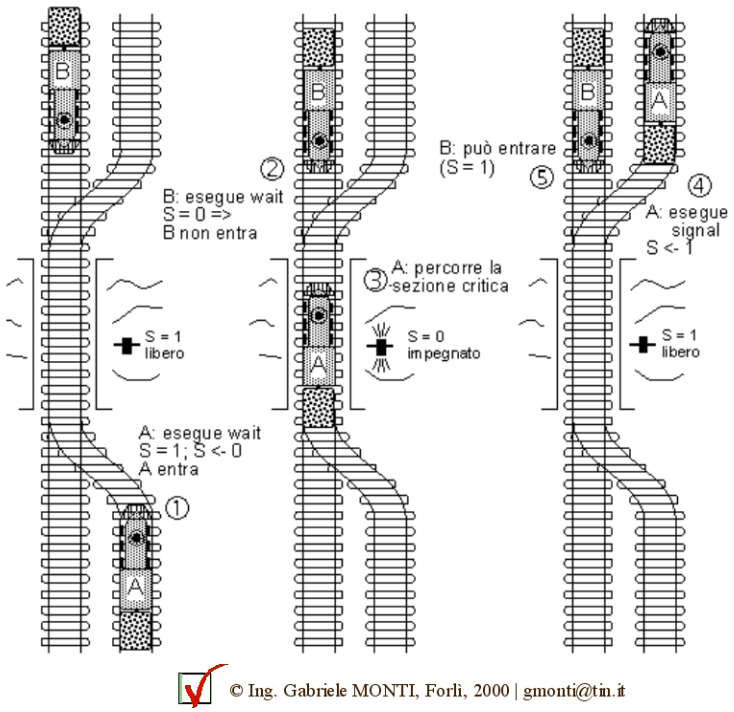

Figura 2: mutua esclusione nell'accesso al ponte

Si dice che l'accesso ad una risorsa condivisa avviene in "**mutua esclusione**" quando i processi che possono accedere a quella risorsa lo possono fare solo uno alla volta.

Le tecniche per l'assicurazione della mutua esclusione nell'accesso ad una risorsa vengono dette "**serializzazione**" dell'accesso o "sequenzializzazione".

Analogo al semaforo ferroviario è il semaforo "informatico", che naturalmente assumerà la figura di un flag, o meglio di in contatore, le cui modifiche sono effettuate con operazioni primitive.

La seguente definizione descrive la forma più semplice di semaforo "informatico":

Un semaforo è un contatore, condiviso in memoria fra i processi che lo utilizzano, cui si può avere accesso con due sole operazioni primitive, dette wait() e signal(). Un semaforo serve a sincronizzare l'esecuzione dei processi che lo utilizzano.

wait() ("attendi"), indica l'entrata nella sezione critica e "**signal()**", "segnala", detta anche "release", marca l'uscita dalla sezione critica.

Funzionamento di un semaforo

Dunque un semaforo è costituito da una variabile intera non negativa, condivisa, usata come un contatore.
Il valore della variabile del semaforo può essere modificato solo tramite le funzioni primitive wait() e signal().
Quando un processo deve usare una risorsa protetta da un semaforo aspetta sul quel semaforo, eseguendo la funzione wait().
La wait() torna solo quando la sezione critica è libera. Altrimenti non torna, per un tempo indeterminato.
Al ritorno della wait() il processo entra nella sezione critica e la esegue tutta. Per "segnalare" agli altri processi che ha liberato la sezione critica, e quindi uno di essi può entrare, il processo chiama la funzione primitiva signal().
wait() e signal() modificano lo stato della variabile semaforo nel modo indicato in Figura 3, ove è mostrato un codice, espresso in pseudo-C, che illustra in linea di principio la realizzazione di wait() e signal().

```
int s   ; variabile statica del programma, visibile a tutti i processi

void CreaSemaforo (s);
    /* ..
    OPERAZIONE PRIMITIVA
    qui esegue del codice che crea e inizializza la struttura dati del semaforo,
    che comprende almeno la variabile s (il "contatore" del semaforo)
```

```
    oltre ad eventuali altri attributi, che dipendono dall'implementazione effettiva
    .. */

    /* la variabile semaforo  (s) viene inizializzata a 1
            (semaforo mutuamente esclusivo (mutex, vedi oltre)) */
    s = 1;
return ();

void wait (s);
    /* OPERAZIONE PRIMITIVA
       se la sezione critica è impegnata il processo non procede nell'esecuzione
       in attesa che essa si liberi */
    while (s == 0); // attesa attiva
    s--;
return ();

void signal(s);
    /* OPERAZIONE PRIMITIVA */
    s++;
    /* ^ incrementare la variabile semaforo vuol dire aumentare il numero
         di processi che possono entrare nella sezione critica */
return ();
```

Figura 3: implementazione di wait() e signal() di semafori

In Figura 3 "s" è la variabile semaforo; essa viene definita in una funzione CreaSemaforo(), che le assegna un valore iniziale di 1.
Come si può vedere wait() sottrae 1 al valore del semaforo, ma lo fa solo quando può, cioè dopo che il ciclo while è finito e si ha l'"autorizzazione" all'ingresso nella sezione critica.
Dunque la variabile s non assume mai valori negativi, perché il processo che prova a decrementarla quando ha valore zero viene bloccato fino a che essa non vale almeno uno.
Il ciclo while (s == 0) sembra non finire mai, perché se s = 0 il programma gira nel ciclo e non ne può uscire, dato che non modifica s. Bisogna però tener conto che s può essere modificato da altri processi concorrenti, precisamente dal primo che ha eseguito la wait(), che ha trovato ancora s = 1 ed ha avuto accesso alla sezione critica. Questo processo proseguirà fino alla signal() e la eseguirà.
La signal() aumenta il valore di s, che diventa 1.
Supponiamo che ci siano altri processi, fermi sul ciclo while (s == 0) della loro wait(). Ora la variabile s vale uno per cui il primo fra i processi in attesa che viene scelto dallo scheduler può uscire dal suo ciclo while, ed anche dalla wait(), ed accedere alla sezione critica.
Vediamo in dettaglio una sequenza di eventi nella quale un processo deve attendere sulla wait(). Supponiamo di avere due processi , PA e PB, e che la sezione critica da eseguire si inizialmente libera. Supponiamo che il semaforo sia mutuamente esclusivo (mutex).
Una sequenza di eventi potrebbe essere la seguente:

..
0. Il valore della variabile semaforo è inizializzato a s = 1
1. **PA** giunge all'inizio della sezione critica ed esegue wait(). Il valore della variabile semaforo è s = 1
    1.1. **S.O.** (wait() di PA) controlla il valore del semaforo; s = 1 allora
    1.2. **S.O.** (wait() di PA) decrementa il valore di s; s = 0; la wait() ritorna
    1.3. **PA** entra nella sezione critica
2. **PB** giunge all'imbocco della sezione critica ed esegue wait(), ora il valore del semaforo è s = 0
    2.1. **S.O.** (wait() di PB) blocca PB nella coda del semaforo, wait() NON ritorna
3. **PA** percorre tutta la sezione critica
4. **PA** segnala l'uscita dalla sezione critica con signal()
    4.1. **S.O.** (signal() di PA) cerca nella coda del semaforo, c'è solo PB che viene messo "ready"
    4.2. **S.O.** (signal() di PA) incrementa il valore di s; s = 1; signal() ritorna
5. **PB** risvegliato dalla signal() di PA, quando è il suo turno viene schedulato; riparte la sua wait(), prima bloccata
    5.1. **S.O.** (wait() di PB) controlla il valore del semaforo; ora s = 1 allora
    5.2. **S.O.** (wait() di PB) decrementa il valore di s; s = 0; la wait() ritorna
    5.3. **PB** entra nella sezione critica

..

L'implementazione di wait() e signal() appena descritta contiene un loop di "attesa attiva" che spreca tutto il time slice dei processi che attendono sulla wait() a guardare il valore di s. Ciò va a detrimento dell'efficienza del computer, che perde tempo per un loop che di fatto è inutile. Si noti che in un sistema monoprocessore nessuno può cambiare il valore della variabile S mentre gira il nostro processo, per cui va sprecato sempre tutto il time slice a disposizione del processo che attende in una wait().

I semafori con attesa attiva vengono anche detti "**spin lock**". Sono usati p.es. nel kernel di Linux, fra le altre cose per coordinare l'accesso mutuamente esclusivo ai device driver, nei sistemi multiprocessore SMP.

Semafori con coda

Per evitare l'attesa attiva si può modificare la funzione wait() in modo che essa chieda al S.O. di mettere il processo in stato di "Wait" ("suspended"). Si può anche aggiungere al semaforo una coda di processi, nella quale vengono tenuti tutti quei processi che sono in attesa di poter entrare nelle sezioni critiche che sono protette dal semaforo.
In questo modo i processi che trovano il semaforo rosso sono sospesi, non vengono più scelti dal process scheduler e non eseguono più l'attesa attiva. Naturalmente ora sorge il problema di "risvegliarli" perchè essi, non eseguendo, non possono più accorgersi da soli che il semaforo è cambiato di "colore".
E' logico che il compito di "risvegliare" i processi sospesi nella coda del semaforo sia assegnato alla signal(). Infatti signal() aggiorna il valore della variabile semaforo e mette in stato di "Ready" uno o più dei processi in coda su quel semaforo.
In Figura 4 è mostrato in pseudo-C il funzionamento di principio di wait() e signal() su semafori con coda.

```
void CreaSemaforo (s);
    /* ..
    OPERAZIONE PRIMITIVA
    creazione la struttura dati del semaforo, che comprende anche
    una coda per i processi in attesa su questo semaforo
    .. */
    s = 1;   // come prima
return ();

void wait (s);
    /* OPERAZIONE PRIMITIVA
       se la sezione critica è impegnata sospende il processo in attesa che si
       liberi (la successiva signal() cambia la variabile semaforo e risveglia
       i processi che sono in coda) */
    if(s == 0) SospendiIlProcesso ();
    s--;// come prima
return ();

void signal(s);
    /* OPERAZIONE PRIMITIVA */
    s++;// come prima
    ScheduleProcessiInCoda (s);
    /* ^ risveglia (rende "Ready" almeno uno dei processi in attesa sulla coda */
return ();
```

Figura 4: funzionamento di wait() e signal() di semafori con coda

In questa versione la signal() effettua il "risveglio" di almeno uno dei processi in coda, tramite la funzione ScheduleProcessiInCoda, quin non ulteriormente specificata.
Il compito di decidere quale fra i processi in coda deve essere risvegliato può essere svolto dalla signal() stessa, oppure lasciato al S.O. . In quest'ultimo caso la signal() mette in stato di "Ready" tutti i processi in attesa sulla coda. Entrerà nella sezione critica quello che verrà scelto per primo dal process scheduler.
Peraltro nulla impedisce di far "schedulare" il processo da risvegliare direttamente alla signal(), eventualmente realizzando meccanismi a priorità. Di solito però viene scelta una politica FIFO per evitare che un processo con bassa priorità venga sempre scalzato da altri più importanti e non riesca più ad uscire dalla coda.
Questo fenomeno viene detto "inversione della priorità" ed è classificabile fra i fenomeni di "**starvation**" (morte per fame!) o "livelock", in contrapposizione al deadlock, che vedremo successivamente.

Realizzazione di un semaforo

In questo paragrafo ci occuperemo della realizzazione a basso livello di un semaforo con attesa attiva. In linea di principio l'implementazione è semplice e può assomigliare al codice di Figura 5, che è la traduzione in pseudo-Assembly dello pseudocodice in Figura 3

```
wait:
    CMP [S], 0   ; <1> guardo il colore del semaforo
    JZ wait      ; <2> aspetto se è rosso
    DEC [S]      ; <3> rendo rosso il semaforo
RET

signal:
    INC [S]      ; <4> rendo verde il semaforo
RET
```

Figura 5: implementazione di wait() e signal() di semafori senza coda (la wait è sbagliata!)

Questo codice contiene una corsa. Infatti se il processo che esegue questo codice subisce una preemption fra <2> e <3>, il valore della variabile S è inconsistente. Per risolvere si può fare come già detto, eseguendo la wait() a interrupt disabilitati, ma, utilizzando istruzioni di macchina speciali, è possibile fare qualcosa di equivalente anche senza escludere gli interrupt.
Le CPU moderne, sin dal tempo dei mainframe, hanno istruzioni di macchina di tipo "Test&Set", pensate proprio per lo scopo, che eseguono un test in una locazione di memoria e la modifica del suo contenuto nella **stessa** istruzione. Dato che le istruzioni di macchina sono "per definizione" primitive, almeno in sistemi monoprocessore, le istruzioni di tipo Test&Set sono l'ideale per realizzare primitive di sincronizzazione.

Vediamo come si può fare in una CPU X86, utilizzando l'istruzione BTR (Bit Test and Reset), che esiste dal 386 in poi. BTR funziona in questo modo:

```
BTR <operando>, <numero del bit su cui lavorare>
```

<operando> può essere da 16 o da 32 bit, <numero del bit su cui lavorare> va da 0 a 31 ed è il numero di bit dell'operando da andare a vedere.

L'istruzione BTR:
1. Legge nel flag di carry il valore del bit <numero del bit su cui lavorare> di <operando>
2. Scrive il valore 0 nel bit <numero del bit su cui lavorare> di <operando>

Eseguendo questa istruzione il valore di <locazione> non rimane mai inconsistente perchè la lettura e l'aggiornamento vengono fatte nella stessa istruzione, in modo primitivo.
Per realizzare la wait() con BTR si può fare così:

```
wait:
    BTR [S], 0   ; <1> leggo nel carry il valore del semaforo, che ho memorizzato
                 ; nel bit meno significativo di S (quello di peso 0), e lo metto a 0
    JNC wait ; <2> aspetto se PRIMA di eseguire BTR in quel bit c'era 0
RET
```

Figura 6: implementazione di wait con BTS

Il codice appena mostrato può funzionare solo con semafori mutuamente esclusivi (binari), non per semafori che ammettono valori maggiori di 1 (generalizzati, vedi oltre). Per i semafori generalizzati si può usare l'istruzione XCHG, in combinazione con altre istruzioni.

Semafori binari e generalizzati

Se un semaforo viene inizializzato ad 1 al momento della sua creazione, esso permette ad un solo processo alla volta di passare, realizzando una sezione critica dall'accesso mutuamente esclusivo.
I semafori di questo tipo vengono detti "**mutex**". I valori assunti dalla variabile semaforo di un mutex possono essere solo 1 e 0. Per questo il mutex viene anche chiamato "**semaforo binario**".
Se si analizza il funzionamento di wait() e signal() si può capire che il valore con cui si inizializza la variabile semaforo al momento della sua creazione è il numero di processi che il semaforo accetta contemporaneamente nella sezione critica.
L'esecuzione "regolare" di wait() e signal() da parte dei processi dà il permesso di entrare nella sezione critica a N processi se il valore iniziale della variabile semaforo è N; se N = 1 si ha un semaforo mutex.
Un semaforo che viene inizializzato ad un valore maggiore di uno viene detto "**semaforo generalizzato**".
Infatti ogni wait() che riesce a passare decrementa il valore della variabile semaforo; se essa è inizializzata ad N diviene zero quando ci sono N processi entro la sezione critica.
Quando la variabile semaforo diviene 0 i processi cominciano a bloccarsi in coda, non passano e non decrementano ulteriormente il valore della variabile semaforo, che rimane 0.
Non appena un processo esegue signal() la variabile semaforo viene incrementata ed un altro processo può entrare nella sezione critica.

```
void CreaSemaforo (s);
    /* la variabile semaforo  (s) viene inizializzata a N, numero di
    processi che sono ammessi contemporaneamente entro la sezione critica */
    s = N;
return ();
```

Figura 7: modifiche ai semafori per semafori generalizzati

Nell'esempio del ponte si può immaginare che M binari portino ad un ponte sul quale possono passare contemporaneamente N treni.

Considerazioni sull'uso dei semafori
Il meccanismo dei semafori, pur funzionale ed efficiente, è piuttosto "pericoloso" nella programmazione, perché è difficile da tenere sotto controllo, per cui è facile sbagliare.
Ogni accesso al semaforo deve essere effettuato correttamente da tutti i processi che condividono la risorsa, sia nella fase della sua acquisizione (wait()), sia nella fase del rilascio (signal()). Basta, per esempio, che un processo termini la sezione critica senza emettere una signal(), o che non la termini mai (p.es. perché si blocca per errore al suo interno), per bloccare per sempre in coda tutti gli altri processi che vi sono entrati e che vi entreranno.
Per questo sono stati inventati dei meccanismi per la sincronizzazione dei processi di più alto livello, più affidabili e semplici da usare.
Un altro problema nell'uso dei semafori è il fatto che se un processo modifica la variabile semaforo senza entrare "ufficialmente" nella sezione critica, cioè senza fare una wait(), non può essere sospeso e può far danni senza che nessuno possa intervenire.
C'è da rilevare il fatto che un singolo semaforo può proteggere più di una risorsa, basta che ogni processo che usa quelle risorse esegua una wait() ed una signal() sul semaforo ad ogni accesso a ciascuna delle risorse protette. Questo potrebbe limitare il **grado di concorrenza** dei processi e quindi l'efficienza del sistema complessivo.

1.0.6 Monitor

Come abbiamo visto è "pericoloso" scrivere programmi concorrenti con i semafori. Per diminuire i rischi della programmazione concorrente è stato introdotto il concetto di "monitor".
Un **monitor** è un insieme di dati da condividere in modo sincronizzato fra vari processi e di procedure che permettono di utilizzare quei dati.
I dati sono di solito del tutto nascosti alle parti di programma che stanno fuori dal monitor e sono accessibili solo attraverso le procedure.
Questa definizione è del tutto simile a quella di tipo di dato astratto, o di "oggetto" nella programmazione a oggetti; i dati sono nascosti all'utente (data hiding) e sono accessibili solo indirettamente, tramite l'uso di procedure (o "metodi").
La differenza fondamentale fra oggetti e monitor è che nei monitor è assicurata "automaticamente" l'esecuzione in mutua esclusione di tutte procedure.
Il compilatore del linguaggio (o la macchina virtuale nel caso di Java) si occupa di inserire tutte le primitive di sincronizzazione necessarie per assicurare la mutua esclusione[4] fra le procedure del monitor.

I monitor "incapsulano" al loro interno le variabili condivise dai processi concorrenti e fanno in modo che il loro uso sia controllato e realizzato esclusivamente attraverso le procedure del monitor, per le quali la corretta sincronizzazione è più facilmente verificabile.
La programmazione con i monitor permette di scrivere programmi in cui la parte del codice che effettua la sincronizzazione è confinata nella stessa "zona". Al contrario l'uso dei semafori implica la gestione della sincronizzazione in punti del codice lontani fra loro, ed eventualmente sotto il controllo di persone diverse, con tutti i problemi che da ciò possono derivare.

Un monitor controlla una risorsa, od un piccolo gruppo di risorse, facendo in modo che l'accesso ad esse sia mutuamente esclusivo.

I processi che entrano in un monitor possono essere sospesi ed entrare in una coda associata al monitor, lasciando che altri processi entrino.

In alcuni casi è possibile far sospendere il processo su una condizione.

1.0.7 Il problema produttori – consumatori

Un problema tipico, che incorre in molti casi pratici, è quello in cui uno o più processi devono utilizzare dati prodotti da altri processi. Sia i processi che "producono" i dati, che quelli che li utilizzano, eseguono separatamente e lavorano a qualsiasi velocità; per questo è indispensabile trovare un meccanismo per sincronizzarli.
Il problema descritto viene solitamente chiamato "problema produttore – consumatore".
In un problema produttore – consumatore l'ordine con cui vengono "consumati" i dati è lo stesso con cui essi sono stati prodotti.
Per far sì che i processi possano proseguire nel loro lavoro, bloccandosi solo quando non ci sono dati disponibili è necessario introdurre una struttura dati di transito, nella quale memorizzare temporaneamente i dati forniti dai produttori, in attesa che ci siano consumatori disponibili per utilizzarli.
Questa struttura dati viene detta "buffer" ed è in grado di disaccoppiare le diverse velocità dei produttori e dei consumatori, memorizzando le informazioni prodotte nei momenti in cui non ci sono consumatori pronti ad utilizzarle. Nei casi più semplici si può considerare che la dimensione del buffer venga sempre scelta con tale abbondanza da esser sicuri che non si esaurirà mai, questo equivale a considerare il buffer "infinito".

4 La mutua esclusione è effettivamente assicurata: con chiamate al S.O., dal compilatore stesso, oppure dalla "macchina virtuale" (Java o linguaggi .NET)

Quando invece si deve tener conto del fatto che il buffer possa esaurirsi, allora solitamente si usa un "buffer circolare".

Buffer circolare

Si può pensare ad un buffer circolare come ad un array nel quale si rincorrono due puntatori (indici nell'array), uno dei quali indica la posizione del prossimo elemento che si può scrivere nel buffer, mentre l'altro punta al prossimo elemento da leggere. In Figura 8 è mostrata, al punto 0, la situazione dei puntatori nel momento della creazione del buffer, mentre il punto 1 mostra la situazione dopo la prima scrittura nel buffer da parte di un produttore.
Ad ogni scrittura o lettura nel buffer il relativo puntatore viene incrementato.
Quando uno dei due puntatori raggiunge l'ultimo elemento dell'array (punto 2 in Figura 8) viene fatto tornare all'inizio (punto 3). Da ciò il nome di buffer "circolare".

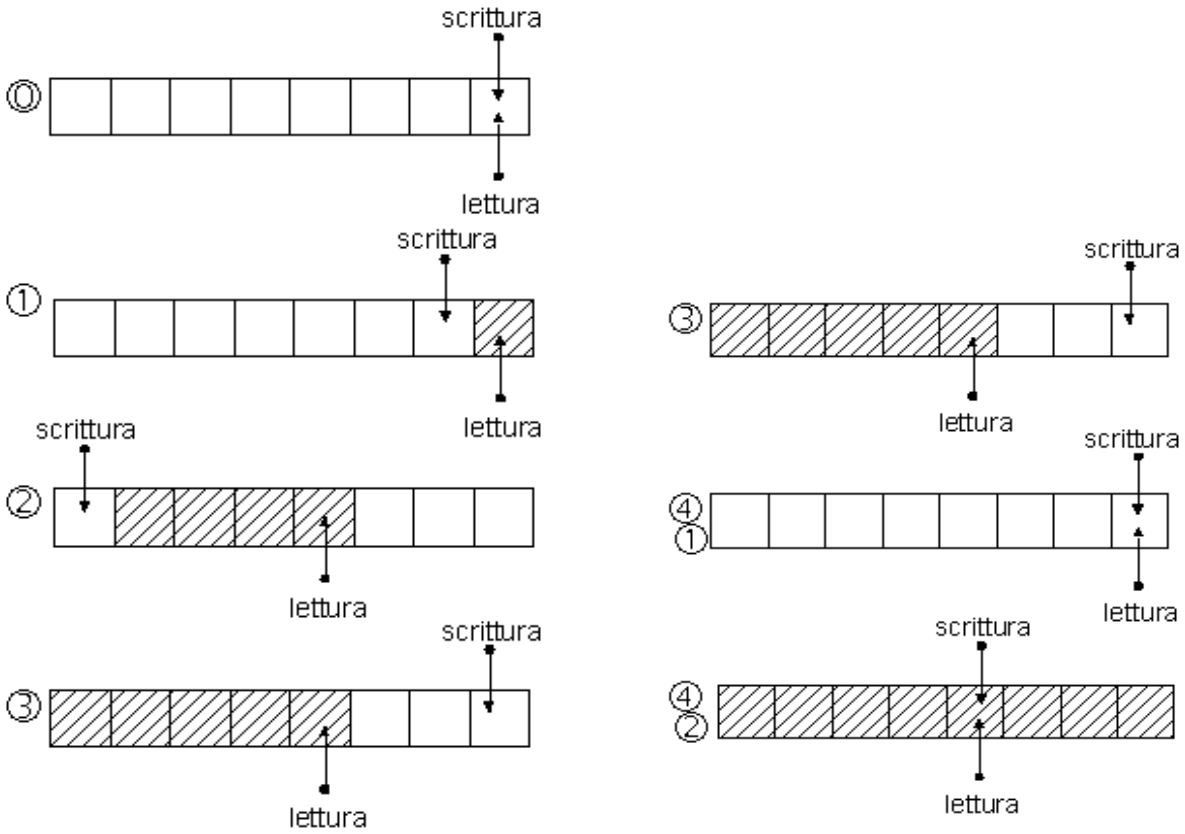

Figura 8: buffer circolare

L'accesso al buffer circolare è concorrente, per cui si devono individuare operazioni primitive che lo gestiscano. Chiamiamo "put" l'operazione che immette nel buffer un nuovo elemento, e "get" l'operazione che toglie un elemento dal buffer e lo restituisce al programma che ne ha fatto richiesta.
Sia "get" che "put" possono essere eseguite da qualsiasi processo interessato al buffer circolare.
Quando un processo esegue la "put" scrive nel buffer all'indice "scrittura", poi aggiorna l'indice, aumentandolo. Se il valore ottenuto eccede la dimensione del buffer lo "riporta a capo", per esempio in questo modo:

```
function put(elemento) {
// i processi la eseguono in modo mutuamente esclusivo
..
    Buffer[scrittura]= elemento;
    scrittura = (scrittura + 1) % nElementiBuffer;            // (1)
    // aggiornamento conteggio degli elementi nel buffer:
    nNelBuffer++;
.. }
```

Il risultato dell'istruzione (1) è il resto della divisione intera (%, "modulo"), che è sempre uguale a (scrittura + 1) tranne quando (scrittura + 1) vale nElementiBuffer. In questo caso la divisione dà risultato 1 con resto di 0, perciò scrittura viene riportato a 0.
In modo del tutto analogo si comporta il puntatore di lettura, mosso dalla primitiva "get":

```
function get() {
// i processi la eseguono in modo mutuamente esclusivo
```

```
..
    temp = Buffer[lettura];
    lettura = (lettura + 1) % nElementiBuffer;
    // aggiornamento conteggio degli elementi nel buffer:
    c--;
.. }
```

Se per lungo tempo non si effettuano operazioni di lettura dal buffer il puntatore di scrittura potrebbe raggiungere quello di lettura "doppiandolo" (situazione del punto (4) (2) in Figura 8). In questa condizione, quando i puntatori sono uguali, il buffer è pieno e si pongono due alternative, fra le quali scegliere in base alle caratteristiche dell'applicazione:
1. ignorare l'errore e sovrascrivere il contenuto del buffer, perdendo i dati più vecchi (in questo caso anche il puntatore di lettura deve essere incrementato)
2. bloccare i produttori, attendendo che un consumatore sposti in avanti il puntatore di lettura.
Quando, al contrario, non avvengono scritture per lungo tempo il buffer si svuota. Se un consumatore chiede un dato al buffer esso si deve sempre bloccare in attesa che un produttore lo introduca nel buffer (situazione del punto (4)(1) in Figura 8).
E' chiaro che le variabili "lettura", "scrittura" e "nNelBuffer" dello pseudocodice precedente devono essere protette dall'accesso contemporaneo. Un modo per farlo potrebbe essere realizzare le due procedure put() e get() come procedure di un monitor, che hanno sempre accesso mutuamente esclusivo (per un esempio pratico di problema produttore - consumatore, vedere il capitolo su Java o su .NET).

1.0.8 Il problema lettori – scrittori

TODO

1.0.9 Deadlock

"blocco critico" o "stallo"

TODO