

Argomenti

- COMUNICAZIONI TRA PROCESSI:
 - Sleep e wakeup
 - Semafori
 - Mutex
 - Mutex in Pthread
 - Monitor
 - Scambio di messaggi
 - Barriera

SCHEDULING:

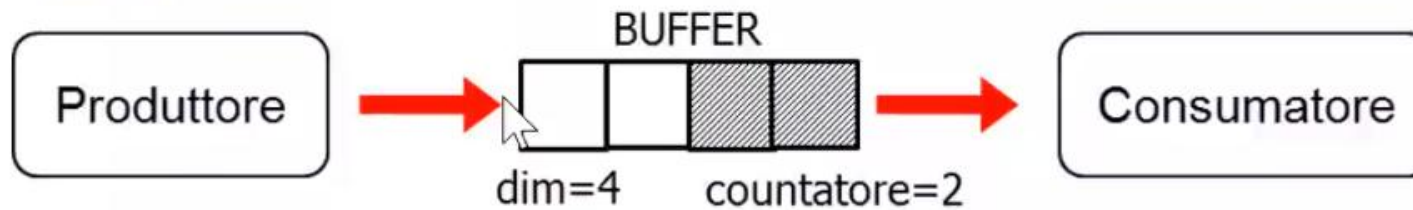
Introduzione
Sistemi batch
Sistemi interattivi
Sistemi real-time
La politica contro il meccanismo
Thread scheduling

Sleep e wakeup

- Sia la soluzione di Peterson che l'istruzione TSL (o XCHG) sono corrette ma hanno due problemi:
 - Fanno **sprecare** tempo di CPU perché richiedono **busy waiting**.
 - Un processo con bassa priorità che è sospeso e mantiene la regione critica non riesce a liberare la risorsa a causa della presenza di un processo con più alta priorità che è in esecuzione ma in busy waiting (**problema dell'inversione delle priorità**).
- Per superare queste problematiche esistono due primitive:
 - **sleep()** blocca il processo chiamante fino a che un altro processo non lo risveglia
 - **wakeup(p)** sveglia il processo p

Il problema del produttore-consumatore

- Due processi condividono un buffer comune di dimensione fissata.





- Il produttore mette degli elementi nel buffer e il consumatore li estrae.

Il codice del produttore-consumatore

```
#define DIM 100 /* dimensione del buffer */
int contatore = 0; /* numero di elementi presenti nel buffer */
void produttore(void){
    int elemento;
    while (TRUE) { /* ciclo infinito */
        elemento = produce(); /* produce un elemento */
        if (contatore==DIM) /* se il buffer è pieno */
            sleep(); /* addormentati */
        inserisce(elemento); /* (altrimenti) inserisce l'elemento nel buffer */
        contatore++; /* incrementa il numero di elementi presenti */
        if (contatore==1) /* se il buffer era vuoto, */
            wakeup(consumatore); /* occorre ri-svegliare il consumatore */
    }
}
void consumatore(void){
    int elemento;
    while (TRUE) { /* ciclo infinito */
        if (contatore==0) /* se il buffer è vuoto, */
            sleep(); /* addormentati */
        elemento = estrai (); /* (altrimenti) estrae un elemento dal buffer */
        contatore--; /* decrementa il numero di elementi presenti */
        if (contatore==DIM-1) /* se il buffer era pieno, */
            wakeup(produttore); /* occorre svegliare il produttore */
        usa(elemento); /* utilizza l'elemento */
    }
}
```



Corse critiche tra produttore e consumatore


- Supponiamo che il buffer inizialmente sia vuoto e sia avviato il consumatore che trova `contatore=0`. 
- A questo punto lo scheduler decide di interromperlo (prima della `sleep()`) e avviare il produttore.
- Il produttore inserisce un elemento nel buffer e incrementa il contatore.
- Ora `contatore=1` quindi cerca di svegliare il consumatore (che non dorme ancora).
- A questo punto lo scheduler attiva il consumatore che va subito in `sleep()`.
- Quando il produttore riempirà il buffer dormiranno entrambi «sogni tranquilli».

```
void consumatore(void){  
    int elemento;  
    while (TRUE) {  
         if (contatore==0)  
            sleep();  
        elemento = estrai();  
        contatore--;  
        if (contatore==DIM-1)  
            wakeup(produttore);  
        usa(elemento);  
    }  
}
```

```
void produttore(void){  
    int elemento;  
    while (TRUE) {  
        elemento = produce();  
        if (contatore==DIM)  
            sleep();  
        inserisce(elemento);  
        contatore++;  
        if (contatore==1)  
            wakeup(consumatore);  
    }  
}
```

Corse critiche tra produttore e consumatore

- Supponiamo che il buffer inizialmente sia vuoto e sia avviato il consumatore che trova `contatore=0`. 
- A questo punto lo scheduler decide di interromperlo (prima della `sleep()`) e avviare il produttore.
- Il produttore inserisce un elemento nel buffer e incrementa il contatore.
- Ora `contatore=1` quindi cerca di svegliare il consumatore (che non dorme ancora).
- A questo punto lo scheduler attiva il consumatore che va subito in `sleep()`.
- Quando il produttore riempirà il buffer dormiranno entrambi «sogni tranquilli».

```
void consumatore(void){  
    int elemento;  
    while (TRUE) {  
         if (contatore==0)  
            sleep();  
        elemento = estrai();  
        contatore--;  
        if (contatore==DIM-1)  
            wakeup(produttore);  
        usa(elemento);  
    }  
}
```

```
void produttore(void){  
    int elemento;  
    while (TRUE) {  
        elemento = produce();  
        if (contatore==DIM)  
            sleep();  
        inserisce(elemento);  
        contatore++;  
        if (contatore==1)  
            wakeup(consumatore);  
    }  
}
```


Corse critiche tra produttore e consumatore

- La sostanza del problema è che un `wakeup()` inviato ad un processo sveglio va perso.
- Per risolvere questa circostanza si può aggiungere **un bit di attesa wakeup()**:
 - se si invia un **wakeup()** ad un processo sveglio, si imposta ad uno questo bit;
 - quando il processo tenterà di addormentarsi, verificherà lo stato del bit:
 - se è uno allora lo azzera e rimane sveglio;
 - se è zero si addormenta.
- Questa è una soluzione tampone che termina la sua efficacia nel momento in cui aumentano i processi ed un bit non è più sufficiente a descrivere gli stati.
- Si possono aumentare i bit di wakeup, ma il principio del problema rimane sempre...

Semafori



- L'idea alla base di un semaforo è quello di contare il numero di risvegli. Un semaforo può essere 0 (nessun risveglio) o assumere un valore positivo quando uno o più risvegli risultano pendenti.
- Su un semaforo sono possibili due operazioni: **down()** e **up()** (rispettivamente generalizzazioni di sleep e wakeup).
- L'operazione **down()**:
 - se $\text{semaforo} > 0$, diminuisce il valore e continua;
 - altrimenti ($\text{semaforo} = 0$), il processo va in sleep senza completare il **down()**.
- L'operazione **up()** aumenta il valore del semaforo.
- Più processi possono essere regolati da un semaforo.
- Dopo un'operazione **up()**, su semaforo con più processi dormienti, il semaforo sarà ancora 0, ma un processo (scelto a caso dal sistema) potrà completare il suo down risvegliandosi.
- Tutte le operazioni sono **atomiche**.

Risoluzione del problema P-C

```

#define N 100                                /* dimensione del buffer */
typedef int semaforo;                        /* i semafori sono interi >0 */
semaforo mutua_esclusione = 1;              /* controlla l'accesso alla regione critica */
semaforo vuoto = N;                         /* conta il numero di posizioni vuote */
semaforo pieno = 0;                         /* conta il numero di posizioni riempite */

void produttore (void){
    int elemento;
    while (TRUE) {
        elemento = produce();               /* genera un element da inserire nel buffer */
        down(&vuoto);                       /* decrementa vuoto */
        down(&mutua_esclusione);            /* entra nella regione critica */
        inserisce(elemento);                /* inserisce l'elemento nel buffer */
        up(&mutua_esclusione);              /* abbandano la regione critica */
        up(&pieno);                         /* incrementa il numero di posizioni riempite */
    }
}

void consumatore (void){
    int elemento;
    while (TRUE) {
        down(&pieno);                       /* decrementa il numero di posizioni riempite */
        down( &mutua_esclusione);          /* entra nella regione critica */
        elemento = estrai();                /* prende un element dal buffer */
        up(&mutua_esclusione);              /* abbandano la regione critica */
        up(&vuoto);                         /* incrementa il numero di posizioni vuote */
        consume(elemento);                  /* utilizza l'elemento estratto */
    }
}

```

Semafori

- I semafori risolvono il problema della perdita dei risvegli che abbiamo visto con `sleep()` e `wakeup()`.
- Normalmente le primitive **down()** e **up()** sono realizzate come system call.
- Il sistema operativo prima di controllare il semaforo, disabilita brevemente tutti gli interrupt in modo da rendere le primitive atomiche.
- Nel caso ci siano più CPU occorre proteggere il semaforo con variabili di lock o istruzioni TSL (o XCHG) in modo da essere sicuri che solo una CPU per volta possa esaminare il semaforo.



Semafori

- Il semaforo `mutua_esclusione` è un semaforo binario poiché assume due valori e fa in modo che un solo processo per volta possa entrare nella sezione critica:

```
down(&mutua_esclusione);  
«regione critica»  
up(&mutua_esclusione);
```



Mutex

- I mutex sono una versione semplificata del semaforo.
- I mutex sono utili **solo** per gestire la mutual esclusione di risorse condivise o pezzi di codice.
- Sono facili da implementare ed efficienti, che li rende particolarmente utili nei pacchetti di thread.
- Un mutex è una variabile binaria che può trovarsi in soli due stati:
 - Locked;
 - unlocked.
- Anche se basta solo 1 bit per rappresentare un mutex, spesso si usa un numero intero:
 - 0 significa unlocked 
 - $\neq 0$ significa bloccato 
- Con i mutex si usano due procedure:
 - **mutex_lock()**, quando un processo ha bisogno di entrare nella regione critica;
 - **mutex_unlock()**, quando ha terminato l'accesso alla regione critica.

Mutex

- Se l'istruzione TSL è disponibile, sono così semplici che possono essere realizzati nello spazio utente. Il codice per l'uso **mutex_lock** e **mutex_unlock** è mostrato di seguito:

mutex_lock:

TSL REGISTER, MUTEX	copia mutex sul registro e pone mutex=1
CMP REGISTER, #0	mutex=0?
JZE ok	se mutex=0, mutex è sbloccato ed esce
CALL thread_yield	≠0 è occupato, schedulare un altro thread
JMP mutex_lock	prova ancora
ok: RET	ritorna al chiamante è nella regione critica

mutex_unlock:

MOVE MUTEX, #0	memorizza 0 nel mutex
RET	ritorna al chiamante

- La soluzione con XCHG è essenzialmente simile.

Mutex nei Pthread

- Nei Pthread le principali chiamate che utilizzano i mutex sono:

Thread call	descrizione
Pthread_mutex_init	crea un mutex
Pthread_mutex_destroy	elimina un mutex
Pthread_mutex_lock	tenta il lock e si blocca
Pthread_mutex_trylock	tenta il lock o fallisce
Pthread_mutex_unlock	rilascia un lock

Mutex nei Pthread

- I Pthread offrono un secondo meccanismo di sincronizzazione: le **variabili condizione**.
- Mutex sono buoni per permettere o meno l'accesso ad una regione critica.
- Le variabili condizione permettono di bloccare i processi se non si verificano alcune condizioni.
- Alcune chiamate sono:

Thread call	descrizione
Pthread_cond_init	crea una variabile condizione
Pthread_cond_destroy	elimina una variabile condizione
Pthread_cond_wait	si blocca in attesa di un segnale
Pthread_cond_signal	segnale di risveglio per un thread
Pthread_cond_broadcast	risveglia tutti i thread addormentati

Monitor

- Con i **semafori** e **mutex** la comunicazione tra processi sembra apparentemente facile, **invece è difficile scrivere programmi corretti.**
- Un **monitor** è un insieme di procedure, variabili e dati strutturati tutti raggruppati in un particolare tipo di modulo o pacchetto.
- I processi possono richiedere le procedure in un **monitor** ogni volta che vogliono, ma non possono accedere direttamente le strutture dati interne del **monitor**.
- I **monitor** hanno una proprietà importante che li rende utili per realizzare la mutua esclusione: in un dato istante, **un solo processo per volta può essere attivo in un monitor.**