Esercitazione 3
Monitor

14 Novembre 2013

Esercizio

Si consideri la pista di pattinaggio sul ghiaccio di una località turistica montana.

La pista di pattinaggio è aperta a tutti.

In particolare i clienti dell'impianto si suddividono in due categorie: **principianti e esperti**. Ogni gruppo di principianti, durante la permanenza all'interno della pista, viene accompagnato da un **istruttore** messo a disposizione dalla società che gestisce l'impianto; a questo proposito, si supponga che il numero totale di istruttori sia **NI**.

I pattinatori entrano ed escono dalla pista a **gruppi omogenei** (ogni gruppo è formato solo da principanti o solo da esperti) e **monolitici** (ogni pattinatore fa parte dello stesso gruppo, sia in ingresso che in uscita), ognuno caratterizzato da una consistenza numerica data.

La capacità della pista è limitata dal valore MAX, che esprime il numero massimo di pattinatori che possono essere in pista contemporaneamente (gli istruttori non vengono conteggiati).

Inoltre, il regolamento dell'impianto prevede che debba essere sempre rispettata la relazione:

$$P \geq E$$

dove: P è il numero dei principianti in pista, e E è il numero degli esperti in pista.

Si sviluppi un'applicazione concorrente in C/pthread che rappresenti i gruppi di pattinatori come thread concorrenti.

In particolare, la soluzione deve implementare una politica di sincronizzazione dei thread che rispetti le specifiche date, ed inoltre i vincoli seguenti:

- nell'**accesso** alla pista: i principianti abbiano la precedenza sugli esperti; a parità di categoria, si privilegino i gruppi meno numerosi.
- nell'**uscita** dalla pista: gli esperti abbiano la precedenza sui principianti; a parità di categoria, si privilegino i gruppi più numerosi.

Spunti e suggerimenti (1)

Quali thread?

- thread iniziale
- gruppo di pattinatori principianti
- gruppo di pattinatori esperti

E gli istruttori?

Quale risorsa comune?

- **pista da pattinaggio: (posti, istruttori)**

- associamo alla Pista un "***monitor***", che controlla gli accessi in base alla specifica politica di accesso. La sincronizzazione viene realizzata mediante variabili condizione.

Struttura thread

```
void *gruppoPrincipiante(void * arg)
{ int num; // numero di componenti
      InPistaP(&Pista, num); // possibilità di attesa
       /* simulazione uso pista*/
      OutPistaP(&Pista, num); // possibilità di attesa!

}

void *gruppoEsperto(void * arg)
{ int num; // numero di componenti
      InPistaE(&Pista, num); // possibilità di attesa
       /* simulazione uso pista*/
      OutPistaE(&Pista, num);
}
```

Spunti e suggerimenti (2)

Strumenti di sincronizzazione:

il monitor esercita due livelli di sincronizzazione:

1. mutua esclusione dei processi nell'esecuzione delle operazioni public: nei pthtread va realizzata esplicitamente attraverso un mutex: *lock;*

1. controllo dell'ordine con il quale i processi hanno accesso alla risorsa: definizione di variabili condizione

Spunti e suggerimenti (3)

Quante/quali variabili condizione?

- **I Principianti e Esperti possono sospendersi in ingresso:**
  Se maxG è la massima numerosità di ogni gruppo, sarà necessario prevedere maxG code per ogni tipo di thread

➔ definiamo 2 array di condition:

- **CodaP_IN[maxG] // principianti in attesa di entrare**
- **CodaE_IN[maxG] // esperti in attesa di entrare**

- **I Principianti possono sospendersi anche in uscita (P ≥ E):**

definiamo 1 array di condition:

- **CodaP_OUT[maxG]**

Spunti e suggerimenti (4)

Politica di allocazione dei posti nella pista basata su priorità:

Entrata		*priorità*	Uscita
Principianti	(1,..maxG)	↑	[Esperti]
Esperti	(1,..maxG)		Principianti (maxG,..1)

Spunti e suggerimenti (5)

Impostazione della struttura dati gestita dal monitor:

```
typedef struct{
  pthread_mutex_t lock;
  pthread_cond_t codaIngresso[2][maxG];
  pthread_cond_t codaUscita[maxG];
  int sospIngresso[2][maxG];
  int sospUscita[maxG];
  int inPista[2];
  int istruttori;
} pista;
```

Spunti e suggerimenti (6)

Data la complessità dei vincoli, può essere utile definire alcune funzioni di utilità:

Verifica di processi più prioritari in attesa:

```
int piu_prioritari_P_IN(pista *p, int num)
{
< restituisce 1 se c'è almeno un gruppo sospeso P
   sospeso (in ingresso) di consistenza numerica minore
   di num, altrimenti 0 >
}

int piu_prioritari_E_IN(pista *p, int num)
{
< restituisce 1 se c'è almeno un gruppo sospeso (in
   ingresso) di priorità superiore al gruppo E di num
   persone>
}

.. Ecc.
```