Esercitazione n.2
25 Ottobre 2024

Obiettivi:

- Programmazione concorrente con **`pthreads`**:
  - **Sincronizzazione** thread posix tramite **semafori condizione e semafori privati**

Esercizio 2.1

Si consideri un parco a tema al quale i visitatori possono accedere singolarmente.

Il parco ha una **capacità limitata pari a MaxP** : pertanto non può accogliere più di MaxP persone contemporaneamente.

Il parco è molto esteso e non può essere visitato a piedi; pertanto per la visita del parco ogni visitatore utilizzerà un veicolo messo a disposizione dal parco. In particolare vengono offerti 2 tipi di veicoli:

- **biciclette**. Si assuma che il numero totale di bici a disposizione sia MaxB
- **monopattini** elettrici. Si assuma che il numero totale di monopattini a disposizione sia **MaxM**.

Ogni visitatore:

1. **richiede l'accesso** al parco, acquisendo contestualmente un veicolo a sua scelta (bici o monopattino);
2. visita il parco per un tempo arbitrario
3. **esce** dal parco, restituendo il veicolo usato per la visita.

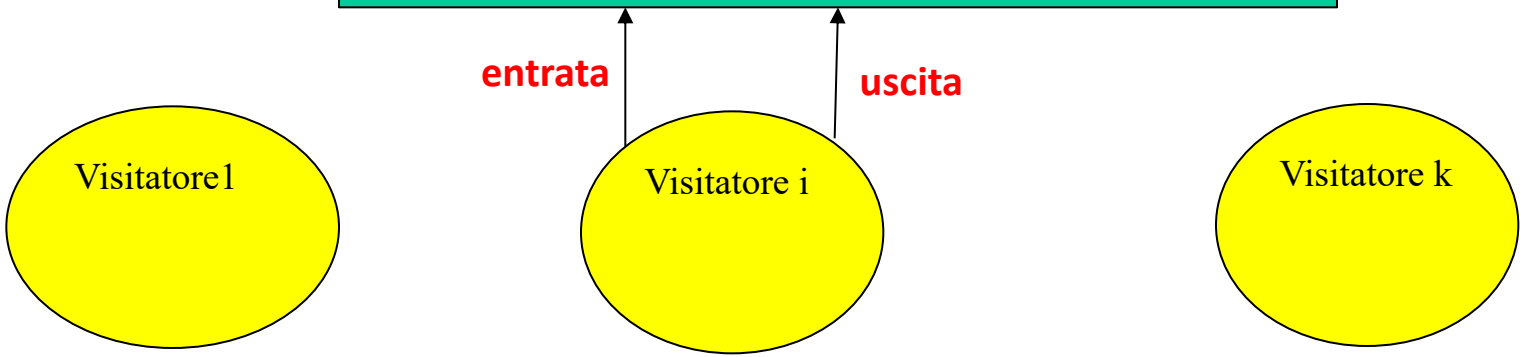
Realizzare un'applicazione concorrente C/pthread, nella quale ogni visitatore sia rappresentato da un thread distinto e la sincronizzazione venga ottenuta tramite **semafori** posix.

Impostazione es 2.1

Gestore del parco

Parco : - MaxP posti
- MaxB Bici
- Max M monopattini

- **Quanti semafori?**
- **Di che tipo?**



Impostazione es 2.1

- **Quanti semafori?**

L'entrata di ogni visitatore V_i è possibile solo se la seguente condizione è verificata:

c'è posto per V_i nel parco ed è disponibile il veicolo richiesto.

Altrimenti V_i aspetta.

Per implementare l'attesa uso un semaforo S.

- **Di che tipo?**

→ **posso usare un semaforo condizione:**

- **i thread in entrata che non soddisfano la condizione di sincronizzazione si sospendono con una p(&S) (sem_wait(&S))**
- **ogni thread in uscita, liberando un posto e un veicolo, può riattivare uno dei processi in attesa con una v(&S) (sem_post(&S))**

Gestore

```
typedef struct{
        int posti_liberi;
        int bici_libere;
        int monop_liberi;
        sem_t S;                    //semaforo condizione
        int sospesi;
        ...
        pthread_mutex_t m;
} parco;
```

Impostazione es 2.1

Schema di sincronizzazione:

attesa circolare o passaggio del testimone ?

Abbiamo un unico semaforo condizione sul quale si sospendono sia i thread in attesa di una bici, sia quelli in attesa di un monopattino → non è detto che il primo processo in attesa, a seguito di una v (sem_post) possa entrare

Esempio:

Tc	Tb	Ta
attende monop.	attende una bici	attende una bici

thread in attesa su S

Un thread Te esce dal parco restituendo un monopattino; occorre riattivare Tc, ma non è il primo della coda → è necessario chiamare la v() per 3 volte.

dovendo fare un ciclo di v(), l'unico schema possibile è l'attesa circolare.

Impostazione es 2.1

Schema di sincronizzazione:

l'unico schema possibile è l'attesa circolare.

```
Gestore G;
```

```
void entrata(...)
{   ...
    while (<non c'è posto> ||
           <non c'è il veicolo desiderato>)
    {     G.sospesi ++;
          ...
          sem_wait(&G.S);
          ...
          G.sospesi--;
    }
    <occupa un posto>
    <prende il veicolo>
    ...
}
```

```
void uscita(...)
{   ...
    <libera un posto>
    <restituisce il veicolo>
    for (int i=0; i< G.sospesi; i++)
          sem_post(&G.S);
    ...
}
```

Esercizio 2.2

Variante dell'esercizio 2.1.

I visitatori possono accedere al parco a **gruppi**. Ogni gruppo può essere composto al massimo da **5 persone**.

Il parco ha una **capacità limitata pari a MaxP** : pertanto non può accogliere più di **MaxP** persone contemporaneamente.

Il parco è molto esteso e non può essere visitato a piedi: per la visita del parco sono **MaxA auto elettriche**, destinate al trasporto di gruppi di 1-5 persone. Si assuma che il numero totale di auto a disposizione sia MaxA

Ogni gruppo:

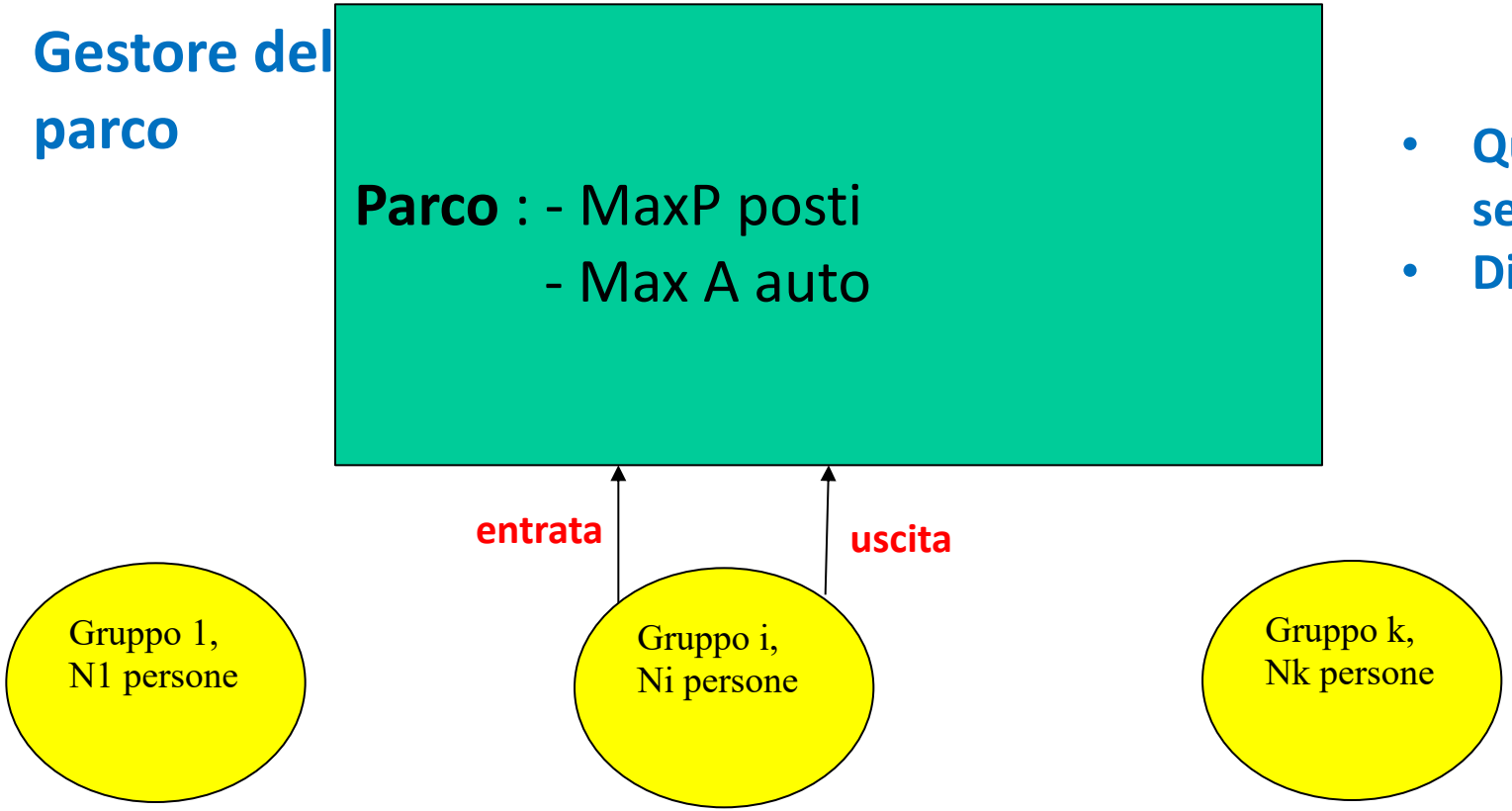
1. **richiede l'accesso** al parco, acquisendo contestualmente l'auto;
2. visita il parco per un tempo arbitrario
3. **esce** dal parco, restituendo l'auto usata per la visita

Realizzare un'applicazione concorrente C/pthread, nella quale ogni gruppo sia rappresentato da un thread distinto e la sincronizzazione venga ottenuta tramite semafori posix.

L'applicazione dovrà garantire che l'ordine di ingresso al parco **rispetti l'ordine cronologico di arrivo alla biglietteria**.

Impostazione es 2.1

Gestore del parco



- Quanti semafori?
- Di che tipo?

Impostazione es 2.2

Quanti semafori?

L'entrata di ogni gruppo G_i è possibile solo se la seguente condizione è verificata:

- **c'è posto per G_i nel parco –AND-**
- **è disponibile un'auto -AND-**
- **non c'è nessun gruppo arrivato prima di G_i in attesa.**

Altrimenti G_i aspetta.

Per implementare l'attesa uso un semaforo S.

Di che tipo?

→ **posso usare un semaforo condizione:**

- **i gruppi in entrata che non soddisfano la condizione di sincronizzazione si sospendono su S con una p(&S) (sem_wait(&S))**
- **ogni gruppo in uscita, liberando posti e un veicolo, può riattivare uno o più processi in attesa con una v(&S) (sem_post(&S)), tenendo conto dell'ordine di arrivo.**

Gestore

```
typedef struct{
        int posti_liberi;
        int auto_libere;
        int ordine_arrivo; // quanti processi arrivati
        sem_t S;                  //semaforo condizione
        int gruppi_sospesi[NUM_THREADS];//indice: ordine di arrivo
        pthread_mutex_t m;
} parco;
```

Un processo richiedente si deve **sospendere se c'è in coda almeno un processo arrivato prima.** →la variabile **ordine_arrivo** permette **tenere traccia dell'ordine di arrivo** di ogni processo:

Un processo arrivato i-simo al parco, **se attende**, incrementerà **gruppi_sospesi[i-1].**

Impostazione es 2.2

Schema di sincronizzazione:

attesa circolare o passaggio del testimone ?

Un processo in uscita, liberando più posti, può riattivare più di un processo un attesa di entrare.

→ l'unico schema possibile è l'attesa circolare.

Esercizio 2.3

Si consideri il problema posto dall'esercizio 2.2.

Si realizzi una soluzione alternativa in cui le richieste di accesso vengano servite **dando la precedenza ai gruppi più piccoli.**

Cosa cambia?

Quanti semafori?

Di che tipo?

Semaforo privato per un processo (richiamo)

- Un semaforo s si dice **privato** per un processo quando solo tale processo può eseguire la primitiva **P** sul semaforo s.

- La primitiva **V** sul semaforo può essere invece eseguita anche da altri processi.

- Un semaforo privato viene inizializzato con il valore zero.

Semaforo privato per una classe di processi

Consideriamo una politica di sincronizzazione basata su priorità: ad ogni processo è associato un valore che ne stabilisce la priorità.

Classe di processi C_k: insieme dei processi che hanno lo stesso valore di priorità k.

- Un semaforo s_k si dice **privato per una classe di processi C_k** quando solo i processi appartenenti a C_k possono eseguire la primitiva **P** su s_k.

- La primitiva **V** sul semaforo può essere invece eseguita da qualunque processo.

- Un semaforo privato per una classe di processi viene inizializzato con il valore **0**.

Suggerimenti

```
semaphore priv[maxprio];
int sosp[maxprio];
pthread_mutex mux;

void Richiesta(int id, int prio)
{       pthread_mutex_lock(mux)
        while (<condizione di acquisiz. non soddisfatta>)
        {       sosp[prio]++;
                pthread_mutex_unlock(mux);
                p(priv[prio]);
                pthread_mutex_lock(mux);
        }
        …
         pthread_mutex_unlock(mux);
}
```

Suggerimenti

```
void Rilascio(int id, int prio)
{       pthread_mutex_lock(mux)
        <liberazione risorsa>
        while (<c'è qualcuno da risvegliare>)
        {       <seleziona il proc. sospeso di massima prio k>
                sosp[k]--;
                v(priv[k]);
        }
        pthread_mutex_unlock(mux);
}
```

Esercizio 2.4

Politiche di sincronizzazione basate su priorità:

- Estendere il problema 1.3 (V. esercitazione 1) assumendo che la fase di visione del film vincitore richieda che ogni thread, prima di vedere il film, debba eseguire un'operazione di download del file.
- Si assuma che sia fissato un numero massimo MAX di download contemporanei consentiti, oltre al quale ogni ulteriore richiesta di download deve essere messa in attesa.
- Si assuma infine che il server, nell'autorizzare i download, applichi una politica che privilegi gli utenti in base alla media dei voti dati: la priorità deve essere data all'utente la cui media dei voti dati (arrotondata all'intero più vicino) è maggiore.

Suggerimenti

Fasi attraversate da ogni thread:

1. <partecipa a sondaggio>
2. <barriera di sincronizzazione>
3. <richiedi download film vincitore> ->Richiesta
4. ...download film...
5. <fine download> ->Rilascio
6. <visione film>

Osservazioni:

- Analogia con il pool di risorse equivalenti
- Politica con priorità: semafori privati ?