Sistemi Operativi

Unità 7: I Thread

Problemi di Sincronizzazione

Martino Trevisan
Università di Trieste
Dipartimento di Ingegneria e Architettura

Argomenti

1. Mutex e Semafori
2. Grafi di precedenza
3. Produttore e consumatore

Mutex e Semafori

Mutex e Semafori

I **Mutex** regolano l'accesso a una sezione critica:

- Solo un thread per volta può avere il lock
- Operazioni: `lock` `unlock`

I **Semafori** sono degli interi positivi condivisi:

- Simili a un contenitore di gettoni
- Operazioni: `post` `wait`

I *Semafori* sono un costrutto più generale

- Un *Semaforo* può facilmente essere usato come *mutex*

Mutex e Semafori

Costruire un Mutex con un Semaforo

Inizializzazione:

Mutex

```
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
```

Semaforo: deve essere inizializzato al valore 1

```
sem_t sem;
sem_init(&sem, 0, 1);
```

Mutex e Semafori

Costruire un Mutex con un Semaforo

Lock:

Mutex

```
pthread_mutex_lock(&lock);
```

Semaforo

```
sem_wait(&sem);
```

Release:

Mutex

```
pthread_mutex_unlock(&lock);
```

Semaforo

```
sem_post(&sem);
```

Mutex e Semafori

Costruire un Mutex con un Semaforo

Vedi implementazione in `esercizi/myMutex.c`

```c
typedef struct{
    sem_t s;
} myMutex;

myMutex myMutex_init(){
    myMutex m;
    sem_init(&(m.s), 0, 1);
    return m;
}

void myMutex_lock(myMutex * m){
    sem_wait( &(m->s) );
}

void myMutex_unlock(myMutex * m){
    sem_post( &(m->s) );
}
```

Mutex e Semafori

Costruire un Semaforo con un Mutex

Si può costruire un semaforo con un `mutex`, ma é **inefficiente**

- Un semaforo é un intero condiviso **positivo**
- Un mutex protegge l'accesso a questo intero

Funzionamento:

- In caso venga effettuato un decremento (`wait`) quando il semaforo ha valore 0:
  Il thread attende che un altro thread effettui un incremento (`post`)
- L'unico modo con cui si attendere, é **busy waiting**
  - Un ciclo `for` che verifica ripetutamente
  - Inefficiente

Mutex e Semafori

Costruire un Semaforo con un Mutex

Implementazione (by ChatGPT):

```c
struct semaphore {
    pthread_mutex_t mutex;
    int count;
};

void semaphore_init(struct semaphore *sem, int count) {
    pthread_mutex_init(&sem->mutex, NULL);
    sem->count = count;
}

void semaphore_wait(struct semaphore *sem) {
    pthread_mutex_lock(&sem->mutex);
    while (sem->count == 0) {
        pthread_mutex_unlock(&sem->mutex);
        pthread_mutex_lock(&sem->mutex);
    }
    sem->count--;
    pthread_mutex_unlock(&sem->mutex);
}

void semaphore_post(struct semaphore *sem) {
    pthread_mutex_lock(&sem->mutex);
    sem->count++;
    pthread_mutex_unlock(&sem->mutex);
}
```

Grafi di precedenza

Grafi di precedenza

I semafori sono pratici da usare per costruire **grafi di precedenza**

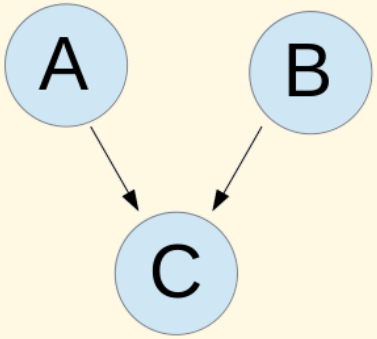
- Un insieme di task che devono essere eseguite in un ordine particolare

I grafi di precedenza modellano molto bene sistemi distribuiti e concorrenti

- Le **Reti di Petri** sono un astrazione per trattare grafi di precedenza con l'utilizzo di semafori
- Non vedremo

Grafi di precedenza

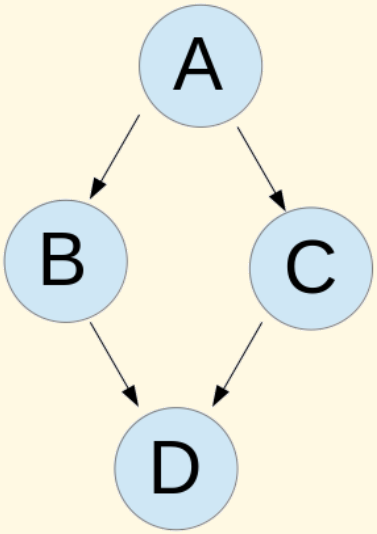
Esempio 1




```
sem_t s1;
void* t_A(void* arg){
    A();
    sem_post(&s1);
}
void* t_B(void* arg){
    B();
    sem_post(&s1);
}
void* t_C(void* arg){
    sem_wait(&s1);
    sem_wait(&s1);
    C();
}
```

Grafi di precedenza

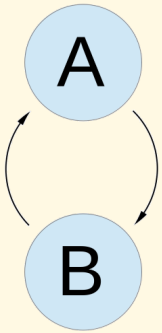
Esempio 2




```
sem_t s1, s2;
void* t_A(void* arg){
    A();
    sem_post(&s1);
    sem_post(&s1);
}
void* t_B(void* arg){
    sem_wait(&s1);
    B();
    sem_post(&s2);
}
void* t_C(void* arg){
    sem_wait(&s1);
    C();
    sem_post(&s2);
}
void* t_D(void* arg){
    sem_wait(&s2);
    sem_wait(&s2);
    D();
}
```

Grafi di precedenza

Esempio 3

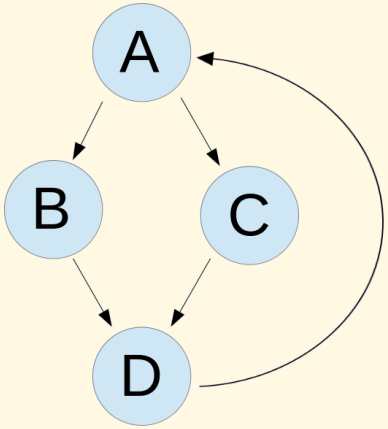



```c
sem_t s1, s2;
sem_init(&s1, 0, 1); // Inizializzato a 1
sem_init(&s2, 0, 0); // Inizializzato a 0
void* t_A(void* arg){
    while (1){
        sem_wait(&s1);
        A();
        sem_post(&s2);
    }
}
void* t_B(void* arg){
    while (1){
        sem_wait(&s2);
        B();
        sem_post(&s1);
    }
}
```

NOTA: esercizio uguale a lettore/scrittore visto in precedenza

Grafi di precedenza

Esempio 4




```
sem_t s1, s2, s3; // s1 inizializzata a 1, gli altri a 0
void* t_A(void* arg){
    while (1){
        sem_wait(&s1);
        A();
        sem_post(&s2);
        sem_post(&s2);
    }
}
void* t_B(void* arg){
    while (1){
        sem_wait(&s2);
        B();
        sem_post(&s3);
    }
}
void* t_C(void* arg){
    while (1){
        sem_wait(&s2);
        C();
        sem_post(&s3);
    }
}
void* t_D(void* arg){
    while (1){
        sem_wait(&s3);
        sem_wait(&s3);
        D();
        sem_post(&s1);
    }
}
```

Produttore e consumatore

Produttore e consumatore

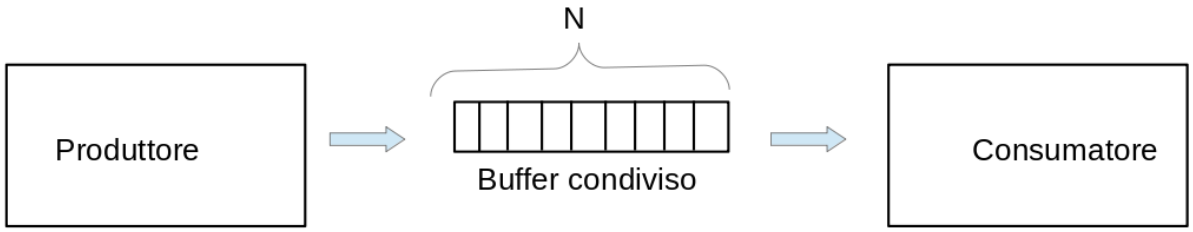
Il problema

Problema classico dell'informatica, applicabile in molti contesti

- Pacchetti di rete
- Calcolo parallelo

Definizione:

- Due thread comunicano tramite un buffer di grandezza limitata, che contiene massimo N oggetti
- Il thread *producer* inserisce gli oggetti nel buffer
- Il thread *consumer* estrae gli oggetti dal buffer, nell'ordine in cui sono stati inseriti



Produttore e consumatore

Struttura dati

Variabili Condivise tra Produttore e Consumatore:

```
<tipo> buffer [N]; // Il buffer
int contatore = 0; // Indicazione di elementi usati nel buffer
```

Variabili NON Condivise:

```
int in;  // Indice dove il produttore inserisce in buffer
         // Gestito in artitmetica Modulo N
int out; // Indice dove il consumatore estrare
```

Produttore e consumatore

Algoritmo

Produttore:

```
while (1) {
    while (contatore == BUFFER_SIZE); /* non fa niente se il buffer è pieno */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    contatore++;
}
```

Consumatore:

```
while (1) {
    while (contatore == 0); /* non fa niente se il buffer è vuoto */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    contatore--;
}
```

Produttore e consumatore

Problema di sincronizzazione

Il codice della slide precedente non funziona.

- C'è accesso concorrente a variabili condivise
  Le istruzioni `contatore++;` e `contatore--;` non possono essere eseguite simultaneamente
- Alcuni incrementi o decrementi potrebbero essere persi
- Il programma ha un **baco**

Produttore e consumatore

Problemi da risolvere

1. Accesso concorrente a `contatore`: è possibile usare un **mutex**

Nota: non c'è mai accesso concorrente a stesso elemento di `buffer`

2. Attesa efficiente:

Le istruzioni `while (contatore == BUFFER_SIZE);` e `while (contatore == 0);` effettuano **Busy Waiting**

- Controlla continuamente la variabile `contatore`
- Spreco enorme di CPU

Produttore e consumatore

Soluzione classica

Si usano due semafori

- Semaforo `empty`: conta quanti posti **liberi** ci sono nel buffer
- Semaforo `full`: conta quanti posti **occupati** ci sono nel buffer

La variabile `contatore` diventa inutile. I semafori già contano quanti posti liberi e occupati ci sono

Soluzione completa nel **materiale** in `esercizi/myProdCons.c`

Produttore e consumatore

Soluzione classica

Inizializzazione

```
<tipo> buffer [N];
sem_t empty, full;

int main(){
    ...
    sem_init(&empty, 0, N); /* Inizialmente N posti liberi */
    sem_init(&full, 0, 0);  /* e 0 occupati */
    ...
}
```

Produttore e consumatore

Soluzione classica

Produttore

```
int in = 0;
while (1) {
    sem_wait(&empty); /* Attende che ci posto libero nel buffer */
    buffer[in] = next_produced;
    in = (in + 1) % N;
    sem_post(&full); /* Un dato un più nel buffer */
}
```

Produttore e consumatore

Soluzione classica

Consumatore

```
int out = 0;
while (1) {
    sem_wait(&full); /* Attende che ci siano dati da consumare */
    <type> next_consumed = buffer[out];
    out = (out + 1) % N;
    sem_post(&empty); /* Un posto libero in più nel buffer */
}
```