

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

```
#define MAXPAROLA 30
#define MAXRIGA 80
```

```
int main(int argc, char *argv[])
{
    int freq[MAXPAROLA]; /* vettore di contatori
    delle frequenze delle lunghezze delle parole */
    char riga[MAXRIGA];
    int i, inizio, lunghezza;
    FILE *f;
```

```
for(i=0; i<MAXPAROLA; i++)
    freq[i]=0;
```

```
if(argc != 2)
```

```
{
    fprintf(stderr, "ERRORE: serve un parametro con il nome del file\n");
    exit(1);
}
```

```
f = fopen(argv[1], "r");
if(f==NULL)
```

```
{
    fprintf(stderr, "ERRORE: impossibile aprire il file %s\n", argv[1]);
    exit(1);
}
```

```
while( fgets( riga, MAXRIGA, f ) != NULL )
```



Sincronizzazione

Problemi di sincronizzazione tipici

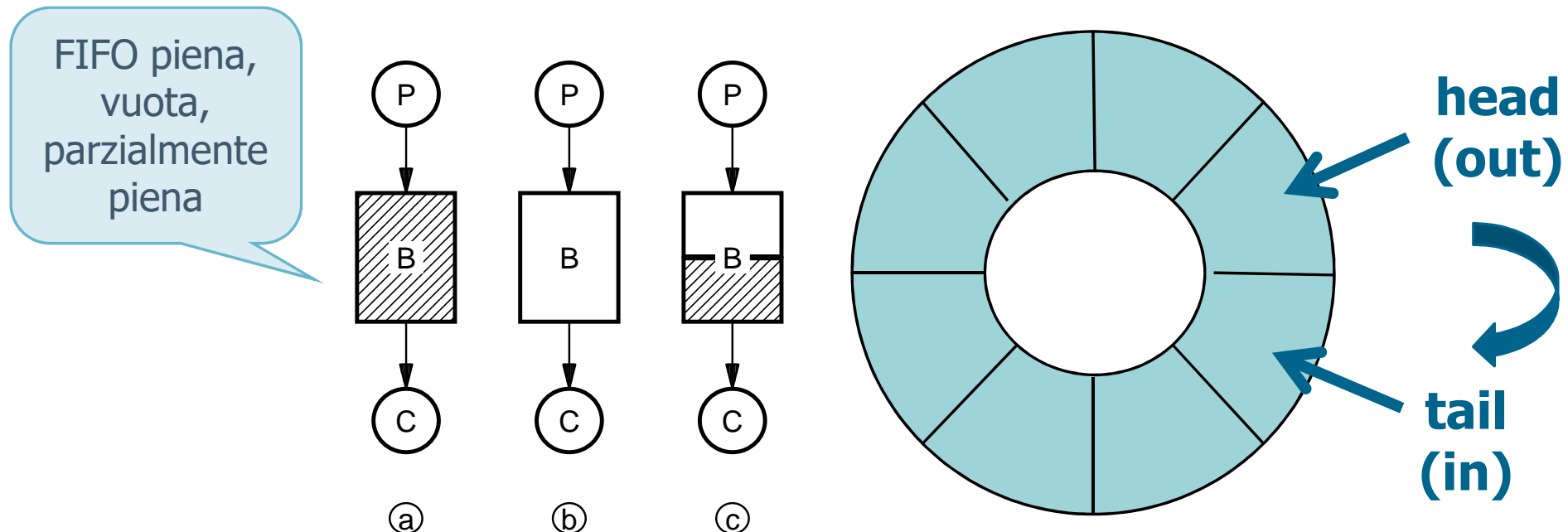
Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

Produttore-Consumatore

- ❖ Produttore e consumatore con memoria limitata
 - Utilizza un buffer circolare di dimensione **SIZE** per memorizzare gli elementi prodotti e da consumare
 - Il buffer circolare implementa una coda (queue) FIFO (First-In First-Out)



Accesso sequenziale

```
#define SIZE ...  
...  
int queue[SIZE];  
int tail, head;  
...  
void init () {  
    tail = 0;  
    head = 0;  
    n = 0;  
}
```

FIFO standard (non ADT)

```
void enqueue (int val) {  
    if (n>SIZE) return;  
    queue[tail] = val;  
    tail=(tail+1)%SIZE;  
    n++;  
    return;  
}
```

```
void dequeue (int *val) {  
    if (n<=0) return;  
    *val=queue[head];  
    head=(head+1)%SIZE;  
    n--;  
    return;  
}
```

Accesso sequenziale vs parallelo

- ❖ Nell'accesso sequenziale **enqueue** e **dequeue** non sono mai contemporanee
- ❖ Nell'accesso parallelo si possono avere due casi
 - **1 solo produttore e 1 solo consumatore**
 - Operazioni di enqueue e dequeue agiscono su estremità diverse della coda ma la variabile n è comunque condivisa
 - **P produttori e C consumatori**
 - Come il caso precedente con in più operazioni di accesso sullo stesso estremo della coda

Accesso concorrente: Versione 1

- ❖ Per un accesso parallelo con 1 produttore e 1 consumatore
 - Occorre inserire
 - Un semaforo "full" che conta il numero di elementi pieni
 - Un semaforo "empty" che conta il numero di elementi vuoti
 - Il contatore n può essere eliminato

Accesso concorrente: Versione 1

```
#define SIZE ...  
...  
int queue[SIZE];  
int tail, head;  
...  
void init () {  
    tail = 0;  
    head = 0;  
}
```

FIFO standard (non ADT)
senza la variabile n

```
void enqueue (int val) {  
    queue[tail] = val;  
    tail=(tail+1)%SIZE;  
    return;  
}
```

```
void dequeue (int *val) {  
    *val=queue[head];  
    head=(head+1)%SIZE;  
    return;  
}
```

Accesso concorrente: Versione 1

1 Produttore
1 Consumatore

Invece di n utilizza
Elementi pieni
Elementi vuoti

```
init (full, 0);  
init (empty, SIZE);
```

```
Producer () {  
    int val;  
    while (TRUE) {  
        produce (&val);  
        wait (empty);  
        enqueue (val);  
        signal (full);  
    }  
}
```

```
Consumer () {  
    int val;  
    while (TRUE) {  
        wait (full);  
        dequeue (&val);  
        signal (empty);  
        consume (val);  
    }  
}
```

Accesso concorrente: Versione 2

- ❖ La soluzione 1 è simmetrica
 - Il produttore produce posizioni piene
 - Il consumatore produce posizioni vuote
- ❖ Può essere facilmente **estesa** al caso in cui coesistano più produttori e più consumatori
 - Produttori e consumatori operano su estremità opposte del buffer
 - Possono farlo **contemporaneamente**
 - Purchè la coda non sia piena oppure vuota
 - Due produttori oppure due consumatori devono invece agire in **mutua esclusione**

Accesso concorrente: Versione 2

P Produttori
C Consumatori

Occorre forzare ME
tra P e tra C

```
init (full, 0);  
init (empty, SIZE);  
init (MEp, 1);  
init (MEc, 1);
```

```
Producer () {  
    int val;  
    while (TRUE) {  
        produce (&val);  
        wait (empty);  
        wait (MEp);  
        enqueue (val);  
        signal (MEp);  
        signal (full);  
    }  
}
```

```
Consumer () {  
    int val;  
    while (TRUE) {  
        wait (full);  
        wait (MEc);  
        dequeue (&val);  
        signal (MEc);  
        signal (empty);  
        consume (val);  
    }  
}
```

Readers & Writers

❖ Problema classico

- Courtois et al. [1971]
- Condividere un base dati tra due insiemi di processi concorrenti
 - Una classe di processi detta **Reader** a cui è consentito accedere a un data-base in **concorrenza**
 - Una classe di processi detta **Writer** a cui è consentito accedere al data-base in **mutua esclusione** sia con altri processi Writers sia con i processi Readers
- Costrutto spesso utilizzato per verificare nuove primitive di sincronizzazione

Readers & Writers

- ❖ Esistono due versioni del problema
 - Primo problema o problema con precedenza ai reader
 - Secondo problema o problema con precedenza ai writer
- ❖ Obiettivi comuni
 - Rispettare il protocollo di precedenza
 - Massimizzare la concorrenza

Precedenza ai reader

- ❖ Dare precedenza ai reader significa
 - Privilegiare l'accesso dei reader rispetto a quello dei writer ovvero
 - I reader non devono attendere a meno che un writer sia nella SC
- ❖ Protocollo di accesso
 - I reader possono accedere in concorrenza al database
 - Sino a quando arrivano reader i writer attendono
 - Quando anche l'ultimo reader termina allora si può svegliare un writer (o un reader ... dipende dallo scheduler)

Precedenza ai reader: Versione 1

Reader

```
wait (meR);  
  nR++;  
  if (nR==1)  
    wait (w);  
signal (meR);  
...  
lettura  
...  
wait (meR);  
  nR--;  
  if (nR==0)  
    signal (w);  
signal (meR);
```

```
nR = 0;  
init (meR, 1);  
init (w, 1);
```

Writer

```
wait (w);  
...  
scrittura  
...  
signal (w);
```

Precedenza ai reader: Versione 2

Reader

```
wait (meR);  
  nR++;  
  if (nR==1)  
    wait (w);  
signal (meR);  
...  
lettura  
...  
wait (meR);  
  nR--;  
  if (nR==0)  
    signal (w);  
signal (meR);
```

```
nR = 0;  
init (meR, 1);  
init (meW, 1);  
init (w, 1);
```

Si rafforza la precedenza ai R
(la signal(w) libera un R)

Writer

```
wait (meW);  
wait (w);  
...  
scrittura  
...  
signal (w);  
signal (meW);
```

Conclusioni

- ❖ La soluzione utilizza
 - Una variabile globale (nR) che conta il numero di reader nella SC
 - Un semaforo di mutua esclusione per la manipolazione della variabile nR (meR)
 - Un semaforo di mutua esclusione per più writer o per un reader e i writer (w)
 - Un semaforo di mutua esclusione per writer (meW)
- ❖ I writer sono soggetti a **starvation**, in quanto possono attendere per sempre
- ❖ Sono possibili soluzioni più complesse senza starvation da parte dei W

Precedenza ai writer

- ❖ Dare precedenza ai writer significa
 - Un writer pronto deve attendere il meno possibile
- ❖ Protocollo di accesso
 - Ogni writer deve attendere che finiscano i reader
 - Ogni writer ha priorità su tutti i reader

Precedenza ai writer

```
nR = nW = 0;  
init (w, 1); init (r, 1);  
init (meR, 1); init (meW, 1);
```

Reader

```
wait (r);  
wait (meR);  
nR++;  
if (nR == 1)  
wait (w);  
signal (meR);  
signal (r);  
...  
lettura  
...  
wait (meR);  
nR--;  
if (nR == 0)  
signal (w);  
signal (meR);
```

Writer

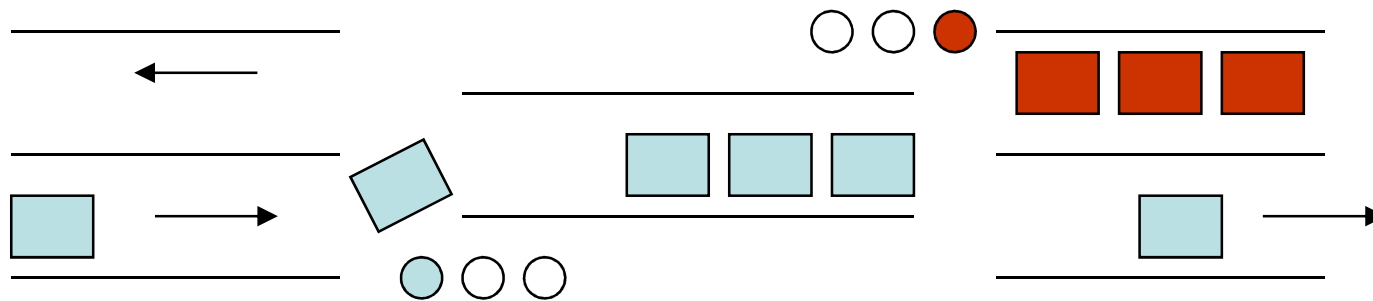
```
wait (meW);  
nW++;  
if (nW == 1)  
wait (r);  
signal (meW);  
wait (w);  
...  
scrittura  
...  
signal (w);  
wait (meW);  
nW--;  
if (nW == 0)  
signal (r);  
signal (meW);
```

Conclusioni

- ❖ La soluzione utilizza
 - Due variabili globali (nR e nW) per il conteggio dei reader e dei writer
 - Due semafori di mutua esclusione (meR e meW) per la manipolazione delle variabili nR e nW
 - Due semafori di mutua esclusione reader/writer (r e w)
- ❖ I reader sono soggetti a starvation, in quanto possono attendere per sempre
- ❖ Sono possibili soluzioni più complesse senza starvation

Il "Tunnel a senso alternato"

- ❖ In un tunnel a senso alternato
 - Permettere a qualsiasi numero di auto (processi) di procedere nella stessa direzione
 - Se c'è traffico in una direzione bloccare il traffico nella direzione opposta



Il "Tunnel a senso alternato"

- ❖ Estensione del problema dei Readers-Writers con due insiemi di reader
- ❖ Struttura dati
 - Due variabili globali di conteggio ($n1$ e $n2$), una per ciascun senso di marcia
 - Due semafori ($s1$ e $s2$), uno per ciascun senso di marcia
 - Un semaforo globale di attesa ($busy$)
- ❖ Nella sua implementazione base può provocare starvation delle auto in una direzione rispetto all'altra

Soluzione

```
n1 = n2 = 0;  
init (s1, 1); init (s2, 1);  
init (busy, 1);
```

left2right

```
wait (s1);  
  n1++;  
  if (n1 == 1)  
    wait (busy);  
signal (s1);  
...  
Run (left to right)  
...  
wait (s1);  
  n1--;  
  if (n1 == 0)  
    signal (busy);  
signal (s1);
```

right2left

```
wait (s2);  
  n2++;  
  if (n2 == 1)  
    wait (busy);  
signal (s2);  
...  
Run (right to left)  
...  
wait (s2);  
  n2--;  
  if (n2 == 0)  
    signal (busy);  
signal (s2);
```

I 5 filosofi

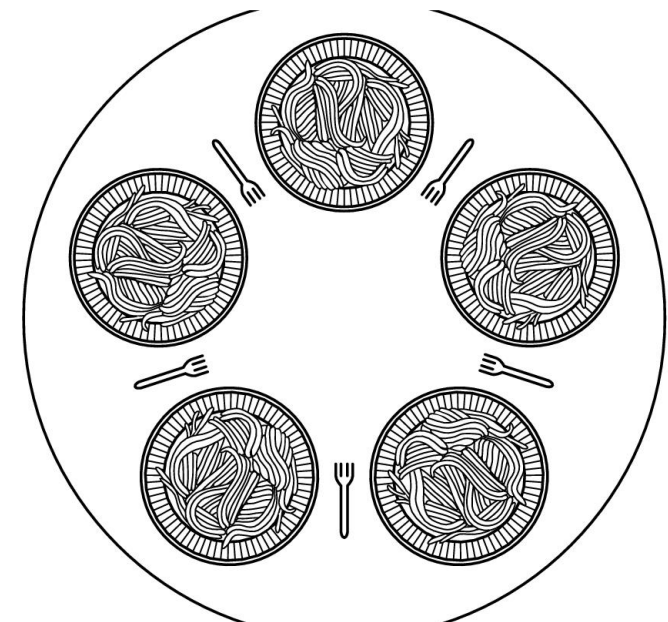
- ❖ Modello del caso in cui diverse risorse sono comuni a diversi processi concorrenti
- ❖ Dovuto a Dijkstra [1965]
- ❖ Definizione del problema
 - Un tavolo è imbandito con
 - 5 piatti di riso
 - 5 bastoncini (cinesi) ciascuno tra due piatti
 - Intorno al tavolo siedono 5 filosofi
 - I filosofi pensano oppure mangiano
 - Per mangiare ogni filosofo ha bisogno di due bastoncini
 - I bastoncini possono essere ottenuti uno alla volta



Modello 0

❖ Soluzioni "filosofiche"

- Insegnare ai filosofi a mangiare con 1 solo bastoncino
- Fornire più di 5 bastoncini
- Permettere solo al più a 4 filosofi di sedersi al tavolo
- Forzare asimmetria
 - I filosofi di posizione pari prendono la forchetta sinistra per prima
 - I filosofi di posizione dispari prendono la forchetta destra per prima

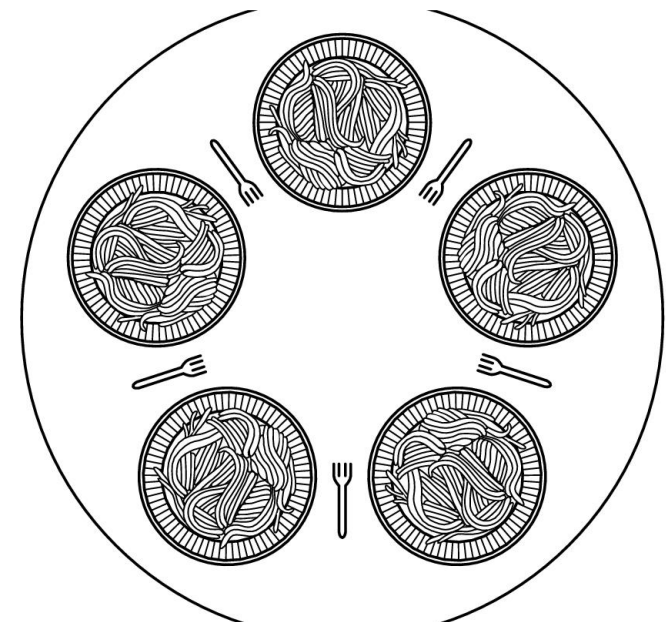


Modello 1

- ❖ Utilizzare un unico semaforo binario (mutex) per proteggere l'unica risorsa "cibo"
 - Annulla la concorrenza
 - Un solo filosofo mangia (potrebbero farlo in due)

```
init (mutex, 1);
```

```
while (true) {  
    Pensa ();  
    wait (mutex);  
    Mangia ();  
    signal (mutex);  
}
```



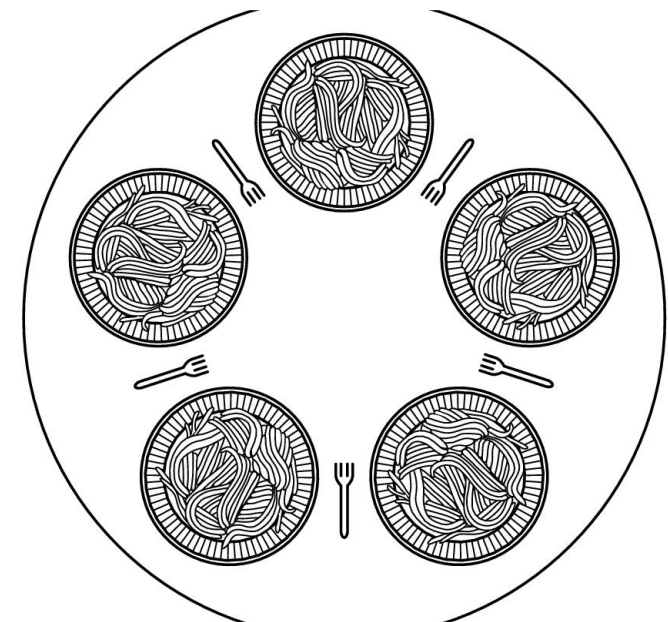
Modello 2

- ❖ Avere un semaforo per bastoncino
 - Può causare deadlock

```
init (chopstick[0], 1);  
...  
init (chopstick[4], 1);
```

$i \in [0, 4]$

```
while (true) {  
  Pensa ();  
  wait (chopstick[i]);  
  wait (chopstick[(i+1)mod5]);  
  Mangia ();  
  signal (chopstick[i]);  
  signal (chopstick[(i+1)mod5]);  
}
```



Soluzione

❖ Struttura dati

- Uno stato per ogni filosofo (THINKING, HUNGRY, EATING)
- Un semaforo per ogni filosofo (per l'accesso al cibo)
- Un semaforo ulteriore unico per l'accesso alla variabile di stato del filosofo stesso

```
while (TRUE) {  
    think ();  
    takeForks (i);  
    eat ();  
    putForks (i);  
}
```

Soluzione

```
int state[N]
init (mutex, 1);
init (sem[0], 0); ...; init (sem[4], 0);
```

```
takeForks (int i) {
    wait (mutex);
    state[i] = HUNGRY;
    test (i);
    signal (mutex);
    wait (sem[i]);
}
```

```
putForks (int i) {
    wait (mutex);
    state[i] = THINKING;
    test (LEFT);
    test (RIGHT);
    signal (mutex);
}
```

```
test (int i) {
    if (state[i]==HUNGRY && state[LEFT]!=EATING &&
        state[RIGHT]!=EATING) {
        state[i] = EATING;
        signal (sem[i]);
    }
}
```