

PROGRAMMAZIONE CONCORRENTE

Controllo d'Errore utile per debug.

**Problemi Classici di programmazione
concorrente risolti usando i Posix Thread.**

**Concetti e Definizioni Formali di
programmazione concorrente.**

Controllo d'Errore utile per debug

Preliminari: Controllo d'Errore e Debug(1)

Funzioni per semplificare il controllo d'errore:

Nella lezione in laboratorio, molti si sono accorti di un problema del loro codice solo grazie al fatto che la funzione pthread_mutex_lock() restituiva un risultato 22 (EINVAL).

Morale della favola: e' **INDISPENSABILE** effettuare il **controllo del risultato** restituito dalle funzioni di libreria dei thread, per capire se tutto procede bene o no.

Purtroppo, quando vi descrivo del codice C, la presenza del codice che controlla il risultato d'errore rende il codice stesso meno facile da seguire e da capire.

Per semplificarvi la comprensione, nei successivi esempi di programmi scritti in C, che vi metto a disposizione via web, adottero' la seguente tecnica semplificativa:

Invece di chiamare le funzioni standard dei pthread, **usero' alcune mie funzioni** DBG,

- che **chiamano le funzioni standard dei pthread** ,
- che **contengono anche il codice per il riconoscimento dell'errore** ,
- e che, a richiesta, **stampano una stringa quando vengono eseguite**.

La gestione dell'errore implementata in queste mie funzioni e' esageratamente basica:

in caso di errore le mie funzioni fanno terminare brutalmente il processo dopo avere emesso un messaggio d'errore.

Lo scopo e' facilitarvi la scrittura del codice ma allo stesso tempo inserire il controllo dell'errore e stampare qualcosa a video quando vengono eseguite, per debugging.

Controllo d'Errore e Debug (2)

ho implementato poche funzioni che hanno **lo stesso nome** delle principali funzioni per i pthread, con in piu' il **prefisso DBG (DeBuG)**. Ad esempio

pthread_mutex_lock -> **DBGpthread_mutex_lock**

Queste funzioni DBG prendono gli stessi argomenti delle funzioni non DBG e in piu' aggiungono in fondo un altro argomento, un puntatore ad una stringa C

```
int      pthread_mutex_lock ( pthread_mutex_t *mutex );
```

```
int DBGpthread_mutex_lock ( pthread_mutex_t *mutex , const char *label );
```

tali funzioni devono perciò essere chiamate come quelle dei pthread aggiungendo una stringa che identifica in che punto del vostro codice o in che funzione voi state chiamando la funzione.

```
void *decrementa(void *arg) {    int rc;  
    printf( " lock decrementa \n" );    ←  
    rc = pthread_mutex_lock ( &mutexdata );  
    if(rc!=0) {  
        printf( "pthread_mutex_lock failed %d - Quit\n", rc );  
        exit(1);    ←  
    }  
    .....  
}
```

stampa per debug

messaggio d'errore

terminazione causa di errore

```
void *decrementa(void *arg) {  
    DBGpthread_mutex_lock( &mutexdata , "decrementa" );  
    .....  
}
```

chiamata alla nuova funzione, il codice diventa piu' semplice

Controllo d'Errore e Debug (3)

L'implementazione delle funzioni e' contenuta in un file **DBGpthread.c**

I prototipi delle poche funzioni sono contenuti nel file **DBGpthread.h**

```
void DBGpthread_mutex_lock( pthread_mutex_t *AddrMutex, const char *stringMsg );
void DBGpthread_mutex_unlock( pthread_mutex_t *AddrMutex, const char *stringMsg );
void DBGpthread_cond_wait( pthread_cond_t *AddrCond,
                           pthread_mutex_t *AddrMutex, const char *stringMsg );
void DBGpthread_cond_signal( pthread_cond_t *AddrCond, const char *stringMsg );
void DBGpthread_cond_broadcast( pthread_cond_t *AddrCond, const char *stringMsg );
void DBGpthread_mutex_init ( pthread_mutex_t *mutex,
                            const pthread_mutexattr_t *attr, const char *stringMsg );
void DBGpthread_mutex_destroy (pthread_mutex_t *mutex,const char *stringMsg);
void DBGpthread_cond_init ( pthread_cond_t *cond,
                            const pthread_condattr_t *attr, const char *stringMsg );
void DBGpthread_cond_destroy (pthread_cond_t *cond,const char *stringMsg);

void DBGsleep( unsigned int NumSeconds, const char *stringMsg );
```

Controllo d'Errore e Debug (4)

Nel file DBGpthread.c e' presente, **commentata**, la define di un simbolo **DEBUG**.

Decomentando quel simbolo, si avra' che ad ogni chiamata della funzione **DBG**funzione verra' stampato a video il nome della funzione "funzione", per vedere cosa sta accadendo.

L'implementazione delle funzioni DBG usa delle macro definite dentro il file **printerror.h** per riconoscere gli errori ed associarli ad una stringa di testo che descrive gli errori. La descrizione dell'errore viene poi stampata a video sullo stdout.

Una delle macro usate per stampare l'errore causato dalle funzioni dei pthread e'

PrintERROR_andExit()

Controllo d'Errore e Debug (5)

Estratto dal file **printerror.h**

```
#ifndef __PRINTERROR_H__           ← impedisce
#define __PRINTERROR_H__           inclusioni
                                → multiple
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>      /* per strerror_r and memset */

#define myMSGLEN 128

#define PrintERROR_andExit( ERRORCODE, STRMSG )
    do {
        char errmsg[myMSGLEN];
        memset( errmsg, 0, (size_t)myMSGLEN );
        strerror_r( ERRORCODE, errmsg, (size_t)myMSGLEN );
        printf("%s: errorcode %d %s - Quit\n",
               STRMSG, ERRORCODE, errmsg );
        exit(1);
    } while(0)

#endif /* __PRINTERROR_H__ */
```

fornisce stringa che descrive il tipo di errore indicato dal numero **ERRORCODE**

Controllo d'Errore e Debug (6)

Nota per l'utilizzo

delle macro contenute in **printerror.h**
e del modulo **DBGpthread.c**

per compilare correttamente i moduli in cui viene incluso il file **printerror.h**
occorre definire, nel makefile oppure nel codice sorgente,
il simbolo

_POSIX_C_SOURCE 200112L

poiché il file printerror.h contiene delle invocazioni
alla funzione strerror_r che richiede queste define.

Problemi Classici di programmazione concorrente risolti usando i Posix Thread

Directory degli esempi (1)

./PTHREAD/

Esempi di uso generico di thread, mutex e condition variable

./PTHREAD/CON_TRUCCO

./PTHREAD/BANALEGIUSTO

./PTHREAD/JOIN_DETACHED

./PTHREAD/MUTEX

./PTHREAD/CONDVAR

Esempi di problemi classici di programmazione concorrente

./PTHREAD/SINCRO_CIRCOLARE_1a1 contiene printerror.h DBGpthread.h DBGpthread.c

./PTHREAD/ALGORITMO_FORNAIO_PER_CODA_FIFO (1)

./PTHREAD/NPROD_MCONS contiene printerror.h DBGpthread.h DBGpthread.c

./PTHREAD/NLett_MScritt (1)

./PTHREAD/5FILOSOFI (1)

./PTHREAD/PASSAGGIO_TESTIMONE (1)

./PTHREAD/BARBIERE (1)

(1) I Makefile sfruttano printerror.h e DBGpthread.* nella directory ./PTHREAD/NPROD_MCONS

Sincronizzazione Circolare 1 a 1 (1)

Descrizione del problema applicato ai pthread:

Un programma crea due thread T1 e T2. Ciascuno dei due thread itera all'infinito eseguendo una propria sequenza di operazioni Op1 ed Op2 in mutua esclusione poiché usano dati condivisi.

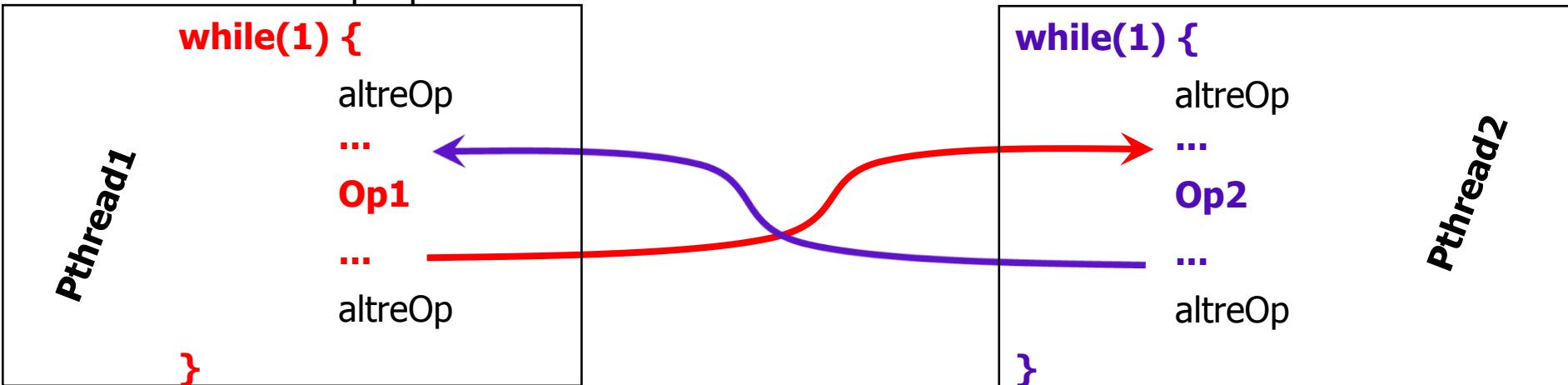
All'inizio e al termine del loop ciascun thread deve eseguire altre operazioni (in colore nero) non in mutua esclusione, quindi anche in parallelo quelle di T1 con quelle di T2.

La sincronizzazione tra i thread deve ottenere che le operazioni dei due thread eseguano in modo alternato, prima tutta Op1, poi tutta Op2, poi ancora tutta Op1 e poi tutta Op2 e così all'infinito. Chiariamo il concetto:

La prima iterazione deve essere svolta dal thread T1.

Al termine di ciascuna propria iterazione il thread T1 passa il controllo al thread T2 e si mette in attesa che il thread T2 gli restituisca il controllo. Quando ottiene il controllo, il thread T1 esegue una altra iterazione e poi passa nuovamente il controllo al thread T2.

Al termine di ciascuna propria iterazione il thread T2 passa il controllo al thread T1 e si mette in attesa che il thread T1 gli restituisca il controllo. Quando ottiene il controllo, il thread T2 esegue una altra iterazione e poi passa nuovamente il controllo al thread T1.



Sincronizzazione Circolare 1 a 1 (2)

Basterebbe che i due thread usassero una mutex per rispettare turno ?????

- `pthread_mutex_t mutex;` per mutua esclusione

```
void *ThreadPrimo ( void *arg ) {  
    while(1) {  
        pthread_mutex_lock(&mutex);      /* Primo aspetta */  
        /* esegue le operazioni Op1 */  
        /* rilascio mutua esclusione, cosi' Secondo parte ????? */  
        pthread_mutex_unlock( &mutex );  
    }  
}  
  
void *ThreadSecondo ( void *arg ) {  
    while(1) {  
        pthread_mutex_lock(&mutex);      /* Secondo aspetta */  
        /* esegue le operazioni Op2 */  
        /* rilascio mutua esclusione, cosi' Primo parte ????? */  
        pthread_mutex_unlock( &mutex );  
    }  
}
```

Sincronizzazione Circolare 1 a 1 (3)

riferendoci alla precedente slide

Basterebbe che i due thread usassero una mutex per rispettare turno ?????

NO !

- se ThreadSecondo esegue la sua mutex_unlock() e viene interrotto dallo scheduler nella istruzione while(1), prima che possa eseguire l'istruzione pthread_mutex_lock()
 - allora ThreadPrimo puo' uscire dalla sua mutex_lock(), eseguire la sua sezione critica, eseguire la mutex_unlock, eseguire ancora la sua mutex_lock() e uscirne subito per eseguire per la seconda volta consecutiva la sua sezione critica.
Quindi **Violazione della Sincronizzazione voluta.**

Sincronizzazione Circolare 1 a 1 (4)

Basterebbe che i thread usassero mutex e cond per fare prima una wait e poi una signal ?????

- pthread_mutex_t mutex; pthread_cond_t cond ;

```
void *ThreadPrimo ( void *arg ) {
    while(1) {
        pthread_mutex_lock(&mutex);
        pthread_cond_wait( &cond, &mutex); /* Primo aspetta */
        ..... /* esegue le operazioni Op1 */
        pthread_cond_signal( &cond );      /* abilito Secondo???? */
        pthread_mutex_unlock( &mutex );   /* cosi' Secondo riparte ??? */
    }
}

void *ThreadSecondo ( void *arg ) {
    while(1) {
        pthread_mutex_lock(&mutex);
        pthread_cond_wait( &cond, &mutex); /* Primo aspetta */
        ..... /* esegue le operazioni Op2 */
        pthread_cond_signal( &cond );      /* abilito Primo???? */
        pthread_mutex_unlock( &mutex );   /* cosi' Primo riparte ??? */
    }
}
```

Sincronizzazione Circolare 1 a 1 (5)

riferendoci alla precedente slide

Basterebbe che i thread usassero mutex e cond per fare prima una wait e poi una signal ?????

NO !

- nel caso presentato nella slide precedente si blocca tutto subito !!!
 - ad esempio, ThreadPrimo esegue la mutex_lock() e poi si blocca sulla sua cond_wait() rilasciando la mutua esclusione.
 - Poi ThreadSecondo esegue la mutex_lock() e poi si blocca sulla sua cond_wait() rilasciando la mutua esclusione.
 - A questo punto, entrambi i thread sono bloccati nelle loro cond_wait() e nessuno riesce più a proseguire.
- Quindi Blocco di tutti i due Thread (Deadlock).

Sincronizzazione Circolare 1 a 1 (8)

Tentativo di mossa del cavallo: "mettere la lock prima del while?"

ThreadPrimo e ThreadSecondo sono uguali, tranne che eseguono o Op1 oppure Op2.

```
void *Thread1/2 ( void *arg ) {  
    pthread_mutex_lock(&mutex);  
    while(1) {  
        .... altreOp  
        pthread_cond_signal( &cond );      /* abilito l'altro */  
        ..... Op1/2      /* esegue le operazioni Op1/Op2 */  
        pthread_cond_wait( &cond, &mutex); /* aspetto */  
        .... altreOp  
    }  
    pthread_mutex_unlock( &mutex );    /* non viene mai fatta */  
}
```

Bel tentativo, **ma presenta 2 problemi**:

1. Come nel caso presentato nella slide precedente, chi impedisce che venga eseguita Op2 per prima? Nessuno, può capitare.
2. **Anche le *altreOp* sono eseguite in mutua esclusione, anche se potrebbero essere eseguite "in parallelo".** Se le altreOp sono lente o attendono input (ricevono da rete), fino a che queste non sono terminate non posso eseguire le Op1 o Op2. In pratica, metto in serie le operazioni altreOp mentre potrei eseguirle in parallelo.

Corretta - Sincronizzazione Circolare 1 a 1 (6)

Variabili Globali per la Sincronizzazione

- `pthread_mutex_t mutex;` per mutua esclusione
- `pthread_cond_t cond;` per wait and signal
- `int PrimoFaiWait, /* Messo a 0 da Secondo quando fa la signal, per evitare che Primo faccia la wait DOPO la signal */`
- `int SecondoFaiWait;`

```
uint64_t valGlobale=0; /* dati da proteggere in sezione critica */
```

```
int main ()  
{    pthread_t th;  
    pthread_cond_init(&cond, NULL);  
    pthread_mutex_init(&mutex, NULL);
```

PrimoFaiWait=0; /* all'inizio Primo **non deve aspettare Secondo** */
SecondoFaiWait=1;

```
pthread_create( &th, NULL, Primo, NULL);  
pthread_create( &th, NULL, Secondo, NULL);  
pthread_exit( NULL );
```

```
}
```

Corretta - Sincronizzazione Circolare 1 a 1 (7)

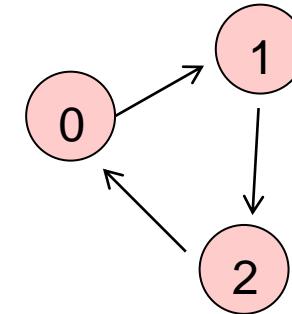
```
void *Primo ( void * arg ) {  
    while(1) {  
        pthread_mutex_lock(&mutex);  
        /* se Secondo ha gia' fatto la signal allora Primo deve proseguire senza fare la wait */  
        if ( PrimoFaiWait ) pthread_cond_wait(&cond,&mutex);  
        /* al prossimo giro Primo devo fare la wait,  
         a meno che Secondo non faccia la signal prima che Primo tenti di fare la wait */  
        PrimoFaiWait=1;  
  
        /* SEZIONE CRITICA : legge cio' che e' stato messo da Secondo e lo cambia */  
        valGlobale=valGlobale+3;  
  
        pthread_cond_signal(&cond);      /* risveglio Secondo */  
        /* Nel caso che Secondo non abbia ancora fatto la wait allora Primo dice a Secondo  
         che non deve aspettare perche' Primo ha gia' fatto la signal */  
        SecondoFaiWait=0;  
        pthread_mutex_unlock(&mutex); /* rilascio mutua esclusione, cosi' Secondo parte */  
    }  
}
```

Corretta - Sincronizzazione Circolare 1 a 1 (8)

```
void *Secondo ( void *arg ) {  
    while(1) {  
        pthread_mutex_lock(&mutex);  
        /* se Primo ha gia' fatto la signal allora Secondo deve proseguire senza fare la wait */  
        if ( SecondoFaiWait ) pthread_cond_wait(&cond,&mutex);  
        /* al prossimo giro Secondo devo fare la wait,  
         a meno che Primo non faccia la signal prima che Secondo tenti di fare la wait */  
        SecondoFaiWait=1;  
  
        /* SEZIONE CRITICA : legge cio' che e' stato messo da Primo e lo cambia */  
        valGlobale=valGlobale+10;  
  
        pthread_cond_signal(&cond);      /* risveglio Primo */  
        /* Nel caso che Primo non abbia ancora fatto la wait allora Secondo dice a Primo  
         che non deve aspettare perche' Secondo ha gia' fatto la signal */  
        PrimoFaiWait=0;  
        pthread_mutex_unlock(&mutex); /* rilascio mutua esclusione, cosi Primo puo partire */  
    }  
}
```

Sincr. Circolare a 3 con turno

```
int turno=0; int var_globale=33; pthread_mutex_t mutex; pthread_cond_t cond;  
void * thread_function (void *arg)  
{  
    intptr_t mioindice = (intptr_t)arg;  
    while(1)  
    {  
        pthread_mutex_lock( &mutex );  
        while( turno != mioindice )  
        {  
            pthread_cond_wait( &cond, &mutex );  
            if( turno != mioindice )  
                pthread_cond_signal(&cond);  
        }  
        valGlobale=valGlobale+3; /* sezione critica */  
        turno= (turno+1) %3;  
        pthread_cond_signal( &cond );  
        pthread_mutex_unlock( &mutex ); /*rilascio mutex */  
    }  
}  
int main () { pthread_t th; intptr_t i;  
    pthread_cond_init(&cond,NULL); pthread_mutex_init(&mutex,NULL);  
    for( i=0; i<3; i++ ) pthread_create( &th, NULL, thread_function, (void*) i  
);  
    pthread_exit( NULL );
```



Algoritmo del fornaio per code FIFO

Descrizione del problema applicato ai pthread:

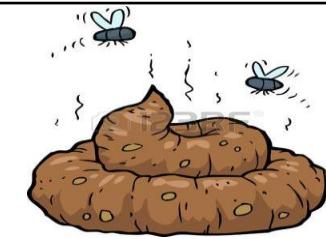
- Ad un fornaio non piace stare al bancone a servire i clienti, perciò i clienti si servono da soli, ma **uno alla volta, nell'ordine in cui prendono il biglietto numerato che stabilisce il loro turno.** Un display visualizza il numero del biglietto che può accedere al bancone.
- Prima di accedere al bancone del pane, perciò, ciascun cliente prende il biglietto contenente un numero che stabilisce il suo turno ed aspetta il suo turno.
- Il biglietto viene preso in mutua esclusione mediante una variabile apposita mutex mutexDistributoreBiglietti, che protegge una variabile bigliettoGlob che mantiene il valore del prossimo biglietto da consegnare al prossimo cliente.
- Il primo biglietto distribuito vale 0, i successivi vengono incrementati di 1 alla volta.
- La variabile globale condivisa bigliettoSulDisplay, indica ai clienti quale e' il numero del biglietto del prossimo cliente che può andare al bancone a servirsi.
- La variabile globale bigliettoSulDisplay viene settata dal cliente che ha appena finito di servirsi al bancone. Quando questo cliente va via dal bancone, incrementa di 1 il valore di bigliettoSulDisplay .
- Una variabile mutex mutexTurno protegge l'accesso alla variabile globale bigliettoSulDisplay.
- **Per evitare Busy Waiting occorrerà aggiungere una condition variable per far attendere i clienti non di turno.**

Ciascun cliente impiega 1 secondo a servirsi al bancone.

Implementazione Non Efficiente Algoritmo del fornaio

```
int bigiettoGlob=0, bigiettoSulDisplay=0;  
pthread_mutex_t mutexDistributoreBiglietti , mutexTurno;  
  
void *cliente (void *arg) { int bigietto;  
    while(1) {  
        /* cliente prende il bigietto */  
        pthread_mutex_lock( &mutexDistributoreBiglietti );  
        bigietto=bigiettoGlob;  
        bigiettoGlob++;  
        pthread_mutex_unlock( &mutexDistributoreBiglietti );  
  
        /* cliente aspetta il suo turno */  
        do {  
            pthread_mutex_lock( &mutexTurno );  
            if ( bigiettoSulDisplay == bigietto )  
                break;  
            pthread_mutex_unlock( &mutexTurno );  
        } while(1);  
        /* il cliente si serve al bancone */  
  
        bigiettoSulDisplay++; /*cliente ha finito, il prossimo é ... */  
        pthread_mutex_unlock( &mutexTurno );  
    }  
}
```

Soluzione NON CORRETTA.
Causa BUSY WAITING



il cliente non di turno,
anche se capisce di
"non essere di turno",
non si ferma
e può acquisire
di nuovo e più volte
consecutivamente
la mutua esclusione,
rubando tempo agli
altri.

Implementazione efficiente Algoritmo del fornaio

```
int bigiettoGlob=0, bigiettoSulDisplay=0;
pthread_mutex_t mutexDistributoreBiglietti , mutexTurno; pthread_cond_t cond;
void *cliente (void *arg) {           int bigietto;
    while(1) {
        /* cliente prende il bigietto */
        pthread_mutex_lock( &mutexDistributoreBiglietti );
        bigietto=bigiettoGlob;
        bigiettoGlob++;
        pthread_mutex_unlock( &mutexDistributoreBiglietti );
        /* cliente aspetta il suo turno */
        pthread_mutex_lock( &mutexTurno );
        while ( bigiettoSulDisplay != bigietto )
            pthread_cond_wait( &cond, &mutexTurno );
    }
    /* il cliente si serve al bancone */

    bigiettoSulDisplay++; /*cliente ha finito, il prossimo é ... */
    pthread_cond_broadcast( &cond );
    pthread_mutex_unlock( &mutexTurno );
}
}
```

il cliente non di turno
appena capisce di
"non essere di turno"
si ferma
fino a che
il cliente di turno
riesce a finire
il suo lavoro.

**fare test sperimentale in aula per
confrontare il numero di clienti
serviti in un minuto:
1926 contro 667531.**

N Produttori e M Consumatori con K Buffer

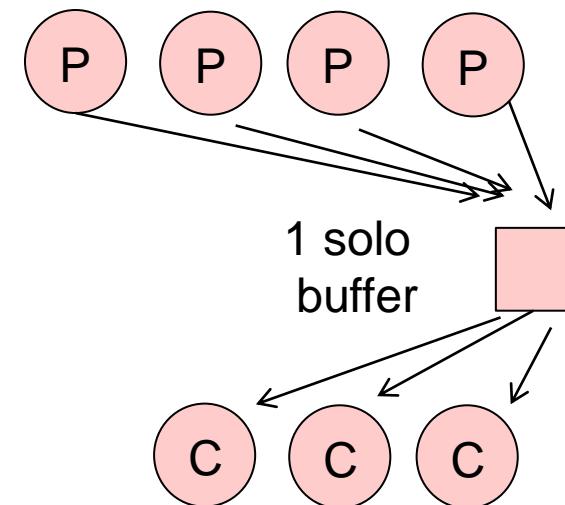
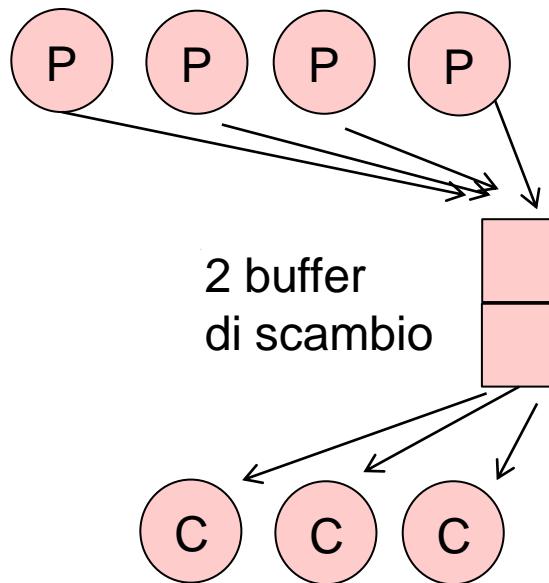
Descrizione del problema applicato ai pthread:

Un programma e' composto da due tipi di thread, i Produttori ed i Consumatori.

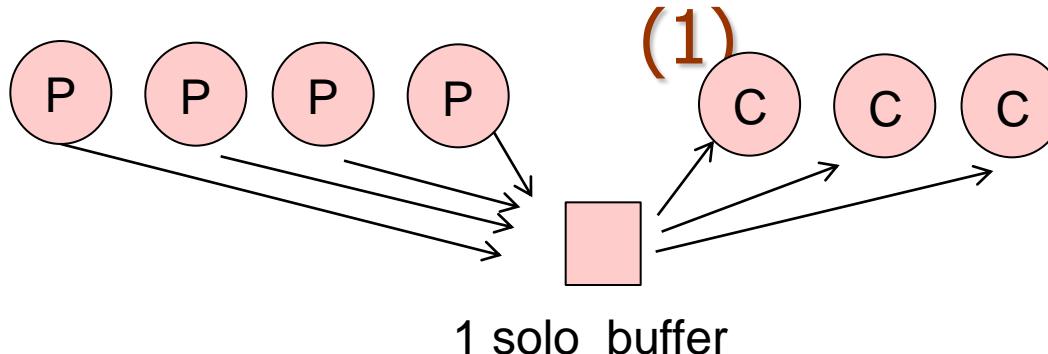
Entrambi i tipi di thread eseguono un loop infinito, durante il quale i Produttori costruiscono un qualche tipo di dato mentre i Consumatori utilizzano i dati prodotti.

Possono esistere uno o piu' thread Produttori ed uno o piu' thread Consumatori.

Lo scambio dei dati prodotti avviene per tramite di uno o piu' buffer, ciascuno dei quali capace di contenere uno dei dati prodotti.



N Produttori e M Consumatori con 1 solo Buffer

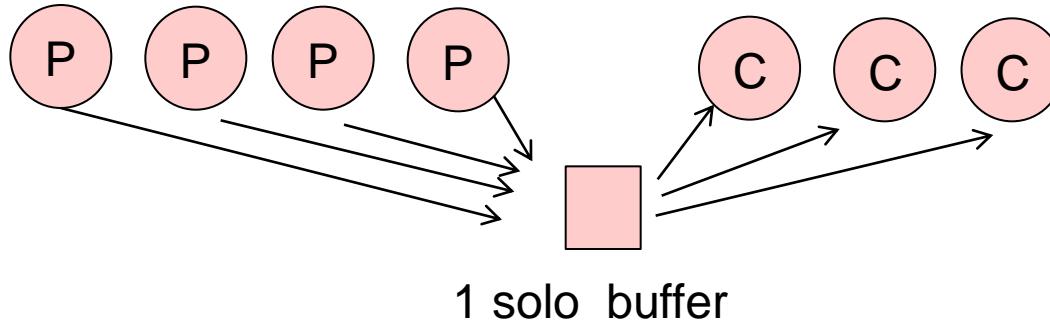


Descrizione del problema applicato ai pthread:

Nel caso in cui ci sia un unico buffer di scambio, i vincoli per la sincronizzazione di Produttori e Consumatori sono i seguenti:

- Il buffer puo' essere acceduto solo da un thread alla volta.
- Ciascun Produttore deve aspettare che il buffer si svuoti prima di depositare il proprio dato nel buffer. Ovviamente, solo un Produttore per volta puo' depositare nel buffer il proprio dato.
- Lo svuotamento del buffer di scambio avviene ad opera di un Consumatore. Solo un Consumatore per volta puo' estrarre il dato dal buffer.
- Tutti i Produttori che hanno prodotto un dato devono, prima o poi, riuscire a depositare il loro dato nel buffer.
- Tutti i Consumatori che possono consumare un dato devono, prima o poi, riuscire a estrarre un dato dal buffer di scambio.

N Produttori e M Consumatori con 1 solo Buffer (2)



La sezione Critica del problema Produttori Consumatori.

Per i Produttori, la sezione critica e' la parte di codice in cui i Produttori depositano il dato da loro prodotto nel buffer di scambio. Per i Consumatori, la sezione critica e' la parte in cui i Consumatori prelevano un dato dal buffer di scambio. Queste sezioni critiche potrebbero avere una durata qualunque.

La parte in cui ciascun Produttore produce il proprio dato non appartiene alla sezione critica. Analogamente, la parte in cui ciascun Consumatore consuma il dato prelevato non appartiene alla sezione critica. Queste parti fuori dalla sezione critica potrebbero avere una durata qualunque.

Si noti che, essendo fuori dalla sezione critica, le fasi di produzione dei dati (effettuate dai Produttori) e le fasi di consumo dei dati (effettuate dai Consumatori) possono eseguire in contemporanea.

N Produttori e M Consumatori con 1 solo Buffer (3)

Qui di seguito, descriviamo una soluzione non efficientissima ma molto più semplice da implementare. Questa prima soluzione comunque **NON FA BUSY WAITING** e perciò è ragionevole utilizzarla soprattutto se non ci sono molti thread.

Subito dopo descriveremo invece una soluzione decisamente più efficiente ma difficile da implementare, che quindi potrebbe non essere consigliabile. L'efficienza di questa seconda soluzione aumenta se ci sono molti thread da sincronizzare.

N Produttori e M Consumatori con 1 solo Buffer (4)

Versione Meno Efficiente ma facile da realizzare

Variabili Globali per la Sincronizzazione (trattiamo il caso generale con K buffer, l'implementazione usa un solo buffer)

- `pthread_mutex_t mutex;` per mutua esclusione
- `pthread_cond_t condProd , condCons;`
DUE cond variable, per far attendere e risvegliare DUE diversi insiemi di pthread, i Prod e i Cons
- `int numBufferPieni=0;`
il numero (0 o 1) di buffer con dentro il dato pronto

N Produttori e M Consumatori con 1 solo Buffer (5)

Versione Meno Efficiente ma facile da realizzare

```
#define NUMPROD 5
#define NUMCONS 3
#define NUMBUFFER 1          /* 1 solo buffer di scambio */
uint64_t valGlobale=0;      /* dati da proteggere in sezione critica */

int main ()
{
    pthread_t th;      int rc;  intptr_t i;
    pthread_cond_init(&condProd, NULL);
    pthread_cond_init(&condCons, NULL);
    pthread_mutex_init(&mutex, NULL);

    numBufferPieni=0;      /* 0 o 1 , ad inizio buffer vuoto */

    for( i=0; i< NUMPROD ; i++ )
        rc = pthread_create( &th, NULL, Prod, (void*)i );
    for( i=0; i< NUMCONS ; i++ )
        rc = pthread_create( &th, NULL, Cons, (void*)i );
    pthread_exit( NULL );
}
```

N Produttori e M Consumatori con 1 solo Buffer (6)

Versione Meno Efficiente ma facile da realizzare

```
void * Prod ( void * arg ) {  
    while(1) { /* qui il Prod PRODUCE IL DATO, un intero. */  
        pthread_mutex_lock( &mutex);  
        while ( numBufferPieni >= NUMBUFFER )  
            pthread_cond_wait ( &condProd, & mutex);  
        valGlobale = valProdotto; /* SEZ CRITICA : riempie il buffer col dato prodotto */  
        numBufferPieni++;  
        pthread_cond_signal ( &condCons ); /* risveglio 1 Cons per svuotare 1 buffer */  
        pthread_mutex_unlock ( &mutex ); /* rilascio mutua esclusione */  
    }  
  
    void * Cons ( void * arg ) {  
        while(1) {  
            pthread_mutex_lock( &mutex);  
            while ( numBufferPieni <= 0 )  
                pthread_cond_wait ( &condCons, & mutex);  
            val = valGlobale; /* SEZ CRITICA : prendo cio; che sta nel buffer di scambio */  
            numBufferPieni--;  
            pthread_cond_signal ( &condProd ); /* risveglio 1 Prod per riempire il buffer */  
            pthread_mutex_unlock ( &mutex ); /* rilascio mutua esclusione */  
        }  
    }
```

N Produttori e M Consumatori con 1 solo Buffer (7) versione efficiente

Variabili Globali per la Sincronizzazione (trattiamo il caso generale con K buffer, l'implementazione usa un solo buffer)

- `pthread_mutex_t mutex;` per mutua esclusione
- `pthread_cond_t condProd , condCons;`

DUE cond variable, per far attendere e risvegliare DUE diversi insiemi di pthread, i Prod e i Cons

- `int numBufferPieni=0;`
il numero (0 o 1) di buffer con dentro il dato pronto
- `int numProdWaiting=0;`
il numero di Prod in attesa sulla `cond_wait`
- `int numProdWaitingAndSignalled=0;`
il numero di Prod in attesa sulla `cond_wait` e gia' abilitati da una signal.
Tale valore deve sempre essere `<= #buffer vuoti` altrimenti `!@#!.....`
- `int numConsWaiting=0;`
il numero di Cons in attesa sulla `cond_wait`
- `int numConsWaitingAndSignalled=0;`
il numero di Cons in attesa sulla `cond_wait` e gia' abilitati da una signal.
Tale valore deve sempre essere `<= #buffer pieni` altrimenti `!@#!`

N Produttori e M Consumatori con 1 solo Buffer (8) versione efficiente

```
#define NUMPROD 5
#define NUMCONS 3
#define NUMBUFFER 1      /* 1 solo buffer di scambio */
uint64_t valGlobale=0;    /* dati da proteggere in sezione critica */

int main ()
{
    pthread_t th;    int rc;  intptr_t i;
    pthread_cond_init(&condProd, NULL);
    pthread_cond_init(&condCons, NULL);
    pthread_mutex_init(&mutex, NULL);

numBufferPieni=0;      /* 0 o 1 , ad inizio buffer vuoto */
numProdWaiting=0;
numProdWaitingAndSignalled=0;
numConsWaiting=0;
numConsWaitingAndSignalled=0;

    for( i=0; i< NUMPROD ; i++ )
        rc = pthread_create( &th, NULL, Prod, (void*)i );
    for( i=0; i< NUMCONS ; i++ )
        rc = pthread_create( &th, NULL, Cons, (void*)i );
    pthread_exit( NULL );
}
```

N Produttori e M Consumatori con 1 solo Buffer (9)

versione efficiente

```
void * Prod ( void * arg ) {  
    while(1) { /* qui il Prod PRODUCE IL DATO, un intero. */  
        pthread_mutex_lock( &mutex);  
        if ( numProdWaitingAndSignalled >= (NUMBUFFER-numBufferPieni) ) {  
            numProdWaiting++;  
            pthread_cond_wait ( &condProd, &mutex);  
            numProdWaiting--;  
            numProdWaitingAndSignalled--;  
        }  
        valGlobale = valProdotto; /* SEZ CRITICA : riempie il buffer col dato prodotto */  
        numBufferPieni++;  
        if ( ( numConsWaitingAndSignalled < numConsWaiting ) &&  
            ( numConsWaitingAndSignalled < numBufferPieni ) ) {  
            pthread_cond_signal ( &condCons ); /*risveglio 1 Cons per svuotare 1 buffer*/  
            numConsWaitingAndSignalled++;  
        }  
        pthread_mutex_unlock ( &mutex ); /* rilascio mutua esclusione */  
    }  
}
```

N Produttori e M Consumatori con 1 solo Buffer (10)

versione efficiente

```
void * Cons( void * arg ) {  
    while(1) {  
        pthread_mutex_lock( &mutex );  
        if ( numConsWaitingAndSignalled  >= numBufferPieni  ) {  
            numConsWaiting++;  
            pthread_cond_wait ( &condCons, & mutex );  
            numConsWaiting--;  
            numConsWaitingAndSignalled--;  
        }  
        val = valGlobale; /* SEZ CRITICA : prendo cio; che sta nel buffer di scambio */  
        numBufferPieni--;  
        if ( ( numProdWaitingAndSignalled < numProdWaiting )  &&  
            ( numProdWaitingAndSignalled < (NUMBUFFER-numBufferPieni ) )  
        ) {  
            pthread_cond_signal ( &condProd ); /*risveglio 1 Prod per riempire buffer */  
            numProdWaitingAndSignalled++;  
        }  
        pthread_mutex_unlock ( &mutex ); /* rilascio mutua esclusione */  
    }  
}
```

N Lettori e M Scrittori su un Buffer (1)

Descrizione del problema applicato ai pthread:

Un programma e' composto da due tipi di thread, i Lettori e gli Scrittori.

Tutti i thread accedono ad **uno** stesso buffer comune.

Entrambi i tipi di thread eseguono un loop infinito, durante il quale i Lettori accedono al buffer comune leggendone il contenuto, mentre gli Scrittori accedono al buffer comune in scrittura cioe' modificandone il contenuto.

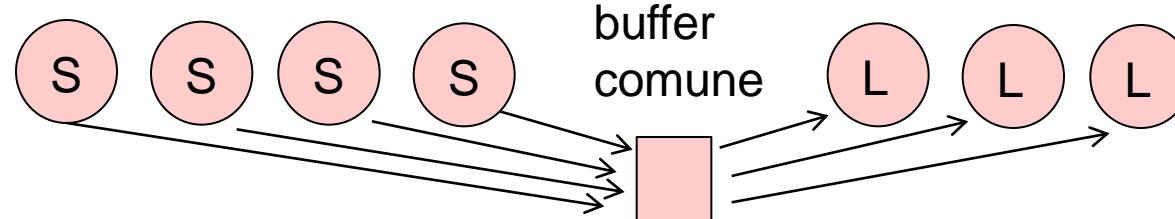
Possono esistere uno o piu' thread Lettori ed uno o piu' thread Scrittori.

Vincoli del problema:

Quando uno Scrittore accede al buffer, poichè ne sta modificando il contenuto, nessun altro thread puo' accedere al buffer.

Quando un Lettore accede al buffer, poichè non ne modifica il contenuto, anche altri Lettori possono accedere in contemporanea al buffer.

Quindi, se nessuno Scrittore accede al buffer, molti Lettori possono accedere assieme.



N Lettori e M Scrittori su un Buffer (2)

Soluzione semplice ma sbagliata: mutua esclusione sul buffer

Una soluzione del problema Readers&Writes potrebbe essere quella di proteggere in mutua esclusione il buffer comune, permettendo l'accesso **ad un solo** thread per volta, indipendentemente dal fatto che questo thread sia un Lettore o uno Scrittore.

In questo modo, però, il sistema non consente a più Lettori di accedere al buffer in contemporanea, mentre ciò sarebbe possibile senza problemi, poiché i Lettori non modificano il contenuto del buffer. La soluzione è assai poco efficiente.

D'altro canto, questa soluzione non fa preferenze tra Lettori e Scrittori, quindi prima o poi tutti accederanno al buffer.

Soluzione corretta: traccia

Una soluzione corretta consente che, fino a che c'e' un Lettore già in lettura sul buffer, un altro lettore possa accedere al buffer. Questo permette a più lettori di accedere al buffer contemporaneamente, con una sorta di passaggio di testimone.

Limiti della Soluzione : Starvation (morte di fame) degli Scrittori

In questo modo, però, gli Scrittori rischiano di non poter mai accedere al buffer se c'e' una sequenza ininterrotta di accessi al buffer da parte dei Lettori.

Purtroppo è un limite del problema, che è intrinsecamente asimmetrico. Si potrebbe pensare di favorire gli Scrittori, ma si rischierebbe la starvation dei Lettori.

N Lettori e M Scrittori su un Buffer (3)

Variabili Globali

- `uint64_t valGlobale =0; /* Buffer condiviso da proteggere */`

Variabili Globali per la Sincronizzazione

- `pthread_mutex_t mutexBuffer;` mutua esclusione sul Buffer
- `pthread_mutex_t mutexVarLettori;` mutua esclusione su var che conta lettori in lettura
- `int numLettoriInLettura=0;` numero lettori che stanno leggendo, all'inizio è ZERO

```
#define NUMLETTORI 5
#define NUMSCRITTORI 3
int main ()
{
    pthread_t th;    int rc;  intptr_t i;

    pthread_mutex_init( &mutexBuffer, NULL);
    pthread_mutex_init( &mutexVarLettori, NULL);
    numLettoriInLettura=0
    for( i=0; i< NUMLETTORI ; i++ )
        rc = pthread_create( &th, NULL, Lettore, (void*)i );
    for( i=0; i< NUMSCRITTORI ; i++ )
        rc = pthread_create( &th, NULL, Scrittore, (void*)i );
    pthread_exit( NULL );
}
```

N Lettori e M Scrittori su un Buffer (4)

```
void *Scrittore (void *arg) {      uint64_t  valProdotto;
    while(1) {
        /* sezione NON critica
           QUI LO SCRITTORE PRODUCE IL DATO */
        valProdotto=qualcosa.....
        /* prendo la mutua esclusione sul buffer*/
        pthread_mutex_lock( &mutexBuffer);
        /* INIZIO SEZIONE CRITICA */
        valGlobale=valProdotto;      /* lo scrittore scrive nel buffer il proprio dato */
        /* FINE SEZIONE CRITICA */
        /* rilascio mutua esclusione sul buffer*/
        pthread_mutex_unlock(&mutexBuffer );
    }
}
```

N Lettori e M Scrittori su un Buffer (5)

```
void *Lettore (void *arg) {      uint64_t val; /* var locale in cui salvo valore letto */
    while(1) {
        /* prendo la mutua esclusione sulla variabile dei lettori*/
        pthread_mutex_lock( &mutexVarLettori);
        if ( numLettoriInLettura <= 0 )
            pthread_mutex_lock(&mutexBuffer);
        numLettoriInLettura++;
        pthread_mutex_unlock ( &mutexVarLettori );

        /* SEZIONE CRITICA: leggo cio' che c'e' nel buffer */
        val=valGlobale;
        /* FINE SEZIONE CRITICA */

        /* riprendo la mutua esclusione sulla variabile dei lettori */
        pthread_mutex_lock(&mutexVarLettori);
        numLettoriInLettura--; /* io non leggo piu' */
        /* se sono ultimo lettore in lettura, rilascio mutua esclus. per gruppo lettori */
        if ( numLettoriInLettura <= 0 )
            /* rilascio mutua esclusione sul buffer */
            pthread_mutex_unlock(&mutexBuffer);
        /* rilascio la mutua esclusione sulla variabile dei lettori */
        pthread_mutex_unlock(&mutexVarLettori);
    }
}
```

5 Filosofi (1)

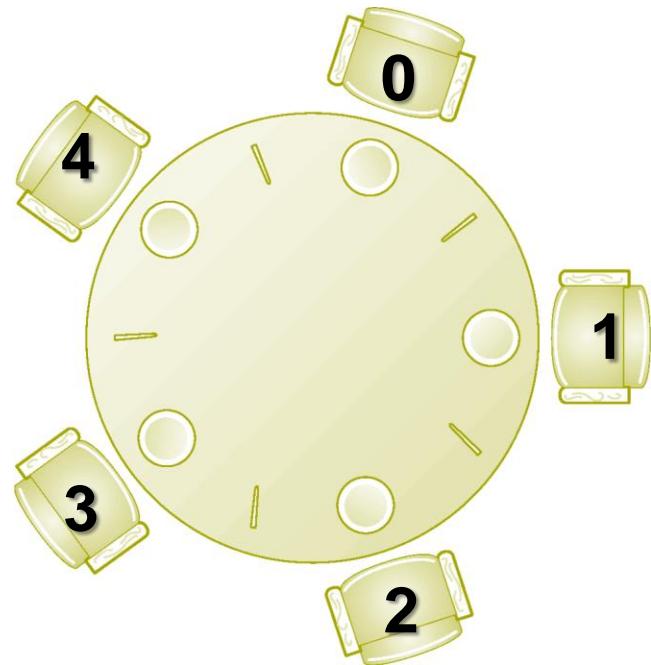
Descrizione del problema applicato ai pthread:

5 filosofi sono a tavola per mangiare.

Per mangiare occorrono due forchette.

Alla destra di ciascun filosofo c'e' una forchetta.

Per mangiare il filosofo deve prendere
sia la forchetta alla sua destra
e anche la forchetta alla sua sinistra.



Dopo aver mangiato il filosofo mette a posto le forchette e pensa per un po'.

Poi gli viene ancora fame e cerca di mangiare.

Evidentemente non potranno mangiare tutti contemporaneamente ma occorre assicurare il **massimo grado di parallelismo**. Cioè devo permettere che 2 filosofi (non adiacenti) mangino assieme utilizzando 4 delle 5 forchette.

Per semplicità, indicizzo i filosofi da 0 a 4, in senso orario cominciando da quello più in alto.

5 Filosofi (2)

Vincoli del problema:

- Evidentemente non potranno mangiare tutti contemporaneamente ma occorre assicurare il **massimo grado di parallelismo**. Cioè devo permettere che 2 filosofi (non adiacenti) mangino assieme utilizzando 4 delle 5 forchette.
- Contemporaneamente andrebbe evitato il **deadlock**: se ciascun filosofo prende la forchetta di sinistra e poi si mette in attesa di quella di destra, si bloccano tutti indefinitamente. Se si accorgono del blocco e tutti posano assieme la loro forchetta e poi le riprendono assieme allora potrebbero non mangiare mai nessuno (starvation).
- Come evitare deadlock:
 - **Prendere due forchette assieme, solo quando sono libere entrambe**
 - ▶ In generale, **prendere tutte le risorse che servono assieme, prima di cominciare ad usarle**. Non è detto che sia sempre possibile.
 - ▶ Questa è la soluzione che vedremo nella prossima slide
 - **Prendere le forchette in ordine diverso** (tutti prendono per prima la forchetta alla destra, uno solo comincia prendendo la forchetta alla sinistra).
 - ▶ In generale, modificare l'ordine di accesso alle risorse per evitare blocchi incrociati.
 - ▶ Nel caso dei filosofi, non si garantisce il massimo parallelismo

5 Filosofi (3) - prendere due forchette assieme

```
#define NUMFILOSOFI 5
```

```
/* stati del filosofo */  
#define PENSA 0  
#define HAFAME 1  
#define MANGIA 2 /* Lui ... lui maaaaaaaaangiaa */
```



```
/* variabili da proteggere */  
int statoFilosofo[NUMFILOSOFI]; /* inizializzato tutti con PENSA */
```

```
/* variabili per la sincronizzazione */  
pthread_mutex_t mutex; /* permette l'accesso alle variabili di stato */  
pthread_cond_t condFilosofo[NUMFILOSOFI];
```

5 Filosofi (4) - prendere due forchette assieme

```
int indiceasinistra ( int indice) {  
    return( (indice+1)%NUMFILOSOFI );  
}  
  
int indiceadestra ( int indice) {  
    if( indice==0 ) return( NUMFILOSOFI-1 );  
    else      return( (indice-1)%NUMFILOSOFI );  
}  
  
/* funzione che deve essere eseguita in mutua esclusione */  
int puomangiare( int indice ) {  
    if (  
        statoFilosofo[indice]==HAFAME  
        &&  
        statoFilosofo[ indiceadestra(indice) ] != MANGIA  
        &&  
        statoFilosofo[ indiceasinistra(indice) ] != MANGIA  
    )  
        return( 1 );  
    else  
        return( 0 );  
}
```

5 Filosofi (5) - prendere due forchette assieme

```
void *filosofo (void *arg) { int indice = indice del filosofo, PASSATO DALL'ARGOMENTO
    while (1) { /* PENSA */
        pthread_mutex_lock(&mutex);
        /* sono affamato, i vicini mi risveglieranno quando finiranno di mangiare */
        statoFilosofo[indice]=HAFAME;
        if ( puomangiare( indice ) ) statoFilosofo[indice]=MANGIA;
        else pthread_cond_wait( &(condFilosofo[indice]), &mutex );
        pthread_mutex_unlock (&mutex );
        /* ora posso prendere le due forchette, so che sono libere */
        /* qui ora mangio ..... */
        /* rilascio le forchette */
        pthread_mutex_lock(&mutex);
        /* dico che ho smesso di mangiare e non mi servono piu' le forchette */
        statoFilosofo[indice]=PENSA;
        /* cerco di svegliare i vicini se hanno fame e se possono mangiare */
        if ( puomangiare( indiceadestra(indice) ) ) {
            pthread_cond_signal( &(condFilosofo[indiceadestra(indice)]) );
            statoFilosofo[ indiceadestra(indice) ]=MANGIA;
        }
        if ( puomangiare( indiceasinistra(indice) ) ) {
            pthread_cond_signal( &(condFilosofo[indiceasinistra(indice)]) );
            statoFilosofo[ indiceasinistra(indice) ]=MANGIA;
        }
        pthread_mutex_unlock(&mutex );
    }
}
```

Passaggio sicuro di testimone - three way handshake

passare il testimone lasciandolo solo quando è stato già preso dal successivo



Passaggio sicuro di testimone - three way handshake (1)

Descrizione del problema :

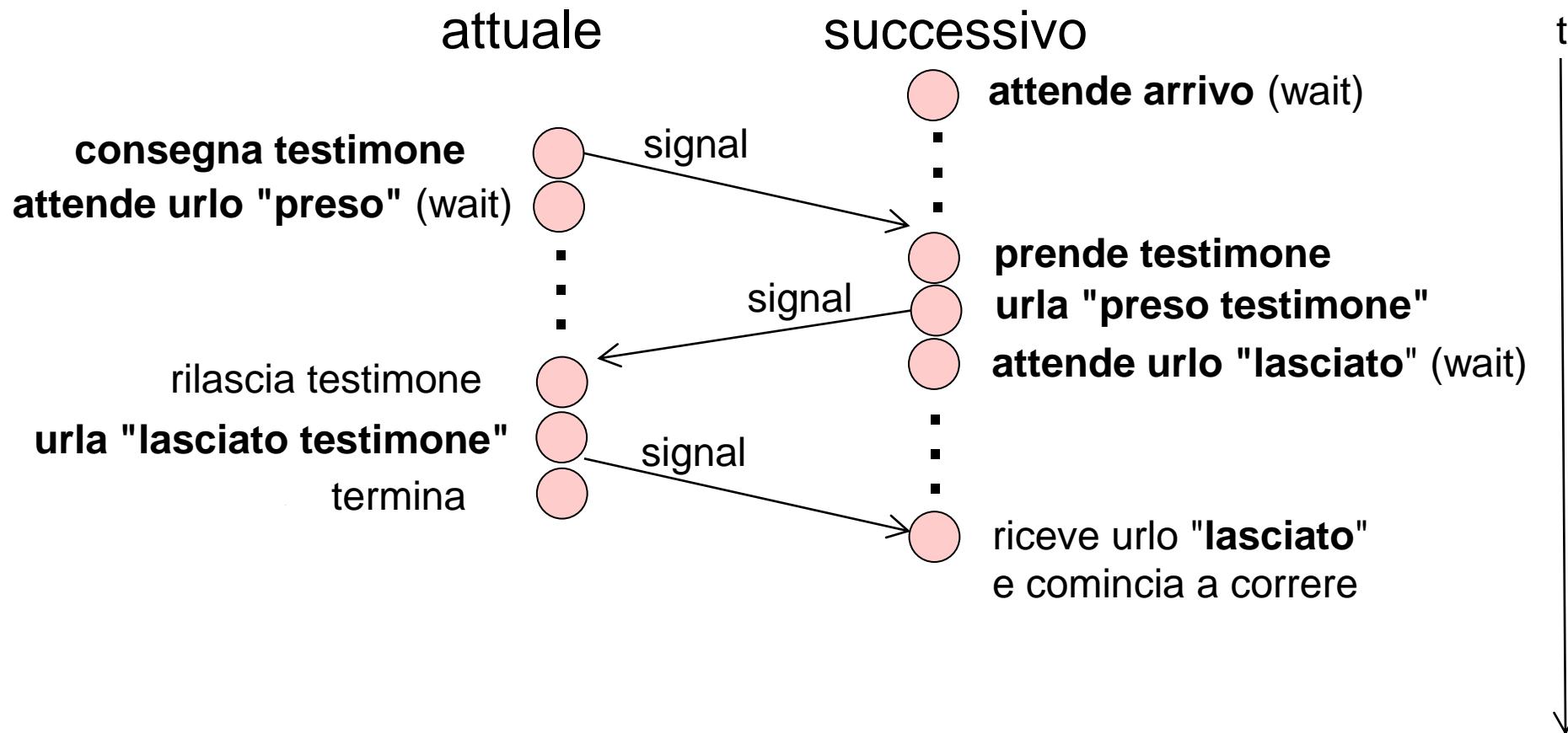
- Un gruppo di N atleti devono effettuare, uno alla volta, un giro di pista, portando con sè il bastoncino (testimone). Al termine del proprio giro di pista, il corridore **attuale** deve consegnare il testimone al **successivo** atleta che dovrà a sua volta effettuare il giro di pista.

Scopo della sincronizzazione : lasciare il testimone solo quando è stato già preso

- Guardiamo il dettaglio del passaggio di testimone, analizzando i requisiti:
 - 1) Il **successivo** deve **essere in attesa** dell'arrivo dell'attuale corridore, col la mano protesa all'indietro per ricevere il testimone.
 - 2) **L'attuale** effettua atomicamente due operazioni:
 - 2.1) **Appoggia il testimone nella mano del successivo**, per informarlo di afferrarlo, ma l'attuale non può lasciare il testimone fino a che il successivo non l'ha afferrato.
 - 2.2) Quindi **l'attuale si mette in attesa** di ricevere dal successivo l'urlo di conferma che il successivo ha afferrato il testimone.
 - 3) Il **successivo sente il testimone in mano** ed effettua atomicamente tre operazioni:
 - 3.1) **Afferra strettamente il testimone**, ma non può partire fino a che l'attuale non ha mollato la presa sul testimone, quindi
 - 3.2) **Urla all'attuale** per chiedergli di lasciare la presa e
 - 3.3) **Si mette in attesa di ricevere** a suo volta dall'attuale l'urlo che conferma di avere lasciato la presa sul testimone, per potersi mettere a correre.
 - 4) **L'attuale sente l'urlo del successivo** ed effettua atomicamente due operazioni:
 - 4.1) **lascia la presa** sul testimone e
 - 4.2) **urla al successivo di avere lasciato il testimone e quindi di partire**. Infine, si mette da parte. Ipotizziamo che per andar via ci metta poco tempo.
 - 5) Il **successivo riceve l'urlo dell'attuale che ha lasciato il testimone** e, non atomicamente, si mette a correre assumendo il ruolo di attuale.

Passaggio sicuro di testimone - three way handshake (2)

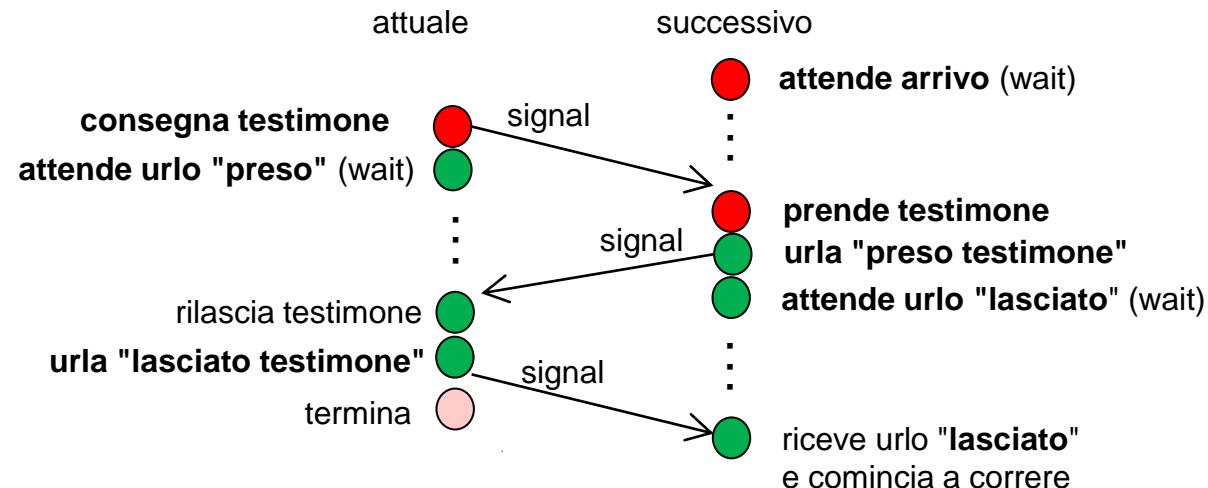
Schema delle azioni del passaggio di testimone (3 wait e 3 signal)



Implementazione Passaggio sicuro di testimone (1)

Quante condition variables? Due

1. **una** mediante la quale tutti i corridori, che ancora non hanno corso, devono attendere l'arrivo del corridore che sta correndo ora.
2. **l'altra** per gli scambi di sincronizzazioni tra i soli due corridori attuale e successivo, dopo che l'attuale ha già appoggiato il testimone in mano al successivo.



Perché NON usare una sola condition variable?

se usassi una sola cond var su quella coda sarebbero in attesa i più corridori in attesa di ricevere il testimone ed anche di volta in volta i due, attuale e successivo, che si stanno scambiando il testimone.

Rischierei di svegliare il thread sbagliato e dovrei fare troppi controlli e risvegli.

Potrei usarne più di due?

Si, potrei usare una condition variable diversa per ciascun punto di attesa (wait), nel nostro caso 3, ma è più efficiente usare meno cond var.
Occorre un compromesso tra efficienza e semplicità di scrittura del codice.

Implementazione Passaggio sicuro di testimone (2)

```
int turno=0; mutex cond_attesa_arrivo_precedente cond_passaggio_testimone

void *corridore (void *arg) { int indice=*((int*)arg); /*indice del corridore*/ int miturno;
    pthread_mutex_lock( &mutex ); turno++; miturno=turno;
    if ( miturno != 1 ) { /* non sono il primo, aspetto di ricevere testimone */
        /* attendo che il corridore precedente mi appoggi il testimone */
        pthread_cond_wait( &cond_attesa_arrivo_precedente,&mutex);
        /* ho preso il testimone, urlo per dire che può lasciarlo */
        pthread_cond_signal( &cond_passaggio_testimone);
        /* per partire aspetto urlo di conferma di avere lasciato il testimone */
        pthread_cond_wait( &cond_passaggio_testimone, &mutex);
    }
    pthread_mutex_unlock( &mutex ); /* parto a correre */
    sleep(1); /* corro */
    if(miturno<NUMCORRIDORI) {
        pthread_mutex_lock( &mutex );
        /* appoggio il testimone in mano al successivo ma non lo mollo */
        pthread_cond_signal(&cond_attesa_arrivo_precedente);
        /* attendo l'urlo del successivo prima di mollare il testimone */
        pthread_cond_wait(&cond_passaggio_testimone,&mutex);
        /* urlo al successivo che ho mollato e puo' partire */
        pthread_cond_signal( &cond_passaggio_testimone);
        pthread_mutex_unlock( &mutex );
    }
    pthread_exit(NULL);
}
```

Il Barbiere appisolato

Il barbiere attende di servire i clienti



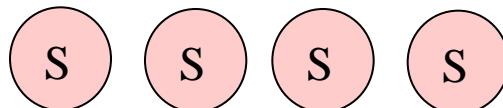
Il Barbiere appisolato (1)

Descrizione del problema :

- Nel negozio di un barbiere ci sono **N sedie** in cui stanno i clienti in attesa del taglio ed **1 poltrona** su cui sta il cliente mentre viene servito e dove pisola il barbiere mentre aspetta che arrivi un qualche cliente.
- Se non ci sono clienti il barbiere pisola.
- Quando un cliente arriva, se c'e' qualche sedia libera allora lui entra e si siede aspettando di essere servito.
- Se tutte le sedie sono occupate invece il cliente se ne va subito.
- Se le sedie sono tutte vuote e il barbiere sta pisolando, il cliente si siede e poi lo chiama per svegliarlo.
- Allora il barbiere si sveglia , fa accomodare il cliente in poltrona (*1) e comincia a servirlo.
- Dopo il taglio (*2), i clienti se ne vanno (*3) ma dopo qualche giorno tornano per un nuovo taglio.

Implementare la sincronizzazione per il sistema descritto, ed in particolare:

- un thread Cliente, di cui saranno lanciate piu' istanze dal main,
- un thread Barbiere di cui sara' lanciata una sola istanza.



N sedie per l'attesa



Poltrona

Il Barbiere appisolato (2) completamento descrizione

Alla descrizione base del barbiere appisolato, occorre aggiungere dei dettagli per ottenere un modello realistico della realtà:

Occorre rispondere a queste domande:

- 1. Come fa il barbiere a sapere che il cliente si è seduto sulla poltrona ?**
- 2. Come fa il cliente a sapere che il barbiere ha finito di servirlo ed è ora di andare via?**
- 3. Per i più puntigliosi, Come fa il barbiere a sapere che il cliente se ne è andato dalla poltrona e quindi che il barbiere può chiamare un altro cliente?**

Aggiungiamo perciò dei dettagli alla descrizione del barbiere:

- Il barbiere dice al cliente di sedersi nella poltrona e attende che questo si sieda.
- Il cliente chiamato dal barbiere si siede e dice al barbiere di essere seduto e pronto ad essere servito, poi si mette in attesa che il barbiere gli dica di andarsene.
- Il barbiere serve il cliente di barba e capelli, poi dice al cliente di andarsene **[e aspetta che il cliente se ne vada.]**
- **Il cliente si alza e dice al barbiere che la poltrona è libera.]**
- Il barbiere chiama un altro cliente perché sa che la poltrona è libera.

Il Barbiere appisolato (3)

Variabili Globali per la Sincronizzazione

- int numclientiincoda=0;
- pthread_mutex_t mutex;
- pthread_cond_t condBarbiereLibero;
- pthread_cond_t condArrivoClienti;
- **pthread_cond_t condSaluto; /* aggiunta per varie sincronizzazioni tra un solo cliente ed il barbiere */**

```
#define NUMSEDIE 5
#define NUMCLIENTI 50

int main () {    pthread_t th;    intptr_t i;    int rc;
    pthread_mutex_init ( &mutex, NULL);
    pthread_cond_init ( &condBarbiereLibero, NULL);
    pthread_cond_init ( &condArrivoClienti, NULL);
    pthread_cond_init ( & condSaluto, NULL);

    numclientiincoda=0;
    rc = pthread_create( &th, NULL, Barbierechesaluta, NULL );
    for( i=0; i< NUMCLIENTI ; i++ )
        rc = pthread_create( &th, NULL, Clientechevuoleesseresalutato, (void*)i );
    pthread_exit( NULL );
}
```

Il Barbiere appisolato (4)

IMPLEMENTAZIONE CHE VA BENE: Usa la stessa condSaluto per 1) accertarsi che il cliente sia arrivato sulla poltrona e quindi sia fermo sulla wait. 2) dire al cliente che puo' alzarsi ed andarsene. 3) attendere che il cliente se ne sia andato dalla poltrona.

```
void *Cliente (void *arg) { int indice = indice passato come argom
    while(1) { /* il cliente aspetta 1/2 sec e poi torna dal barbiere */
        pthread_mutex_lock(&mutex);
        /* se non ci sono sedie libere me ne vado incazzato % @!$&^ */
        if ( numclientiincoda >= NUMSEDIE ) {
            pthread_mutex_unlock ( &mutex );
        } else { /* ci sono sedie libere, mi siedo */
            numclientiincoda++;
            if ( numclientiincoda <= 1 ) /* il barbiere DORME devo svegliarlo */
                pthread_cond_signal ( &condArrivoClienti );
            /* aspetto un assenso dal barbiere per andare alla poltrona */
            pthread_cond_wait ( &condBarbiereLibero, &mutex );
            numclientiincoda--; /* vado nella poltrona del barbiere */
            /* NB: ho spostato la unlock piu' in basso */
            pthread_cond_signal ( &condSaluto ); /* ok, cliente in poltrona */
            printf("il cliente aspetta mentre viene servito\n");fflush(stdout);
            /* aspetto il saluto del barbiere per andare via dalla poltrona */
            pthread_cond_wait ( &condSaluto, &mutex );
            pthread_cond_signal( &condSaluto ); /* cliente dice vado via*/
            pthread_mutex_unlock ( &mutex); /* rilascio mutua esclusione */
            printf("il cliente se ne va \n"); fflush(stdout);
        }
    }
}
```

Il Barbiere appisolato (5)

IMPLEMENTAZIONE CHE VA BENE: Usa la stessa condSaluto per 1) accertarsi che il cliente sia arrivato sulla poltrona e quindi sia fermo sulla wait. 2) dire al cliente che puo' alzarsi ed andarsene. 3) attendere che il cliente se ne sia andato dalla poltrona.

```
void *Barbiere (void *arg) {
    int indice = 0
    while(1) {
        pthread_mutex_lock ( &mutex );
        if ( numclientiincoda <= 0 )          /* il barbiere si appisola */
            pthread_cond_wait( &condArrivoClienti, &mutex );
        /* c'e' clienti in coda, ha cambiato valore di numclientiincoda, ne risveglio 1 */
        pthread_cond_signal( &condBarbiereLibero);
        /* abilito un cliente ad uscire dalla coda e andare nella poltrona del barbiere */
        /*aspetto che il cliente sia in poltrona, pronto a ricevere il saluto del barbiere*/
        pthread_cond_wait ( &condSaluto, &mutex );
        pthread_mutex_unlock ( &mutex);
        /* barbiere serve il cliente */

        printf("barbiere %s finisce il cliente \n"); fflush(stdout);
        pthread_mutex_lock ( &mutex );
        /* barbiere saluta per far alzare il cliente dalla poltrona e buttarlo fuori */
        pthread_cond_signal( &condSaluto );
        pthread_cond_wait ( &condSaluto, &mutex ); /*aspetto cliente va*/
        pthread_mutex_unlock ( &mutex);
        /* il cliente se ne è andato */
    }
}
```

Concetti e Definizioni formali di Programmazione Concorrente

E ora che ormai sapete di cosa parliamo

concetti di programmazione concorrente (1)

Se due thread (o processi) eseguono **in modo concorrente**, l'esecuzione della sequenza di istruzioni macchina di ciascun thread/processo può essere interrotta in un momento qualunque per eseguire un diverso thread o processo (**interleaving**).

Il momento dell'interruzione dipende solo da come il SO schedula i thread, e quindi non è predicibile né controllabile dai thread stessi. La causa più semplice dell'interleaving è il timer: il thread può terminare il proprio quanto di **tempo di esecuzione, assegnatogli dallo scheduler**, in mezzo alla sequenza di istruzioni macchina.

Se due thread/processi cercano di aggiornare una variabile condivisa in modo concorrente, l'esecuzione della sequenza di istruzioni macchina di ciascun thread può essere interrotta in un momento qualunque. In questi casi il valore finale della variabile condivisa dipende dalla sequenza di accessi alla variabile determinata dal SO.

Il termine **Race Condition**: indica proprio la situazione in cui più thread o processi concorrenti accedono (e manipolano) dati condivisi senza controlli di sincronizzazione. In una race condition, il contenuto finale dei dati condivisi dipende dalla sequenza di esecuzione e può causare una **inconsistenza dei dati** dal punto di vista del singolo thread.

Riuscire a far eseguire le sequenze di istruzioni che manipolano dati condivisi come se fossero "**atomiche**" (indivisibili, **senza interleaving con altre sequenze di istruzioni che manipolano quegli stessi dati condivisi**) risolve il problema delle race conditions e assicura la consistenza dei dati condivisi.

concetti di programmazione concorrente (2)

Sezione critica (Critical section) o regione critica o area critica è un segmento di codice in cui un thread ha la necessità di accedere ad una risorsa condivisa avendo la certezza di esserne l'unico ad utilizzare la risorsa condivisa per tutta la durata della propria sezione critica stessa.

Consideriamo perciò alcuni thread T_0, T_1, \dots, T_{n-1} e supponiamo che ciascuno di questi abbia una o più sezioni critiche che accedono alle stesse risorse condivise.

DEFINIZIONE

Definiamo "**Atomica in senso esteso**" l'esecuzione di una sezione critica di uno di quei thread se l'esecuzione di quella sezione critica NON PUO' ESSERE INTERROTTA E INTERVALLATA dall'esecuzione della sezione critica di un altro thread.

In sostanza, l'esecuzione di una sezione critica è atomica in senso esteso se non può esserci interleaving tra sezioni critiche.

Resta invece possibile che l'esecuzione di una sezione critica possa venire interrotta dalla esecuzione di una porzione di codice che NON APPARTIENE a nessuna sezione critica.

concetti di programmazione concorrente (3)

Sezione critica (Critical section) o regione critica o area critica è un segmento di codice in cui un thread accede ad una risorsa condivisa avendo la certezza di esserne l'unico utilizzatore, grazie alla presenza di meccanismi che rendono "**atomica in senso esteso**" l'esecuzione del segmento di codice.

Costruendo una sezione critica intorno all'utilizzo dei dati condivisi in lettura/scrittura si può evitare l'inconsistenza dei dati stessi.

- Più formalmente, consideriamo un sistema costituito da **n** thread $\{T_0, T_1, \dots, T_{n-1}\}$.
- Ogni thread ha una critical section in cui accede in modifica alle variabili condivise.
- L'esecuzione delle regioni critiche deve garantire **mutua esclusione** (un solo thread alla volta esegue la critical section). Più precisamente occorre garantire:
 1. **Mutual Exclusion**: se il thread T_i sta eseguendo la sua critical section, allora nessun altro processo può eseguire critical section.
 2. **Liveness**: il thread non deve rimanere indefinitamente bloccato. La liveness è definita da due proprietà:
 1. **Progress (progresso)**: Se nessun processo è all'interno della sua critical section e ci sono thread in attesa di entrare nella critical section, allora la decisione su quale sarà il primo ad entrare non può essere rimandata indefinitamente.
 2. **Bounded waiting (attesa limitata)**: se un thread richiede l'accesso alla critical section può essergli negata un numero finito di volte (a critical section libera).

concetti di programmazione concorrente (3)

Problemi da evitare nelle Sezione critiche La competizione per le risorse e la conseguente necessità di mutua esclusione e sincronizzazione producono vari possibili **problemi** che dobbiamo cercare di evitare a tutti i livelli architetturali, nel S.O. quando realizziamo syscall e API che possono essere usati da più processi ed anche nelle applicazioni quando scriviamo codice concorrente.

I 3 principali problemi sono:

■ **Busy waiting** («attesa attiva»)

- è un meccanismo di sincronizzazione nel quale un thread che attende il verificarsi di una condizione, lo fa verificando continuamente se la condizione sia diventata vera.
 - ▶ Svantaggio: il thread resta costantemente in esecuzione (ready) e consuma tempo (di CPU + overhead) mentre potrebbe essere sospeso (entrare nella stato di wait).

■ **Deadlock:** stallo, blocco

- (definizione «intuitiva», ne esiste una più formale, che vedremo più avanti). Due o più thread aspettano indefinitamente un evento che può essere causato solo da uno dei thread bloccati.

■ **Starvation** (o indefinite blocking) morte per inedia

- un thread in attesa di una certa risorsa, non riesce mai ad accedervi perché arrivano continue richieste da thread con privilegi maggiori che lo sorpassano e gli rubano la risorsa. Il maggior privilegio può essere causato da un errato algoritmo di sincronizzazione.