Mutex e Variabili condizione

Laboratorio Sistemi Operativi

Antonino Staiano
Email: antonino.staiano@uniparthenope.it

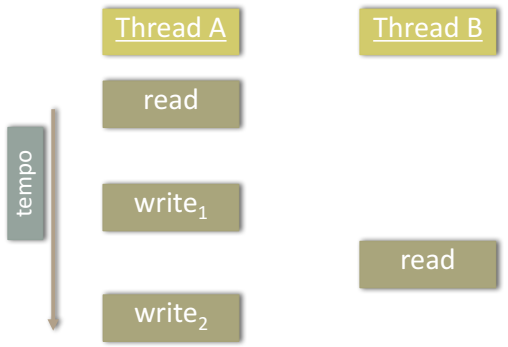
Introduzione

- Quando molteplici thread di un processo condividono la stessa memoria è necessario che ciascun thread mantenga la coerenza dei propri dati
  - Se ciascun thread usa variabili che altri thread non leggono o modificano, allora non c'è alcun problema di consistenza
  - Quando, invece, un thread può modificare una variabile che altri thread possono leggere o modificare, dobbiamo sincronizzare i thread per assicurare che questi non utilizzino un valore non valido quando accedono al contenuto di memoria della variabile

Introduzione (cont.)

- Quando un thread modifica una variabile, altri thread potenzialmente possono vedere delle inconsistenze quando leggono il valore della stessa
- Nella figura che segue è mostrato il caso in cui due thread leggono e scrivono la stessa variabile
  - Il *thread A* legge la variabile e poi scrive un nuovo valore, ma l'operazione di scrittura richiede due cicli di memoria
  - Se il *thread B* legge la stessa variabile tra i due cicli di scrittura, il valore sarà inconsistente

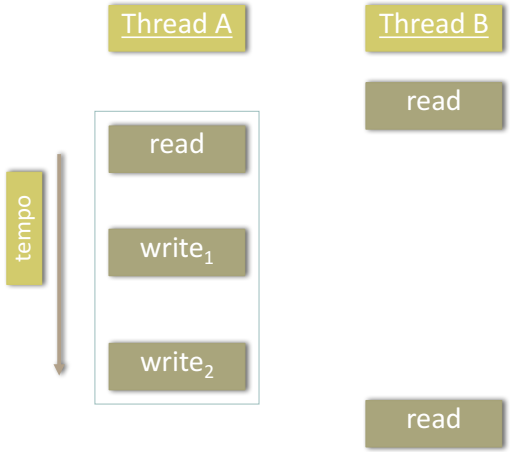
Cicli di memoria intrecciati con due thread



Introduzione (cont.)

- Per risolvere il problema, i thread devono usare un "lock" che consente ad un solo thread alla volta di accedere alla variabile
- Se il *thread B* intende leggere la variabile, allora deve acquisire un lock
  - Similmente, quando il *thread A* aggiorna la variabile, deve acquisire lo stesso lock
  - In questo modo, il *thread B* non sarà in grado di leggere la variabile fino a che il *thread A* rilascia il lock

Due thread che sincronizzano l'accesso in memoria



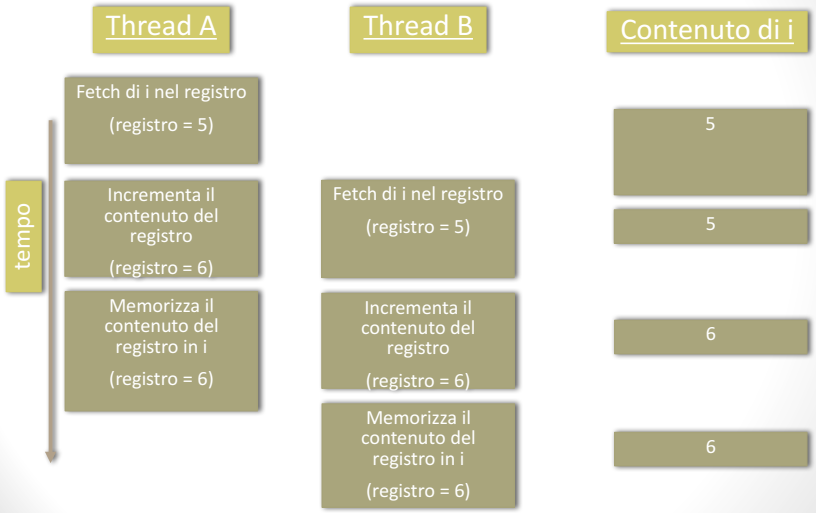
Introduzione (cont.)

- Un altro caso in cui è necessario sincronizzarsi è quando due o più thread cercano di modificare la stessa variabile nello stesso tempo
- Consideriamo il caso in cui viene incrementata una variabile
  - L'operazione di incremento è suddivisa in tre passi:
    1. Leggere la locazione di memoria in un registro
    2. Incrementare il valore nel registro
    3. Scrivere il nuovo valore nella locazione di memoria

Introduzione (cont.)

- Se due thread cercano di incrementare la stessa variabile nello stesso (quasi) momento senza sincronizzarsi, il risultato può essere inconsistente
  - Quando il secondo thread inizia l'operazione, si arriva ad un valore della variabile di uno o due unità più grande rispetto al valore precedente, in base al valore osservato dal secondo thread quando inizia le proprie operazioni
  - Se il secondo thread esegue il passo 1 prima che il primo thread esegue il passo 3, il secondo thread leggerà lo stesso valore iniziale del primo thread, lo incrementa, lo scrive e non si avrà alcun effetto

Due thread non sincronizzati che incrementano la stessa variabile



Introduzione (cont.)

- Se la modifica è atomica, non c'è alcuna "race"
  - Nell'esempio precedente, se l'incremento richiede solo un ciclo di memoria, non avviene alcuna race
- Se i nostri dati appaiono sempre *sequenzialmente consistenti*, allora non c'è necessità di sincronizzazione
  - Le operazioni sono *sequenzialmente consistenti* quando thread multipli non possono osservare inconsistenze nei dati
- Tuttavia, nei moderni calcolatori, gli accessi in memoria avvengono con cicli di bus multipli e i sistemi multiprocessore generalmente intrecciano cicli di bus tra processori multipli, dunque non c'è alcuna garanzia che i nostri dati siano sequenzialmente consistenti

Esempio (race)

```
//Interferenza…include omessi…
void * thread_function(void *);
int myglobal; // variabile globale int
int main(void) {
  pthread_t mythread;
  int i;
   if(pthread_create(&mythread,NULL,thread_function,NULL)){
        printf("creazione del thread fallita.\n");
        exit(1);
  }
  for (i=0; i<20; i++) {
        myglobal=myglobal+1;
        printf("o");
        fflush(stdout);
        sleep(1);
  }
  if (pthread_join(mythread,NULL)) {
        printf("errore nel join dei thread.");
        exit(2);
  }
  printf("\nmyglobal e' uguale a %d\n",myglobal);
  exit(0);}…
```


Esempio (cont.)

```
…
void *thread_function(void *arg) {
  int i,j;
  for (i=0; i<20;i++) {
      j=myglobal;
      j=j+1;
      printf(".");
      fflush(stdout);
      sleep(1);
      myglobal=j;
  }
  return NULL;
}
```

Esempio (cont.)

```
$ gcc race.c -o race –lpthread
Esecuzione:
$ ./race
Possibile Output:
o.o.o.o.o.o..oo.o..o.oo..o.o.oo.o..o.oo.
myglobal e’ uguale a 21
```


Sincronizzazione dei thread

- POSIX.1c mette a disposizione due primitive per la sincronizzazione dei thread nei processi
  - mutex
  - variabili condizione
- POSIX.1b permette di sincronizzare i thread nei processi con i semafori

Mutex

- Un *mutex* (*mut*ual *ex*clusion) è un oggetto che permette a processi o thread concorrenti di sincronizzare l’accesso a dati condivisi
- Un mutex possiede due stati: bloccato e non bloccato:
  - Quando un mutex è bloccato da un thread, gli altri thread che tentano di bloccarlo restano in attesa;
  - Quando il thread bloccante rilascia il mutex, uno dei thread in attesa lo acquisisce
- Ogni volta che un processo o thread ha bisogno di accedere ai dati condivisi, acquisisce il mutex
- Quando l’operazione è terminata, il mutex viene rilasciato, permettendo ad un altro processo o thread di acquisirlo per eseguire le sue operazioni

Mutex (cont.)

- Un mutex è utilizzato per proteggere una sezione critica assicurando che solo un thread per volta esegua il codice nella regione
- Il codice normalmente è del tipo

```
lock_the_mutex(…);
regione critica
unlock_the_mutex(…);
```

- Poiché un solo thread per volta può bloccare il mutex ciò garantisce che un solo thread alla volta possa eseguire le istruzioni nella regione critica

Mutex libreria Pthread

- Un mutex è una variabile rappresentata dal tipo di dato pthread_mutex_t
  - Prima di usare un mutex è necessario inizializzarlo in modo:
    - *Statico*: impostandolo al valore della costante PTHREAD_MUTEX_INITIALIZER
    - *Dinamico*: invocando pthread_mutex_init()
  - Se allochiamo un mutex dinamicamente, è necessario invocare pthread_mutex_destroy() prima di liberare la memoria

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, const
  pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
   /* restituiscono 0 se OK, numero di errore se falliscono */
```


Acquisizione e rilascio dei mutex

- Per bloccare un mutex, un thread usa pthread_mutex_lock()
  - La funzione ritorna quando il mutex è stato bloccato dal thread chiamante
  - Il mutex resta bloccato fino a quando non è sbloccato dal thread chiamante
- Per sbloccare un mutex si usa pthread_mutex_unlock()
  - Se vi sono più thread in attesa di acquisire il mutex, la politica di scheduling dei thread stabilisce chi lo acquisisce
- Per acquisire il blocco o restituire codice di errore (EBUSY), senza bloccare realmente, si usa pthread_mutex_trylock()
  - Tale chiamata permette di far decidere se ci sono alternative rispetto alla semplice attesa

Acquisizione e rilascio dei mutex (cont.)

- I prototipi delle funzioni di acquisizione e rilascio sono

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);

/* restituiscono 0 se OK, numero di errore se
  falliscono */
```


Attributi dei mutex

- Per default, un mutex può essere usato solo da thread che appartengono allo stesso processo
- Utilizzando l'attributo PTHREAD_PROCESS_SHARED si permette a thread di altri processi di utilizzare il mutex (altrimenti si usa PTHREAD_PROCESS_PRIVATE)
- Gli oggetti attributo vanno inizializzati e, se non più necessari, distrutti per non sprecare le risorse di memoria del processo e di sistema
- Per allocare dinamicamente gli attributi di un mutex si usa:
  `int pthread_mutexattr_init(pthread_mutexattr_t *attr)`
- Per deallocare dinamicamente gli attributi di un mutex si usa:
  `int pthread_mutexattr_destroy(pthread_mutexattr_t *attr)`

Assegnazione

- L'attributo PTHREAD_PROCESS_SHARED si assegna mediante

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t
*attr, int pshared);
/* restituisce 0 se OK, numero di errore se fallisce
*/
```

  - Il primo parametro rappresenta l'oggetto attributo inizializzato con pthread_mutexattr_init()
  - Il secondo parametro contiene il valore dell'attributo

- Per sapere quale sia il valore dell'attributo si usa

```
int pthread_mutexattr_getpshared(pthread_mutexattr_t
*attr, int *pshared)
/* restituisce 0 se OK, numero di errore se fallisce */
```


Esempio (no race)

```
void * thread_function(void *);
int myglobal;
pthread_mutex_t mymutex=PTHREAD_MUTEX_INITIALIZER;
int main(void) {
  pthread_t mythread;
  int i;
  if (pthread_create(&mythread,NULL,thread_function,NULL)) {
       printf("creazione del thread fallita.");
       exit(1);}
  for (i=0; i<20; i++) {
       pthread_mutex_lock(&mymutex);
       myglobal = myglobal+1;
       pthread_mutex_unlock(&mymutex);
       printf("o");fflush(stdout);
       sleep(1);
  }
  if(pthread_join(mythread,NULL)) {
       printf("errore nel join con il thread.\n");
       exit(2);
  }
  printf("\nmyglobal è uguale a %d\n",myglobal);
  exit(0);}…
```


Esempio (cont.)

```
…
void *thread_function(void *arg) {
  int i,j;
  for ( i=0; i<20; i++ ) {
       pthread_mutex_lock(&mymutex);
       j=myglobal;
       j=j+1;
       printf(".");
       fflush(stdout);
       sleep(1);
       myglobal=j;
       pthread_mutex_unlock(&mymutex);
  }
  return NULL;
}
```


Esempio (cont.)

```
$ gcc norace.c -o norace -lpthread
Esecuzione:
$ ./norace
Possibile Output:
o.o.o.o..o..o..o.o.o.o.o...o..o..ooooooo
myglobal e' uguale a 40
```

Variabili condizione

- Le variabili condizione costituiscono un ulteriore meccanismo di sincronizzazione per i thread
  - Mentre i mutex implementano la sincronizzazione controllando l'accesso dei thread ai dati usando il polling, le variabili di condizione permettono di sincronizzare i thread sulla base dell'attuale valore dei dati (senza polling)
- Una variabile di condizione è sempre associata ad un mutex lock
- Quando un altro thread causerà l'occorrenza di tale evento, uno o più thread in attesa riceveranno un segnale e si risveglieranno

Variabili condizione

- Le variabili condizione hanno tre componenti: la variabile condizione, un mutex associato, un predicato
- Il programmatore ha il compito di definire tutte e tre le componenti:
  - Il predicato è la condizione (o il valore) che un thread controllerà per determinare se deve attendere;
  - il mutex è il meccanismo che protegge il predicato;
  - la variabile condizione è il meccanismo con cui il thread attende il verificarsi della condizione

Variabili condizione libreria Pthread

- Prima di usare una variabile condizione (che è di tipo pthread_cond_t) è necessario inizializzarla
  - *Staticamente*: mediante la costante PTHREAD_COND_INIZIALIZER
  - *Dinamicamente*: mediante pthread_cond_init()
    - Si usa, successivamente, pthread_cond_destroy() per deallocare una variabile condizione prima di liberare la memoria

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
/* restituiscono 0 se OK, numero errore se falliscono
  */
```


Attendere e segnalare una condizione

- Un thread può attendere su una variabile condizione per un tempo indefinito, invocando `pthread_cond_wait()`
- Oppure per un tempo specifico, invocando `thread_cond_timedwait()`
- Quando la condizione si verifica, si può risvegliare almeno un thread in attesa (`pthread_cond_signal()`) oppure tutti i thread in attesa (`pthread_cond_broadcast()`)

Attendere e segnalare una condizione

```
#include<pthread.h>
int pthread_cond_wait(pthread_cond_t *cptr, pthread_mutex_t *mptr);
int pthread_cond_signal(pthread_cond_t *cptr);
int pthread_cond_broadcast(pthread_cond_t *cptr);
int pthread_cond_timedwait(pthread_cond_t *cptr, pthread_mutex_t
  *mptr, const struct timespec *abstime);
/* ritornano 0 se OK, un errore se falliscono */
```

- Il mutex passato a pthread_cond_wait() protegge la condizione
  - Il chiamante lo passa bloccato alla funzione
  - La funzione pone il thread chiamante nella lista dei thread in attesa della condizione e sblocca il mutex (tutto in modo atomico)
  - Quando pthread_cond_wait() ritorna, il mutex viene di nuovo bloccato

Tipica sequenza di azioni

Thread principale

- Dichiara ed inizializza dati/variabili globali che richiedono sincronizzazione
- Dichiara ed inizializza una variabile condizione
- Dichiara ed inizializza un mutex associato
- Crea thread A e B

Thread A	**Thread B**
•Esegue fino al punto in cui una certa condizione deve verificarsi •Lock il mutex associato e controlla il valore di una variabile globale •Chiama pthread_cond_wait() per effettuare una wait bloccante in attesa del risveglio da parte del thread B (automaticamente e atomicamente corrisponde ad un unlock del mutex associato in modo tale che possa essere usato dal thread B). •Quando risvegliato, lock il mutex in modo automatico e atomico •Unlock il mutex in modo esplicito •Continua	Lavora Lock il mutex associato Modifica il valore della variabile globale su cui il thread A è in attesa Controlla il valore della variabile globale di attesa del thread A. Se si verifica la condizione desiderata, risveglia il thread A invocando pthread_cond_signal() Unlock il mutex Continua

Thread principale

Join / Continua

Esempio (mutex e variabili condizione)

- La condizione è lo stato di una coda di lavoro
  - Proteggiamo la condizione con un mutex e valutiamo la condizione in un ciclo while
  - Quando poniamo un messaggio sulla coda di lavoro, manteniamo il mutex bloccato che, però, non è necessario mantenere bloccato quando segnaliamo ai thread in attesa

Esempio (mutex e variabili condizione)

```
#include<stdio.h>
struct msg {
  struct msg *m_next;
   /* … altra roba … */
};
struct msg *workq;
pthread_cond_t qready = PTHREAD_COND_INITIALIZER;
pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;

void process_msg(void) {
  struct msg *mp;
  for ( ; ; ) {
       pthread_mutex_lock(&qlock);
       while(workq==NULL)
              pthread_cond_wait(&qready, &qlock);
       mp = workq;
       workq = mp -> mnext;
       pthread_mutex_unlock(&qlock);
       / * ora elabora il messaggio mp */
  }
}
```

Esempio (cont.)

```
void enqueue_msg(struct msg *mp) {
   pthread_mutex_lock(&qlock);
   mp->m_next = workq;
   workq = mp;
   pthread_mutex_unlock(&qlock);
   pthread_cond_signal(&qready);
}
```


Produttore Consumatore

- Uno o più produttori (thread o processi) creano elementi che sono elaborati successivamente da uno o più consumatori (thread o processi)
- Gli elementi sono passati tra i produttori e i consumatori usando qualche forma di IPC
- Quando si usa memoria condivisa come forma di IPC tra il produttore ed il consumatore è necessaria una forma appropriata di sincronizzazione
- Cominciamo utilizzando i mutex

Produttore-Consumatore

- Consideriamo thread multipli *produttori* ed un singolo thread *consumatore* in un singolo processo
- Un array di interi `buff` contiene gli elementi prodotti e consumati (i dati condivisi)
- Assumiamo che i produttori si limitano ad impostare `buff[0]` a 0, `buff[1]` a 1 e così via
- Assumiamo che il consumatore accede all'array e verifica che ciascuna entrata sia corretta
- In questo primo esempio ci limitiamo a sincronizzare solo i thread produttori (non avviamo il thread consumatore fino a che tutti i produttori hanno finito)

Produttore-Consumatore 1

```
#include     …

#define      MAXNITEMS           1000000
#define      MAXNTHREADS         100

int          nitems; //sola lettura per prod. e cons.
struct {
  pthread_mutex_t   mutex;
  int buff[MAXNITEMS];
  int nput;
  int nval;
} shared = { PTHREAD_MUTEX_INITIALIZER };
void  *produce(void *), *consume(void *);
```

Variabili globali condivise tra i thread

- Le raggruppiamo in una struttura chiamata `shared` insieme al mutex per sottolineare che a queste variabili si accede solo quando il mutex è bloccato (acquisito)
- `nput` è il prossimo indice in cui bisogna memorizzare nell'array `buff`
- `nval` è il prossimo valore da memorizzare (0,1,2..)
- La struttura è allocata ed il mutex è inizializzato per sincronizzare i thread produttori


```
int main(int argc, char **argv)
{
  int                   i, nthreads, count[MAXNTHREADS];
  pthread_t    tid_produce[MAXNTHREADS], tid_consume;
  if (argc != 3)
       {printf("usage: prodcons <#items> <#threads>");exit(-1);}
  nitems = MIN(atoi(argv[1]), MAXNITEMS);
  nthreads = MIN(atoi(argv[2]), MAXNTHREADS);
       /* inizia tutti i thread produttore */
  for (i = 0; i < nthreads; i++) {
       count[i] = 0;
       pthread_create(&tid_produce[i], NULL, produce, &count[i]);
  }
       /* aspetta tutti i thread produttore */
  for (i = 0; i < nthreads; i++) {
       pthread_join(tid_produce[i], NULL);
       printf("count[%d] = %d\n", i, count[i]);
  }
       /* inizia e poi aspetta il thread consumatore */
  pthread_create(&tid_consume, NULL, consume, NULL);
  pthread_join(tid_consume, NULL);

  exit(0);
}
```


Creazione thread produttori

- I thread produttori sono creati ed ognuno esegue la funzione produce
- L'argomento per ogni produttore è un puntatore ad un elemento dell'array counter
  - Prima si inizializza il contatore a 0 e ogni thread incrementa questo contatore ogni volta che memorizza un elemento nel buffer
- Aspettiamo che tutti i thread produttori terminino e dopo si avvia il thread consumatore
  - Aspettiamo che il consumatore finisca e terminiamo

Produttore

```
void *produce(void *arg)
{
  for ( ; ; ) {
       pthread_mutex_lock(&shared.mutex);
       if (shared.nput >= nitems) {
              pthread_mutex_unlock(&shared.mutex);
              return(NULL); /* array pieno */
       }
       shared.buff[shared.nput] = shared.nval;
       shared.nput++;
       shared.nval++;
       pthread_mutex_unlock(&shared.mutex);
       *((int *) arg) += 1;
  }
}
```

Produttore

- La regione critica per i produttori consiste nel verificare se il buffer è pieno
  - Proteggiamo questa porzione di codice con il mutex, assicurandoci di sbloccarlo appena finito il controllo e l'esecuzione delle relative istruzioni
- Osserviamo che l'incremento dell'elemento count (attraverso arg) non fa parte della regione critica perché ogni thread ha il proprio contatore (una locazione dell'array count nella funzione main)

Consumatore

```
void *
consume(void *arg)
{
  int       i;

  for (i = 0; i < nitems; i++) {
      if (shared.buff[i] != i)
         printf("buff[%d] = %d\n", i, shared.buff[i]);
  }
  return(NULL);
}
```


Consumatore

- Il consumatore verifica il contenuto dell'array controllando che ogni elemento dell'array è corretto e stampa un messaggio in caso di errore

```
$ ./prodcons 1000000 5
count[0] = 167165
count[1] = 249891
count[2] = 194221
count[3] = 191815
count[4] = 196908
```


Locking contro Waiting

- Vogliamo dimostrare che i mutex sono appropriati per il *locking* e non altrettanto per il *waiting* (attesa)
- Modifichiamo l'esempio del produttore-consumatore precedente ed avviamo il thread consumatore appena dopo che i thread produttori sono stati avviati
  - Ciò consente al consumatore di elaborare i dati non appena questi sono generati dai produttori
  - Dobbiamo sincronizzare il consumatore con i produttori per essere certi che il consumatore elabori solo i dati che sono già stati memorizzati dai produttori

Produttore-Consumatore 2

- Di seguito vediamo il codice relativo al main
  - Tutte le linee precedenti la dichiarazione del main non sono cambiate rispetto alla versione 1
- Creiamo il thread consumatore immediatamente dopo aver creato i thread produttori
- La funzione `produce` non cambia rispetto alla versione 1
- Mentre la funzione `consume` chiama una nuova funzione `consume_wait`

Produttore-Consumatore 2: main

```
int main(int argc, char **argv)
{
  int                   i, nthreads, count[MAXNTHREADS];
  pthread_t    tid_produce[MAXNTHREADS], tid_consume;
  if (argc != 3)
       {printf("usage: prodcons2 <#items> <#threads>");exit(-1);}
  nitems = MIN(atoi(argv[1]), MAXNITEMS);
  nthreads = MIN(atoi(argv[2]), MAXNTHREADS);

  for (i = 0; i < nthreads; i++) {
       count[i] = 0;
       pthread_create(&tid_produce[i], NULL, produce, &count[i]);
  }
  pthread_create(&tid_consume, NULL, consume, NULL);
       /* aspetta tutti i produttori e il consumatore */
  for (i = 0; i < nthreads; i++) {
       pthread_join(tid_produce[i], NULL);
       printf("count[%d] = %d\n", i, count[i]);
  }
  pthread_join(tid_consume, NULL);

  exit(0);}
```


Produttore-consumatore 2: produce

```
void *produce(void *arg)
{
  for ( ; ; ) {
       pthread_mutex_lock(&shared.mutex);
       if (shared.nput >= nitems) {
              pthread_mutex_unlock(&shared.mutex);
              return(NULL);      /* array pieno */
       }
       shared.buff[shared.nput] = shared.nval;
       shared.nput++;
       shared.nval++;
       pthread_mutex_unlock(&shared.mutex);
       *((int *) arg) += 1;
  }
}
```


Produttore-consumatore 2: consume

```
void consume_wait(int i)
{
  for ( ; ; ) {
       pthread_mutex_lock(&shared.mutex);
       if (i < shared.nput) {
              pthread_mutex_unlock(&shared.mutex);
              return;          /* un elemento è pronto */
       }
       pthread_mutex_unlock(&shared.mutex);
  }
}
void * consume(void *arg)
{
  int          i;

  for (i = 0; i < nitems; i++) {
       consume_wait(i);
       if (shared.buff[i] != i)
              printf("buff[%d] = %d\n", i, shared.buff[i]);
  }
  return(NULL);
}
```

Consumatore

- Il consumatore deve aspettare
  - La funzione consume chiama consume_wait prima di prelevare il prossimo elemento dall'array
- La funzione consume_wait deve attendere fino a che i produttori hanno generato l'i-esimo elemento
- Per controllare questa condizione, il mutex del produttore è bloccato ed i è confrontato con l'indice nput del produttore
- Dobbiamo acquisire il blocco del mutex prima di controllare nput poiché questa variabile può essere in corso di aggiornamento da uno dei thread produttori

Consumatore

- La questione fondamentale è: cosa possiamo fare quando l'elemento non è disponibile?
  - Effettuiamo un ciclo sbloccando e bloccando il mutex ogni volta
  - Questa operazione è denominata polling e comporta un notevole spreco di tempo di CPU
  - Abbiamo bisogno di un altro tipo di sincronizzazione che consenta ai thread di dormire fino a che si verifichi qualche evento

Variabili condizione: attesa e segnalazione

- I mutex sono per il locking e una variabile condizione è per l'attesa
  - Sono due tipi differenti di sincronizzazione
- E' necessario scegliere la "condizione" da aspettare e notificare
  - Questa è testata nel codice
- Ad una variabile condizione è sempre associato un mutex
  - Quando chiamiamo pthread_cond_wait() per attendere che qualche condizione sia vera, specifichiamo l'indirizzo della variabile condizione e l'indirizzo del corrispondente mutex

Produttore-consumatore 3

- Illustriamo l'uso delle variabili condizione modificando il codice del produttore-consumatore visto in precedenza
- Le due variabili nput ed nval sono associate con il mutex, e mettiamo tutte e tre le variabili in una struttura chiamata put
  - struttura usata dai produttori
- L'altra struttura, nready, contiene un contatore, una variabile condizione e un mutex. Inizializziamo la variabile condizione a PTHREAD_COND_INIZIALIZER

Produttore-consumatore 3

```
#include        …
#define         MAXNITEMS               1000000
#define         MAXNTHREADS             100

        /* globali condivise dai thread */
int             nitems;     /* sola lettura per prod. e cons. */
int             buff[MAXNITEMS];
struct {
  pthread_mutex_t               mutex;
  int           nput;    // indice successivo in cui memorizzare
  int           nval;    // valore successivo da memorizzare
} put = { PTHREAD_MUTEX_INITIALIZER };

struct {
  pthread_mutex_t  mutex;
  pthread_cond_t   cond;
  int              nready;// numero a disposizione del cons.
} nready = { PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER};
/* fine globali */

void    *produce(void *), *consume(void *);
```


Produttore-consumatore 3

```
int
main(int argc, char **argv)
{
  int                   i, nthreads, count[MAXNTHREADS];
  pthread_t     tid_produce[MAXNTHREADS], tid_consume;
  if (argc != 3)
        {printf("usage: prodcons3 <#items> <#threads>");exit(-1);}
  nitems = MIN(atoi(argv[1]), MAXNITEMS);
  nthreads = MIN(atoi(argv[2]), MAXNTHREADS);
        /* crea tutti i produttori ed un consumatore */
  for (i = 0; i < nthreads; i++) {
        count[i] = 0;
        pthread_create(&tid_produce[i], NULL, produce, &count[i]);
  }
  pthread_create(&tid_consume, NULL, consume, NULL);
        /* aspetta tutti i produttori ed il consumatore */
  for (i = 0; i < nthreads; i++) {
        pthread_join(tid_produce[i], NULL);
        printf("count[%d] = %d\n", i, count[i]);
  }
  pthread_join(tid_consume, NULL);
  exit(0);
```


Produttore

```
void *produce(void *arg)
{
  for ( ; ; ) {
        pthread_mutex_lock(&put.mutex);
        if (put.nput >= nitems) {
                pthread_mutex_unlock(&put.mutex);
                return(NULL);           /* array pieno */
        }
        buff[put.nput] = put.nval;
        put.nput++;
        put.nval++;
        pthread_mutex_unlock(&put.mutex);

        pthread_mutex_lock(&nready.mutex);
        if (nready.nready == 0)
                pthread_cond_signal(&nready.cond);
        nready.nready++;
        pthread_mutex_unlock(&nready.mutex);

        *((int *) arg) += 1;
  }
}
```


Produttore

- Usiamo il mutex put.mutex per bloccare la sezione critica quando il produttore pone un nuovo elemento nell'array
- Incrementiamo il contatore nready.nready che conta il numero di elementi pronti per il thread consumatore
- Prima dell'incremento, se il valore del contatore era 0, chiamiamo pthread_cond_signal per risvegliare un qualsiasi thread (l'unico consumatore) in attesa che tale valore diventi diverso da 0
- Possiamo vedere l'interazione del mutex e della variabile condizione associati al contatore (nready)
  - Il contatore è condiviso tra i produttori e il consumatore quindi l'accesso deve avvenire quando il mutex associato (nready.mutex) è bloccato
  - La variabile condizione è usata per aspettare e segnalare

Consumatore

```
void *consume(void *arg)
{
  int        i;

  for (i = 0; i < nitems; i++) {
       pthread_mutex_lock(&nready.mutex);
       while (nready.nready == 0)
          pthread_cond_wait(&nready.cond, &nready.mutex);
       nready.nready--;
       pthread_mutex_unlock(&nready.mutex);

       if (buff[i] != i)
          printf("buff[%d] = %d\n", i, buff[i]);
  }
  return(NULL);
}
```


Consumatore

- Il consumatore aspetta che nready.nready sia diverso da zero
- Poiché esso è condiviso tra i produttori ed il consumatore, possiamo testare il suo valore solo mentre il mutex associato è bloccato
- Se, mentre il mutex è bloccato, il valore è 0, chiamiamo pthread_cond_wait() per attendere. Ciò effettua due azioni in modo atomico:
  - Il mutex nready.mutex è sbloccato e
  - Il thread è messo in attesa fino a che qualche altro thread chiama pthread_cond_signal() per questa variabile condizione

Consumatore

- Prima di ritornare, phtread_cond_wait blocca il mutex nready.mutex
  - quando ritorna e troviamo che il contatore è diverso da zero, decrementiamo il contatore (sapendo che il mutex è bloccato) e poi sblocchiamo il mutex
- In questa implementazione, la variabile che mantiene la condizione è un contatore intero e l'impostazione della condizione è semplicemente l'incremento del contatore
  - In questo caso si è ottimizzato il codice in modo che il segnale si verifica solo quando il contatore va da 0 a 1

Esempio 1(uso dei mutex)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define MAX 10
pthread_mutex_t M; /* def.mutex condiviso tra thread */
int DATA=0;        /* variabile condivisa */
int accessi1=0;  // num. accessi thread 1 alla sez. critica
int accessi2=0;  // num. accessi thread 2 alla sez. critica
void *thread1_process (void * arg) {
  int k=1;
  while(k) {
       accessi1++;
       pthread_mutex_lock(&M);
       DATA++;
       k=(DATA>=MAX?0:1);
       pthread_mutex_unlock(&M);
       printf("accessi di T1: %d\n", accessi1);
       sleep(1);
  }
pthread_exit (0);
}
```

Esempio 1 (cont.)

```
void *thread2_process (void * arg) {
  int k=1;
  while(k)
  {
      accessi2++;
      pthread_mutex_lock(&M);
      DATA++;
      k=(DATA>=MAX?0:1);
      pthread_mutex_unlock(&M);
      printf("accessi di T2: %d\n", accessi2);
      sleep(1);
  }
pthread_exit (0);
}
```


Esempio 1(cont.)

```
int main () {
pthread_t th1, th2; // il mutex è inizialmente libero
pthread_mutex_init (&M, NULL);
if (pthread_create(&th1, NULL, thread1_process, NULL) < 0)
  {
    fprintf (stderr, "errore creazione per thread 1\n");
    exit (1);
  }
if (pthread_create(&th2, NULL,thread2_process,NULL) < 0)
  {
      fprintf (stderr, "errore creazione per thread 2\n");
      exit (1);
  }
pthread_join (th1, NULL);
pthread_join (th2, NULL);
printf("Accessi: T1: %d, T2 %d\n",accessi1,accessi2);
printf("Totale accessi: %d\n",DATA);
exit(0);
}
```


Esercizio 1

- Si realizzi un programma in C e Posix sotto Linux che, utilizzando la libreria Pthread
  - lancia n thread per cercare un elemento in una matrice nxn di caratteri
  - Ognuno dei thread cerca l'elemento in una delle righe della matrice
  - Non appena un thread ha trovato l'elemento cercato, rende note agli altri thread le coordinate dell'elemento e tutti i thread terminano (sono cancellati)

Esercizio 2

- Si realizzi un programma C e Posix in ambiente Linux che, impiegando la libreria Pthread, generi tre thread
  - I primi due thread sommano 1000 numeri generati casualmente ed ogni volta incrementano un contatore
  - Il terzo thread attende che il contatore incrementato dai due thread raggiunga un valore limite fornito da riga di comando, notifica l'avvenuta condizione e termina
  - Utilizzare le variabili condizione