Sincronizzazione avanzata in Linux

Condition Variables: teoria, semantica e pratica POSIX

Roberto Marino, PhD

Università di Messina
CdL in Informatica — Reti e Sistemi Distribuiti Mod.B

7 maggio 2026



UniMe
1548

Un mutex non basta

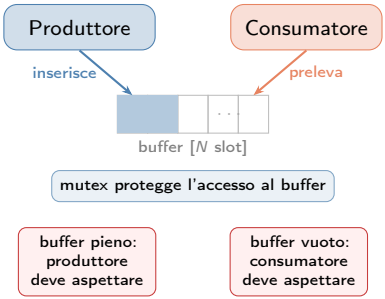
Il **mutex** garantisce mutua esclusione: un thread per volta. Non risponde però a una domanda diversa:

Il problema della sincronizzazione temporale

Come fa un thread ad **aspettare** che lo stato condiviso soddisfi una condizione, *senza tenere il mutex bloccato* e *senza sprecare CPU*?

Scenario: **buffer a capacità limitata** (N slot).

- Il **consumatore** non può prelevare se il buffer è vuoto
- Il **produttore** non può inserire se il buffer è pieno
- Entrambi *devono bloccarsi* finché l'altro non agisce



Tentativo 1: busy-wait con mutex tenuto

Prima idea: controllare `count` dentro la sezione critica, aspettando in un loop che diventi positivo.

```
1 /* consumatore */
2 pthread_mutex_lock(&mtx);
3 while (count == 0)
4     ; /* aspetta... ma il mutex
5          e' ancora tenuto! */
6 item = buf[head++ % N];
7 count--;
8 pthread_mutex_unlock(&mtx);
```

```
1 /* produttore */
2 pthread_mutex_lock(&mtx); /* BLOCCA */
3 buf[tail++ % N] = item;
4 count++;
5 pthread_mutex_unlock(&mtx);
```

Cosa succede:

1. Il consumatore acquisisce il mutex
2. Trova `count == 0`, entra nel loop
3. Gira a vuoto tenendo il mutex bloccato
4. Il produttore tenta `lock`: **si blocca**
5. Nessuno dei due può avanzare

Deadlock garantito

Il consumatore aspetta che `count` cresca, ma tiene il mutex. Il produttore aspetta il mutex per far crescere `count`. Ognuno aspetta l'altro per sempre.

Tentativo 2: rilasciare il mutex e riprovare

```
1  /* consumatore */
2  while (1) {
3      pthread_mutex_lock(&mtx);
4      if (count > 0) break;
5      pthread_mutex_unlock(&mtx);
6      /* rilasciato: il produttore
7         puo' ora inserire */
8  }
9  item = buf[head++ % N];
10 count--;
11 pthread_mutex_unlock(&mtx);
```

Rispetto al tentativo 1 il mutex viene rilasciato prima di aspettare: il deadlock scompare.

Il produttore può inserire mentre il consumatore gira nel loop. Prima o poi `count` diventa positivo e il consumatore esce.

Il problema che resta

Il consumatore controlla `count` in continuazione, anche quando il buffer è vuoto da ore. Occupa un intero core della CPU senza fare nulla di utile.

Serve un modo per **sospendere** il thread finché non c'è lavoro, senza tenerlo in loop. Questo è esattamente il problema che le condition variables risolvono.

Il requisito: cosa serve davvero

Emergono due requisiti **non negoziabili**:

Requisito 1 — Atomicità

Rilasciare il mutex e sospendere il thread devono avvenire come un'unica operazione **indivisibile**.

Altrimenti, qualsiasi signal emesso tra il rilascio e la sospensione va **perso** — il thread dorme per sempre.

Questa è la *finestra di race* del tentativo 2: non è eliminabile a livello utente, serve supporto del kernel.

Requisito 2 — Efficienza

Un thread in attesa non deve consumare CPU. Deve essere **rimosso dallo scheduler** finché la condizione non è soddisfatta.

Il busy-wait del tentativo 1 e 2 non lo garantisce: spreca core, scalda la CPU, disturba gli altri thread.

Su sistemi embedded o con molti thread il degrado è catastrofico.

La condition variable è la primitiva del kernel che soddisfa entrambi i requisiti: `wait(cv, mutex)` rilascia il mutex e sospende il thread in modo *atomico*, senza nessuna finestra.

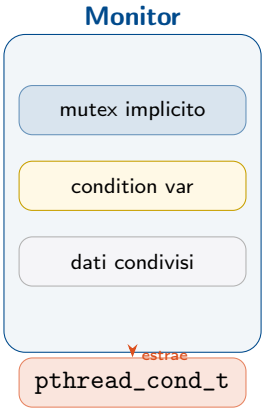
Dalle origini: i Monitor di Hoare

C.A.R. Hoare (1974) e **P.B. Hansen (1975)** propongono il *monitor*: una struttura che incapsula dati, mutex implicito e meccanismo di attesa.

Un monitor garantisce:

- un solo thread attivo per volta (mutex implicito)
- la possibilità di **sospendersi** dentro il monitor cedendo temporaneamente il lock
- il risveglio automatico quando la condizione cambia

POSIX (1993/2001) non implementa i monitor direttamente, ma ne estrae la primitiva essenziale: la **condition variable**.



Semantica: le tre operazioni fondamentali

`wait(cv, mutex)`

1. Rilascia il mutex
2. Sospende il thread
3. Al risveglio riacquisisce il mutex

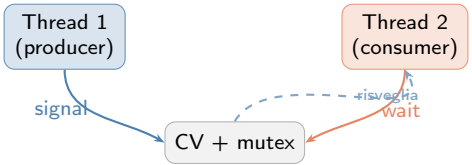
I passi 1 e 2 sono **atomici**: nessuna race.

`signal(cv)`

- Sveglia **uno** dei thread in attesa
- Se nessuno aspetta: no-op
- Il thread svegliato compete per il mutex

`broadcast(cv)`

- Sveglia **tutti** i thread in attesa
- Ognuno compete per il mutex
- Solo uno per volta entra nella sezione critica



Soluzione corretta: Produttore/Consumatore con condition variables

Produttore

```
1  pthread_mutex_lock(&mtx);
2
3  while (count == N)
4      pthread_cond_wait(
5          &not_full, &mtx);
6
7  buf[tail++ % N] = item;
8  count++;
9
10 pthread_cond_signal(&not_empty);
11 pthread_mutex_unlock(&mtx);
```

Consumatore

```
1  pthread_mutex_lock(&mtx);
2
3  while (count == 0)
4      pthread_cond_wait(
5          &not_empty, &mtx);
6
7  item = buf[head++ % N];
8  count--;
9
10 pthread_cond_signal(&not_full);
11 pthread_mutex_unlock(&mtx);
```

`not_full`: segnalata dal consumatore quando libera uno slot. `not_empty`: segnalata dal produttore quando inserisce un elemento.

Produttore/Consumatore: anatomia del codice (1/2)

```
1  pthread_mutex_lock(&mtx);       // (1)
2
3  while (count == N)              // (2)
4      pthread_cond_wait(          // (3)
5          &not_full, &mtx);
6
7  buf[tail++ % N] = item;         // (4)
8  count++;
9
10 pthread_cond_signal(&not_empty); // (5)
11 pthread_mutex_unlock(&mtx);
```

(1) — Acquisizione del mutex
Il mutex non protegge solo `buf`: protegge *l'invariante* dell'intero sistema, cioè il fatto che `count`, `head` e `tail` siano sempre coerenti tra loro. Senza il mutex, due produttori potrebbero scrivere nella stessa cella contemporaneamente.

(2) — `while`, non `if`
Al risveglio dalla `wait` la condizione *potrebbe non essere più vera*. Esempio: due produttori aspettano su `not_full`; il consumatore libera un solo slot e chiama `signal`; uno dei due produttori si sveglia e riempie lo slot; il secondo si sveglia per spurious wakeup e trova `count == N` di nuovo. Senza `while` scriverebbe fuori dal buffer.

Produttore/Consumatore: anatomia del codice (2/2)

```
1  pthread_mutex_lock(&mtx);        // (1)
2
3  while (count == N)               // (2)
4      pthread_cond_wait(           // (3)
5          &not_full, &mtx);
6
7  buf[tail++ % N] = item;          // (4)
8  count++;
9
10 pthread_cond_signal(&not_empty); // (5)
11 pthread_mutex_unlock(&mtx);
```

(3) — Atomicità di `wait`
`pthread_cond_wait` fa tre cose in sequenza atomica: rilascia il mutex, sospende il thread nella coda della CV, al risveglio riacquisisce il mutex. L'atomicità dei primi due passi è fondamentale: se tra il rilascio e la sospensione arrivasse un `signal`, andrebbe perso e il thread dormirebbe per sempre.

(4) — Sezione critica
Qui `count < N` è una garanzia assoluta. Nessun altro thread può modificare `count` tra il controllo e la scrittura.

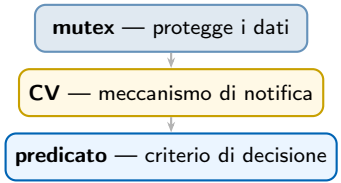
(5) — `signal` e non `broadcast`
Abbiamo inserito *un* elemento: al massimo un consumatore può procedere. Svegliarne più d'uno con `broadcast` è corretto ma inefficiente: tutti si sveglierebbero, uno prenderebbe l'elemento, gli altri tornerebbero a dormire dopo aver riacquisito e rilasciato il mutex inutilmente.

CV e condizione: due cose distinte

La condition variable **non trasporta informazioni**.

`pthread_cond_signal` non dice *cosa* è cambiato: dice solo "qualcosa è cambiato, vai a controllare".

Ogni sistema con CV ha quindi **tre componenti separati**:



I tre lavorano insieme: senza il predicato la CV non ha senso da sola.

```c
pthread_mutex_lock(&mtx);

/*  CV: il meccanismo che sospende
 *  e sveglia il thread.
 *
 *  "count == 0" e' il PREDICATO:
 *  la proprieta' dello stato che
 *  il thread controlla ogni volta
 *  che viene svegliato.
 *
 *  Il while ri-valuta il predicato
 *  dopo ogni risveglio, perche'
 *  la CV non garantisce che il
 *  predicato sia diventato vero:
 *  garantisce solo che qualcosa
 *  e' cambiato.              */

while (count == 0)
    pthread_cond_wait(&cv, &mtx);

/* Qui il predicato e' sicuramente
 * vero: count > 0.           */
count--;

pthread_mutex_unlock(&mtx);
```

Signal vs Broadcast

`pthread_cond_signal`

Sveglia **un solo** waiter.

- Usare quando tutti i waiter hanno *la stessa* condizione da controllare
- Usare quando solo un thread può comunque procedere
- Più efficiente: un solo risveglio, una sola competizione

Esempio: buffer con un solo slot libero

`pthread_cond_broadcast`

Sveglia **tutti** i waiter.

- Usare quando i waiter hanno condizioni *diverse*
- Usare per eventi globali (shutdown, resize, . . .)
- Sempre corretto; meno efficiente con molti thread

Esempio: barriera, reader/writer

▷ **Regola pratica:** in dubbio usa `broadcast`. Ottimizza con `signal` solo dopo aver misurato e solo se sei certo che tutti i waiter abbiano la stessa condizione.

API POSIX — `pthread_cond_t`

Funzione	Scopo
`pthread_cond_init(&cv, NULL)`	Inizializzazione dinamica
`PTHREAD_COND_INITIALIZER`	Inizializzazione statica (globale)
`pthread_cond_wait(&cv, &mtx)`	Attesa (rilascia mutex, dorme, riprende)
`pthread_cond_timedwait(&cv, &mtx, &ts)`	Come wait, con timeout assoluto
`pthread_cond_signal(&cv)`	Sveglia un waiter
`pthread_cond_broadcast(&cv)`	Sveglia tutti i waiter
`pthread_cond_destroy(&cv)`	Distruzione

Nota su `timedwait`:
il timeout è un *tempo assoluto* (`struct timespec`), non un intervallo. Si calcola con `clock_gettime` + offset.

Una CV, un mutex:
ogni CV va usata *sempre* con lo stesso mutex. Mischiare mutex diversi con la stessa CV è comportamento non definito.