

SINCRONIZZAZIONE DEI PROCESSI NEI S.O.

Danilo Croce

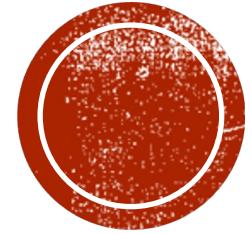
Novembre 2024



OUTLINE

- **Programmazione concorrente e thread:**
 - Analisi del problema
- **La mutua esclusione:**
 - I "semafori"
 - Mutex e Pthreads
 - I monitor
 - Scambi di messaggi





PROGRAMMAZIONE CONCERENTE E THREAD: IL PROBLEMA

SINCRONIZZAZIONE E COMUNICAZIONE TRA PROCESSI (IPC)

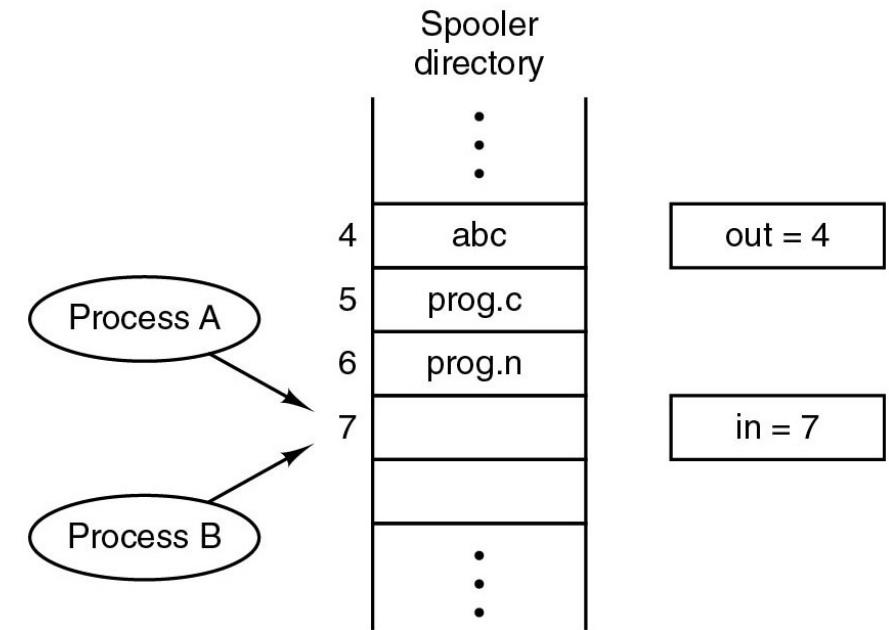
- I processi hanno bisogno di un modo per **comunicare**:
 - Condividere i dati durante l'esecuzione
- Nessuna condivisione esplicita tra processi:
=> I dati devono essere scambiati normalmente tra i processi
- I processi hanno bisogno di un modo per **sincronizzarsi**:
 - Per tenere conto delle dipendenze
 - Per evitare che si intralcino a vicenda
 - Si applica anche all'esecuzione multithread



RACE CONDITIONS



- Il processo **A** legge $in=7$ e decide di aggiungere il suo file in quella posizione.
 - **A** viene sospeso dal Sistema operativo (perché il suo slot è scaduto)
 - Anche il processo **B** legge $in=7$ e inserisce il suo file in quella posizione.
 - **B** imposta $in=8$ e alla fine viene sospeso.
 - **A** scrive il suo file nella posizione 7
-
- **Problema:** la lettura/aggiornamento di un file dovrebbe essere un'azione atomica. Se non lo è, i processi possono “gareggiare” tra loro e giungere a conclusioni errate.



CRITICAL REGION (1)

Requisiti per evitare “race conditions”:

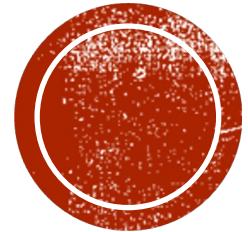
1. Due processi non possono trovarsi contemporaneamente all'interno delle rispettive regioni critiche.
2. Non si possono fare ipotesi sulla velocità o sul numero di CPU.
3. Nessun processo in esecuzione al di fuori della propria regione critica può bloccare altri processi.
4. Nessun processo deve aspettare all'infinito per entrare nella propria regione critica.



CRITICAL REGION (2)

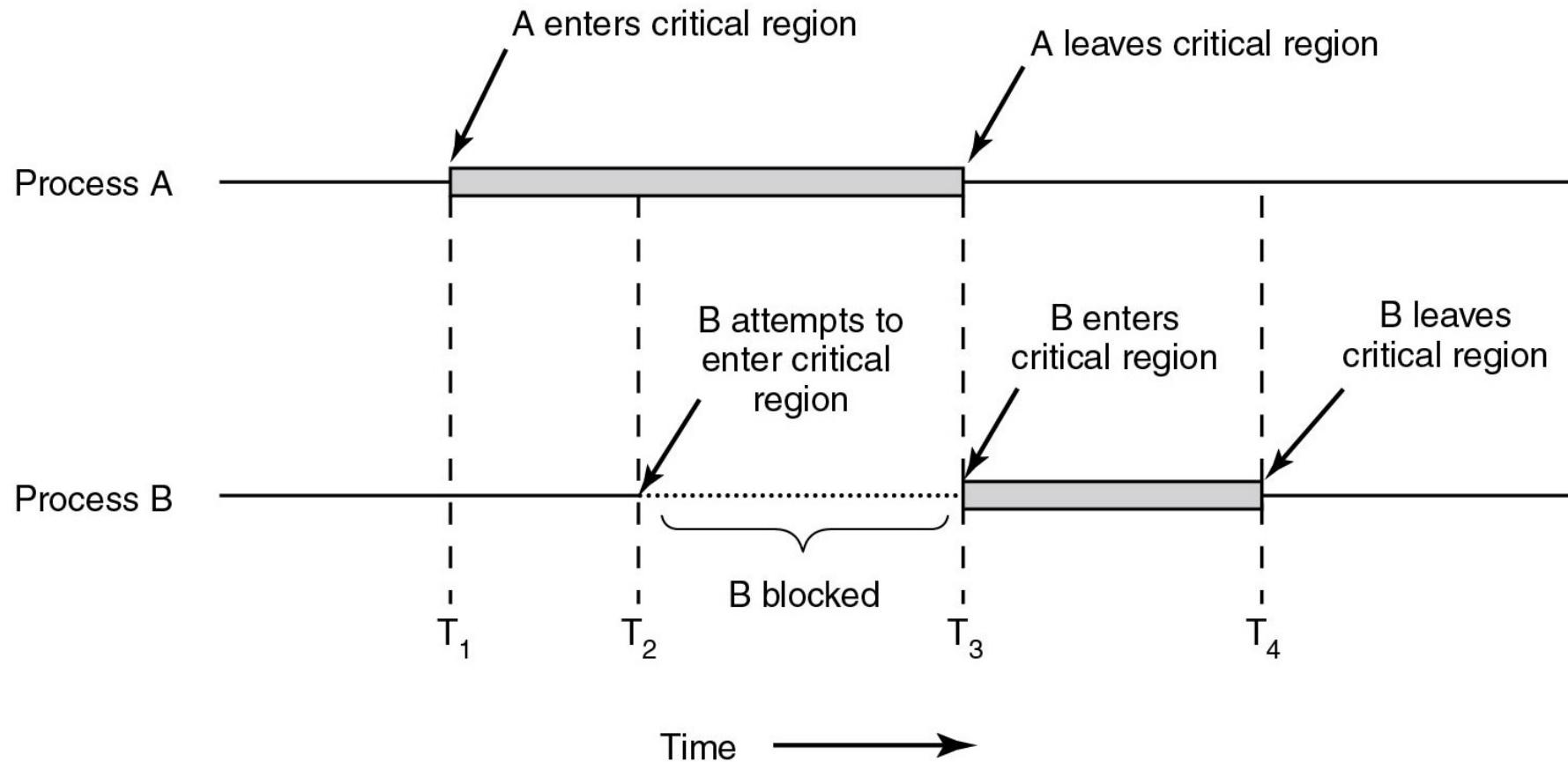
- (NON) soluzioni:
 - **Disabilitare gli interrupt:** impedisce semplicemente che la CPU possa essere riallocata. Funziona solo per sistemi a CPU singola
 - **Bloccare le variabili:** proteggere le regioni critiche con variabili 0/1. Le “corse” si verificano ora sulle variabili di blocco.





LA MUTUA ESCLUSIONE

CRITICAL REGION (3)



Esclusione reciproca tramite regioni critiche.



ESCLUSIONE RECIPROCA CON *BUSY WAITING*: ALTERNANZA RIGOROSA

```
while (TRUE) {  
    while(turn != 0);  
    critical_region();  
    turn = 1;  
    noncritical_region();  
}
```

```
while (TRUE) {  
    while(turn != 1);  
    critical_region();  
    turn = 0;  
    noncritical_region();  
}
```

- Purtroppo, questa è un'altra (non) soluzione:
 - Non permette ai processi di entrare nelle loro regioni critiche per due volte di seguito.
 - **Un processo fuori dalla regione critica può effettivamente bloccarne un altro**



ESCLUSIONE RECIPROCA CON BUSY WAITING: PETERSON'S ALGORITHM

- Alice e Bob vogliono usare un'unica postazione computer in un ufficio. Ma ci sono delle regole:
 - Solo una persona può usare il computer alla volta.
 - Se entrambi vogliono usarlo contemporaneamente, devono decidere chi va per primo.
- **Idea** dell'algoritmo
 - Alice o Bob devono segnalare il loro interesse a usare il computer.
 - Se l'altro non è interessato, la persona interessata può usarlo subito.
 - Se entrambi mostrano interesse, registrano il loro nome su un foglio. Ma se scrivono quasi allo stesso tempo, l'ultimo nome sul foglio ha la precedenza.
 - La persona che non ha la precedenza aspetta finché l'altra ha finito.
 - Una volta finito, la persona che ha usato il computer segnala che ha finito, e l'altra può iniziare.



ESCLUSIONE RECIPROCA CON BUSY WAITING: PETERSON'S ALGORITHM

```
#define N      2                                /* numero di processi */

int turn;                                     /* A chi tocca? */
int interested[N];                            /* Tutti i valori inizialmente 0 (FALSE) */

void enter_region(int process);                /* process è 0 o 1 */

{
    int other;                                  /* numero dell'altro processo */
    other = 1 - process;                        /* l'opposto del processo */
    interested[process] = TRUE;                 /* mostra che si è interessati */
    turn = process;                            /* imposta il flag */
    while (turn == process && interested[other] == TRUE) /* istruzione null */ ;
}

void leave_region(int process)                  /* process: chi esce */

{
    interested[process] = FALSE;                /* indica l'uscita dalla regione critica */
}
```



COME EVITARE I BUSY WAITING?

- Le soluzioni finora adottate consentono a un processo di tenere occupata la CPU in attesa di poter entrare nella sua regione critica. (**spin lock**)
SPRECO DI RISORSE!!! 😞
- Soluzione:** lasciare che un processo in attesa di entrare nella sua regione critica restituisca volontariamente la CPU allo scheduler

```
void sleep() {  
    set own state to BLOCKED;  
    give CPU to scheduler;  
}
```

```
void wakeup(process) {  
    set state of process to READY;  
    give CPU to scheduler;  
}
```



PROGRAMMAZIONE CONCORRENTE NEL PROBLEMA PRODUTTORE-CONSUMATORE

- Nel problema del **produttore-consumatore**, due processi condividono un buffer di dimensioni fisse.
- Il produttore inserisce informazioni nel buffer, mentre il consumatore le preleva.
- Il produttore si addormenta (entra in modalità «sleep») se il buffer è pieno e viene risvegliato (viene riattivato) quando il consumatore preleva dati.
- Analogamente, il consumatore dorme se il buffer è vuoto e viene risvegliato quando il produttore inserisce dati.



PRODUTTORE-CONSUMATORE (1 OF 4)

```
#define N 100
int count=0;

void producer(void) {
    int item;
    while(TRUE) {
        item = produce_item();
        if(count==N) sleep();
        insert_item(item);
        count++;
        if(count==1) wakeup(cons);
    }
}

void consumer(void) {
    int item;
    while(TRUE) {
        if(count==0) sleep();
        item = remove_item();
        count--;
        if(count==N-1) wakeup(prod);
        consume_item(item);
    }
}
```



PRODUTTORE-CONSUMATORE (2 OF 4)

```
#define N 100
int count=0;

void producer(void) {
    int item;
    while(TRUE) {
        item = produce_item();
        if(count==N) sleep();
        insert_item(item);
        count++;
        if(count==1) wakeup(cons);
    }
}
```

Producer sleeps
when buffer is full

```
void consumer(void) {
    int item;
    while(TRUE) {
        if(count==0) sleep();
        item = remove_item();
        count--;
        if(count==N-1) wakeup(prod);
        consume_item(item);
    }
}
```



PRODUTTORE-CONSUMATORE (3 OF 4)

```
#define N 100
int count=0;

void producer(void) {
    int item;
    while(TRUE) {
        item = produce_item();
        if(count==N) sleep();
        insert_item(item);
        count++;
        if(count==1) wakeup(cons);
    }
}
```

```
void consumer(void) {
    int item;
    while(TRUE) {
        if(count==0) sleep();
        item = remove_item();
        count--;
        if(count==N-1) wakeup(prod);
        consume_item(item);
    }
}
```

Consumer sleeps
when buffer is empty



PRODUTTORE-CONSUMATORE (4 OF 4)

Il consumatore potrebbe essere
risvegliato un attimo prima di andare
a dormire... e nessuno lo
risveglierebbe più ☺

Problem: wake up
events may get lost!
Sample run:
1.Con, 2.Prd, 3.Con
→ Cause? Effect?

```
#define N 100
int count=0;
```

```
void producer(void) {
    int item;
    while(TRUE) {
        item = produce_item();
        if(count==N) sleep();
        insert_item(item);
        count++;
        if(count==1) wakeup(cons);
    }
}
```

```
void consumer(void) {
    int item;
    while(TRUE) {
        if(count==0) sleep();
        item = remove_item();
        count--;
        if(count==N-1) wakeup(prod);
        consume_item(item);
    }
}
```



BIT DI ATTESA DEL WAKEUP

- **Problema:**

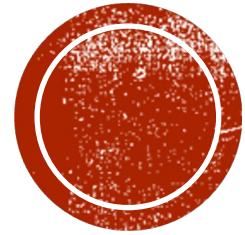
- Il consumatore potrebbe essere risvegliato un attimo prima di andare a dormire
- Bit di Attesa può essere aggiunto come rimedio per segnali di risveglio persi.

- **Possibile soluzione**

- Si attiva quando un processo non dormiente riceve un 'wakeup'.
- Se acceso, previene l'entrata in 'sleep' del processo e viene poi spento.
- Funziona come accumulatore per segnali di risveglio.
- Viene resettato dal consumatore ad ogni ciclo.

- *E' un workaround, non funziona sempre...*





LA MUTUA ESCLUSIONE: I SEMAFORI

I SEMAFORI

- Creato da E. W. Dijkstra nel 1965 per contare e gestire i "wakeup".
- **Valori:** Può essere:
 - 0 (nessun wakeup)
 - positivo (wakeup in attesa).
- **Operazioni:**
 - down
 - Se il valore del semaforo è maggiore di zero, questo valore viene decrementato, e il processo continua la sua esecuzione.
 - Se il valore del semaforo è 0, il processo che ha invocato `down` viene bloccato e messo in una coda di attesa associata al semaforo.
 - In altre parole, il processo "va a dormire".
 - Up
 - Se il valore è 0, ci sono processi nella coda di attesa, vengono «svegliati» (eventualmente per entrare in competizione ed eseguire di nuovo `down`).
 - In ogni caso, il valore viene incrementato e il processo continua la sua esecuzione.



I SEMAFORI

- **Atomicità:** Le operazioni sui semafori sono «indivisibili», evitando conflitti.
- **Problema Produttore-Consumatore:** Uso dei semafori per gestire accesso e capacità di un buffer.
 - Tipi di Semafori:
 - mutex (mutual exclusion, accesso esclusivo),
 - full (tutti posti occupati)
 - empty (tutti posti liberi).
 - Uso:
 - mutex previene accessi simultanei,
 - full e empty coordinano attività.



SEMAFORI: PRODUTTORE-CONSUMATORE

```
#define N 100

typedef int sema;
sema mutex=1;
sema empty=N, full=0;

void producer(void) {
    int item;
    while(TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

```
void consumer(void) {
    int item;
    while(TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

mutex serializes
access to the shared
buffer



SEMAFORI: PRODUTTORE-CONSUMATORE

```
#define N 100

typedef int sema;
sema mutex=1;
sema empty=N, full=0;

void producer(void) {
    int item;
    while(TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

```
void consumer(void) {
    int item;
    while(TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

empty semaphore
blocks the producer
when the shared
buffer is full



SEMAFORI: PRODUTTORE-CONSUMATORE

```
#define N 100

typedef int sema;
sema mutex=1;
sema empty=N, full=0;

void producer(void) {
    int item;
    while(TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

```
void consumer(void) {
    int item;
    while(TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

full semaphore
blocks the consumer
when the buffer is
empty



SEMAFORI: PRODUTTORE-CONSUMATORE

```
#define N 100

typedef int sema;
sema mutex=1;
sema empty=N, full=0;

void producer(void) {
    int item;
    while(TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void) {
    int item;
    while(TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```



LETTORI E SCRITTORI: REGOLE E PROBLEMI

- **Regola Base:** In ogni momento, possono essere ammessi
 - R lettori
 - solo 1 scrittore.
- **Esempio:** Si possono avere molteplici letture su un database, ma solo un singolo scrittore.
- **Funzionamento Sintetico:**
 - Il primo lettore blocca l'accesso al database.
 - Lettori successivi incrementano un contatore.
 - L'ultimo lettore libera l'accesso al database così gli scrittori possono fare il loro lavoro.



READERS/WRITERS (1 OF 6)

- N processi accedono (cioè leggono o scrivono) ad alcuni dati condivisi.
- In qualsiasi momento: **R lettori o 1 scrittore ammessi**. Soluzione di base:

```
typedef int sema;  
sema mutex = 1;  
sema db = 1;  
int rc = 0;
```

```
void reader() {  
    while(TRUE) {  
        down(&mutex);  
        rc++;  
        if(rc==1) down(&db);  
        up(&mutex);  
        read_db();  
        down(&mutex);  
        rc--;  
        if(rc==0) up(&db);  
        up(&mutex);  
        use_data_read();  
    }  
}
```

```
void writer() {  
    while(TRUE) {  
        think_up_data();  
        down(&db);  
        write_db();  
        up(&db);  
    }  
}
```



READERS/WRITERS (2 OF 6)

- N processi accedono (cioè leggono o scrivono) ad alcuni dati condivisi.
- In qualsiasi momento: **R lettori o 1 scrittore ammessi**. Soluzione di base:

```
typedef int sema;  
sema mutex = 1;  
sema db = 1;  
int rc = 0;
```

```
void reader() {  
    while(TRUE) {  
        down(&mutex);  
        rc++;  
        if(rc==1) down(&db);  
        up(&mutex);  
        read_db();  
        down(&mutex);  
        rc--;  
        if(rc==0) up(&db);  
        up(&mutex);  
        use_data_read();  
    }  
}
```

```
void writer() {  
    while(TRUE) {  
        think_up_data();  
        down(&db);  
        write_db();  
        up(&db);  
    }  
}
```

mutex serializes
access to the shared
rc counter



READERS/WRITERS (3 OF 6)

- N processi accedono (cioè leggono o scrivono) ad alcuni dati condivisi.
- In qualsiasi momento: **R lettori o 1 scrittore ammessi**. Soluzione di base:

```
typedef int sema;  
sema mutex = 1;  
sema db = 1;  
int rc = 0;
```

```
void reader() {  
    while(TRUE) {  
        down(&mutex);  
        rc++;  
        if(rc==1) down(&db);  
        up(&mutex);  
        read_db();  
        down(&mutex);  
        rc--;  
        if(rc==0) up(&db);  
        up(&mutex);  
        use_data_read();  
    }  
}
```

```
void writer() {  
    while(TRUE) {  
        think_up_data();  
        down(&db);  
        write_db();  
        up(&db);  
    }  
}
```

db semaphore
controls RW access
to the shared db



READERS/WRITERS (4 OF 6)

- N processi accedono (cioè leggono o scrivono) ad alcuni dati condivisi.
- In qualsiasi momento: **R lettori o 1 scrittore ammessi**. Soluzione di base:

```
typedef int sema;  
sema mutex = 1;  
sema db = 1;  
int rc = 0;
```

```
void reader() {  
    while(TRUE) {  
        down(&mutex);  
        rc++;  
        if(rc==1) down(&db);  
        up(&mutex);  
        read_db();  
        down(&mutex);  
        rc--;  
        if(rc==0) up(&db);  
        up(&mutex);  
        use_data_read();  
    }  
}
```

```
void writer() {  
    while(TRUE) {  
        think_up_data();  
        down(&db);  
        write_db();  
        up(&db);  
    }  
}
```

db is a regular mutex
from the writers'
perspective



READERS/WRITERS (5 OF 6)

- N processi accedono (cioè leggono o scrivono) ad alcuni dati condivisi.
- In qualsiasi momento: **R lettori o 1 scrittore ammessi**. Soluzione di base:

```
typedef int sema;  
sema mutex = 1;  
sema db = 1;  
int rc = 0;
```

```
void reader(){  
    while(TRUE) {  
        down(&mutex);  
        rc++;  
        if(rc==1) down(&db);  
        up(&mutex);  
        read_db();  
        down(&mutex);  
        rc--;  
        if(rc==0) up(&db);  
        up(&mutex);  
        use_data_read();  
    }  
}
```

```
void writer(){  
    while(TRUE) {  
        think_up_data();  
        down(&db);  
        write_db();  
        up(&db);  
    }  
}
```

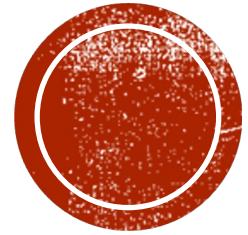
First / last reader
issues down / up
operations on db



READERS/WRITERS (6 OF 6)

- **Problema:** Se nuovi lettori arrivano mentre uno scrittore è in attesa, lo scrittore potrebbe mai ottenere l'accesso, portando a un blocco perpetuo.
- **Soluzione Proposta:**
 - Nuovi lettori vengono posti in coda dietro gli scrittori in attesa.
 - Gli scrittori ottengono accesso dopo i lettori già attivi.
- **Implicazioni:**
 - Questo metodo riduce la concorrenza.
 - Potenziale impatto sulle prestazioni.
- **Alternative:**
 - Esistono soluzioni che danno priorità agli scrittori.
 - Ogni strategia ha i suoi vantaggi e svantaggi.





LA MUTUA ESCLUSIONE: MUTEX E PTHREADS

INTRODUZIONE AI MUTEX

- Un "mutex" è **una versione esplicita e semplificata dei semafori**, usata per gestire la mutua esclusione di risorse o codice condiviso, quando **non bisogna contare** accessi o altri fenomeni.
- Può essere in due stati:
 - **locked** (bloccato)
 - **unlocked** (sbloccato)
- Un bit basta per rappresentarlo, ma spesso viene usato un intero (0 = unlocked, altri valori = locked).
- Due procedure principali: `mutex_lock` e `mutex_unlock`.



FUNZIONAMENTO DEI MUTEX

- Quando un thread vuole accedere a una regione critica, chiama `mutex_lock`.
- Se il mutex è `unlocked`, il thread può entrare; se è `locked`, il thread attende.
- Al termine dell'accesso, il thread chiama `mutex_unlock` per liberare la risorsa.
- Importante: **Non si utilizza il "busy waiting"**.
 - Se un thread non può acquisire un lock, chiama `thread_yield` per cedere la CPU ad un altro thread.



CONSIDERAZIONI AGGIUNTIVE

- I mutex possono essere implementati nello spazio utente con istruzioni come TSL o XCHG.
- Alcuni pacchetti di thread offrono mutex_trylock
 - tenta di acquisire il lock o restituisce un errore, senza bloccare.
- I mutex sono efficaci quando i thread operano in uno spazio degli indirizzi comune.
- La condivisione di memoria tra processi può essere gestita tramite il kernel o con l'aiuto di sistemi operativi che permettono la condivisione di parti dello spazio degli indirizzi.



PTHREAD E MUTEX

- **Pthread:** fornisce funzioni per sincronizzare i thread.
- **Mutex:**
 - variabile che può essere ***locked*** o ***unlocked***
 - protegge le regioni critiche.
- **Funzionamento:**
 - Thread tenta di bloccare (lock) un mutex per accedere alla regione critica.
 - Se mutex è unlocked, l'accesso è immediato e atomico.
 - Se locked, il thread attende.



MUTEXES IN PTHREADS (1)

Thread Call	Description
pthread_mutex_init	Inizializza un oggetto mutex per l'uso, configurando le risorse necessarie.
pthread_mutex_destroy	Distrugge un oggetto mutex, liberando le risorse associate. Deve essere chiamato solo se il mutex non è detenuto da alcun thread.
pthread_mutex_lock	Blocca un mutex, sospendendo l'esecuzione del thread chiamante se il mutex è già occupato da un altro thread.
pthread_mutex_trylock	Tenta di bloccare un mutex senza sospendere l'esecuzione. Se il mutex è già bloccato, la funzione restituisce immediatamente con un codice di errore specifico.
pthread_mutex_unlock	Sblocca un mutex, permettendo ad altri thread di acquisirlo. Deve essere chiamato solo dal thread che detiene il lock.

QUANDO USARE LOCK VS TRYLOCK

- **pthread_mutex_lock:** Quando l'accesso esclusivo è necessario e possiamo attendere se il lock non è disponibile.
 - Esempio: Protezione di risorse condivise in operazioni critiche, dove i thread possono aspettare in coda.
- **pthread_mutex_trylock:** Quando vogliamo solo tentare di acquisire il lock senza aspettare, proseguendo con altre operazioni se il lock è già in uso.
 - Esempio: Evitare deadlock in scenari complessi o verificare la disponibilità della risorsa senza bloccare il thread.

```
if (pthread_mutex_trylock(&mutex) == 0) {  
    // Se otteniamo il lock, eseguiamo le operazioni protette  
    printf("Ho ottenuto il lock e sto eseguendo operazioni.\n");  
    sleep(2); // Simuliamo qualche operazione lunga  
    printf("Ho terminato e rilascio il lock.\n");  
    // Rilasciamo il lock  
    pthread_mutex_unlock(&mutex);  
} else {  
    // Se il lock non è disponibile, continuiamo con altre operazioni  
    printf("Non sono riuscito a ottenere il lock, continuo con altre operazioni.\n");  
}
```



SEMAFORI O MUTEX?

- **Finalità:**

- **Mutex:** È utilizzato principalmente per **garantire l'esclusione mutua**. È destinato a proteggere l'accesso a una risorsa condivisa, garantendo che una sola thread possa accedervi alla volta.
- **Semaforo:** Può essere utilizzato per **controllare l'accesso a una risorsa condivisa**, ma è anche spesso usato per la sincronizzazione tra thread (vedi esempio produttore/consumatore).

- **Semantica:**

- **Mutex:** Di solito ha una semantica di "proprietà", il che significa che solo il thread che ha acquisito il mutex può rilasciarlo.
- **Semaforo:** Non ha una semantica di proprietà. Qualsiasi thread può aumentare o diminuire il conteggio del semaforo, indipendentemente da chi lo ha modificato l'ultima volta.

- **Casistica:**

- **Per l'esclusione mutua:** Un **mutex è generalmente preferibile**. È più semplice (di solito ha solo operazioni di lock e unlock) e spesso offre una semantica più rigorosa e un comportamento più prevedibile.
- **Per la sincronizzazione tra thread:** Un **semaforo può essere più adatto**, specialmente quando si tratta di coordinare tra diversi thread o di gestire risorse con un numero limitato di istanze disponibili.



MUTEX VS VARIABILI CONDIZIONALI

- **Mutex (`pthread_mutex`): Blocca l'accesso a una risorsa**
 - garantisce che solo un thread alla volta possa accedere a una risorsa condivisa, evitando conflitti di accesso.
 - Tuttavia, i mutex da soli non consentono di sincronizzare l'attesa su condizioni specifiche.
- **Possibili problemi:**
 - Se due thread vogliono accedere a una variabile condivisa (ad esempio, un contatore), il mutex può bloccare uno dei thread, garantendo che solo uno acceda alla variabile per volta.
 - Ma se uno dei thread ha bisogno di aspettare che il contatore raggiunga un certo valore (ad esempio, ≥ 10), il mutex da solo non è sufficiente per far attendere il thread in modo efficiente.



VARIABILI CONDIZIONALI (pthread_cond): ATTENDERE E NOTIFICARE EVENTI SPECIFICI

- Le variabili condizionali (pthread_cond) aggiungono la possibilità di **sincronizzare i thread basandosi su condizioni specifiche**, piuttosto che solo su accesso esclusivo.
- **Perché usare pthread_cond?**
 - Le variabili condizionali permettono ai thread di mettersi in attesa di un evento specifico e di essere notificati solo quando questo evento si verifica.
 - Ad esempio, in uno scenario produttore-consumatore, un thread produttore può notificare al consumatore che è stato prodotto un nuovo elemento e che ora è possibile consumarlo.



VARIABILI CONDIZIONALI (pthread_cond): COME FUNZIONA

- Un thread può attendere con `pthread_cond_wait` su una variabile condizionale finché una condizione specifica non è soddisfatta.
 - Durante l'attesa, ***il thread rilascia temporaneamente il mutex***, permettendo ad altri thread di accedere alla risorsa e di soddisfare la condizione.
 - Usando una variabile condizionale, il consumatore può sospendersi in attesa finché il buffer non è pieno, senza consumare risorse CPU.
 - Quando il produttore aggiunge un elemento al buffer, invia un segnale (`pthread_cond_signal`) che risveglia il consumatore. Questo sistema è molto più efficiente e coordinato.



MUTEXES IN PTHREADS (2)

Thread Call	Description
pthread_cond_init	Inizializza una variabile di condizione (pthread_cond_t) per consentire la sincronizzazione tra thread. La variabile di condizione viene associata a un mutex per coordinare l'accesso a risorse condivise.
pthread_cond_destroy	Distrugge una variabile di condizione, liberando le risorse associate. Deve essere chiamata solo quando non ci sono thread in attesa sulla condizione.
pthread_cond_wait	Blocca il thread chiamante in attesa della segnalazione di una condizione. Il thread deve detenere un lock sul mutex associato, che viene rilasciato automaticamente durante l'attesa e riacquisito al termine.
pthread_cond_signal	Risveglia uno dei thread in attesa sulla variabile di condizione. Se nessun thread è in attesa, la segnalazione viene persa. Questo è utile per notificare il cambiamento di stato di una risorsa condivisa.
pthread_cond_broadcast	Risveglia tutti i thread in attesa sulla variabile di condizione. Questo è utile quando un evento deve notificare più thread contemporaneamente, ad esempio, quando una risorsa diventa disponibile per tutti.



CONFRONTO NELLO SCENARIO PRODUTTORE/CONSUMATORE

- **Senza pthread_cond_wait:** Se hai solo un mutex, un thread (qui il consumatore) deve usare il busy-waiting per controllare continuamente una condizione, come in questo esempio:

```
pthread_mutex_lock(&mutex);
while (buffer == 0) { // Controllo continuo della condizione (inefficiente)
    pthread_mutex_unlock(&mutex);
    usleep(1000); // Ritardo per ridurre il busy-waiting (non ottimale)
    pthread_mutex_lock(&mutex);
}
consume(buffer);
pthread_mutex_unlock(&mutex);
```

- **Problema:** Il consumatore controlla continuamente `buffer == 0`, sprecando risorse CPU.



CONFRONTO NELLO SCENARIO PRODUTTORE/CONSUMATORE

- Con le variabili condizionali, il consumatore può sospendersi in modo efficiente:

```
pthread_mutex_lock(&mutex);
while (buffer == 0) {
    // Attesa passiva finché il buffer non è pieno
    pthread_cond_wait(&cond, &mutex);
}
consume(buffer);
pthread_mutex_unlock(&mutex);
```

- **Soluzione:** Il thread consumatore si sospende senza consumare CPU finché il produttore non segnala (`pthread_cond_signal`) che il buffer non è più vuoto.
- **IMPORTANTE:** vedere esempio `6.4_producer_consumer_pthread.c`



NOTA ESEMPIO:

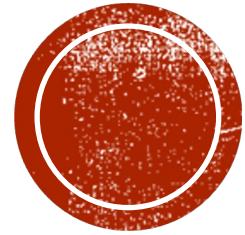
6.4_producer_consumer_pthread.c

...

```
for (i = 1; i <= MAX; i++) {  
    pthread_mutex_lock(&the_mutex);  
    while (buffer != 0) {  
        pthread_cond_wait(&condp, &the_mutex);  
    }  
    ...
```

- **Protezione della risorsa condivisa:** `pthread_mutex_lock` assicura che solo un thread alla volta possa accedere e modificare la risorsa condivisa (in questo caso, il buffer).
- **Attesa condizionale:** Quando un thread chiama `pthread_cond_wait`, due operazioni avvengono atomicamente. Il thread:
 - Rilascia il mutex.
 - Mette il thread in uno stato di attesa sulla variabile condizionale.
- Quindi, anche se il produttore ha acquisito il mutex, non lo detiene mentre è in attesa sulla variabile condizionale.
 - Questo permette al consumatore (o a un altro thread) di acquisire il mutex, fare le sue operazioni, e poi mandare un segnale alla variabile condizionale usando `pthread_cond_signal`.





LA MUTUA ESCLUSIONE: I MONITOR

MONITOR

- La comunicazione tra processi usando **mutex e semafori non è semplice** come potrebbe sembrare.
 - **Programmare con semafori richiede estrema attenzione:** piccoli errori possono causare comportamenti imprevisti come *race conditions* o *deadlock*.
- Brinch Hansen e Hoare proposero un concetto di sincronizzazione ad alto livello chiamato "**monitor**" per semplificare la scrittura di programmi.
- Un **monitor raggruppa procedure**, variabili e strutture dati. I processi possono chiamare le procedure di un monitor ma non possono accedere direttamente alle sue strutture dati interne.
 - **Solo un processo può essere attivo in un monitor in un dato momento**, garantendo la mutua esclusione.
 - **Il compilatore gestisce la mutua esclusione dei monitor**, riducendo la probabilità di errori da parte del programmatore.



MONITOR (2)

- Per gestire situazioni in cui i processi devono attendere, i **monitor utilizzano variabili condizionali e due operazioni su di esse**: wait e signal.
 - A differenza dei semafori, le variabili condizionali **non accumulano segnali**
 - se un segnale viene inviato e non c'è un processo in attesa, il segnale viene perso.
- **Linguaggi come Java** supportano i monitor, permettendo una sincronizzazione e mutua esclusione più sicura e semplice in contesti multithreading.
 - I metodi sono dichiarati synchronized in modo che solo un thread (in Java la programmazione concorrente è basata su thread) può accedervi



MONITOR (1)

```
monitor example
    integer i;
    condition c;

    procedure producer( );
        .
        .
        .

    end;

    procedure consumer( );
        .
        .
        .

    end;

end monitor;
```

Un esempio di monitor.



MONITOR: PRODUTTORE-CONSUMATORE

```
monitor ProdCons{
    condition full, empty;
    int count=0;
    void enter(int item) {
        if(count==N) wait(full);
        insert_item(item);
        count++;
        if(count==1) signal(empty);
    }
    void remove(int *item) {
        if(count==0) wait(empty);
        *item = remove_item();
        count--;
        if(count==N-1) signal(full);
    }
}
```

```
void producer() {
    int item;
    while(TRUE) {
        item = produce_item();
        ProdCons.enter(item);
    }
}

void consumer() {
    int item;
    while(TRUE) {
        ProdCons.remove(&item);
        consume_item(item);
    }
}
```



MONITOR: PRODUTTORE-CONSUMATORE

Access to enter
and remove is
serialized by the
monitor

```
monitor ProdCons{
    condition full, empty;
    int count=0;
    void enter(int item) {
        if(count==N) wait(full);
        insert_item(item);
        count++;
        if(count==1) signal(empty);
    }
    void remove(int *item) {
        if(count==0) wait(empty);
        *item = remove_item();
        count--;
        if(count==N-1) signal(full);
    }
}

void producer() {
    int item;
    while(TRUE) {
        item = produce_item();
        ProdCons.enter(item);
    }
}

void consumer() {
    int item;
    while(TRUE) {
        ProdCons.remove(&item);
        consume_item(item);
    }
}
```



MONITOR: PRODUTTORE-CONSUMATORE

wait suspends
caller on a condition
variable.
→ *Monitor state?*

```
monitor ProdCons{
    condition full, empty;
    int count=0;
    void enter(int item) {
        if(count==N) wait(full);
        insert_item(item);
        count++;
        if(count==1) signal(empty);
    }
    void remove(int *item) {
        if(count==0) wait(empty);
        *item = remove_item();
        count--;
        if(count==N-1) signal(full);
    }
}
```

```
void producer() {
    int item;
    while(TRUE) {
        item = produce_item();
        ProdCons.enter(item);
    }
}

void consumer() {
    int item;
    while(TRUE) {
        ProdCons.remove(&item);
        consume_item(item);
    }
}
```



MONITOR: PRODUTTORE-CONSUMATORE

signal wakes up
one waiter on a
condition variable.
→ Monitor state?
→ Lost wakeups?

```
monitor ProdCons{
    condition full, empty;
    int count=0;
    void enter(int item) {
        if(count==N) wait(full);
        insert_item(item);
        count++;
        if(count==1) signal(empty);
    }
    void remove(int *item) {
        if(count==0) wait(empty);
        *item = remove_item();
        count--;
        if(count==N-1) signal(full);
    }
}

void producer() {
    int item;
    while(TRUE) {
        item = produce_item();
        ProdCons.enter(item);
    }
}

void consumer() {
    int item;
    while(TRUE) {
        ProdCons.remove(&item);
        consume_item(item);
    }
}
```



DIFFERENZE TRA SLEEP/WAKEUP E WAIT/SIGNAL

▪ sleep/wakeup:

- meccanismi più primitivi utilizzati per mettere un processo/thread in attesa (sleep) e poi sveglierlo (wakeup).
- **Problema:** possono portare a delle *race condition* (visto prima...)
 - Immagina che il processo A voglia svegliare il processo B.
 - Se il processo A chiama wakeup per il processo B proprio mentre B sta per chiamare sleep...
 - ... B potrebbe finire per dormire indeterminatamente perché ha perso il segnale di sveglia.

• wait/signal (nei monitor):

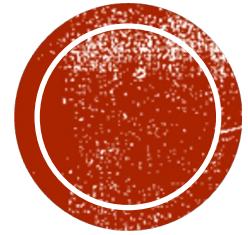
- Differenza cruciale: `wait` e `signal` sono protetti dalla mutua esclusione all'interno del monitor.
 - una volta che un thread/processo entra in una procedura del monitor, ha l'esclusività completa di quella procedura fino a quando non termina o chiama `wait`.
 - In JAVA la procedura ha il modificatore `synchronized`
- Se un thread/processo chiama `wait` all'interno di un monitor, può essere certo che non verrà interrotto (ad esempio, dallo scheduler) finché non ha terminato di posizionarsi in uno stato di attesa.
- Questo elimina la possibilità di perdere un segnale come poteva accadere con `sleep/wakeup`.



MONITOR E SEMAFORI

- I monitor sono costrutti di linguaggio, riconosciuti dal compilatore per garantire la mutua esclusione.
- Molti linguaggi, come C e Pascal, non hanno monitor o semafori.
 - Ma, si possono aggiungere semafori attraverso routine in assembly.
- I semafori sono pratici per risolvere la mutua esclusione in sistemi con memoria condivisa, ma non in sistemi distribuiti.
- Conclusione: I semafori sono a basso livello; i monitor sono limitati ai linguaggi che li supportano.





LA MUTUA ESCLUSIONE: SCAMBI DI MESSAGGI

SCAMBIO DI MESSAGGI

- Metodo di comunicazione tra processi usando due primitive: send e receive.
- Può essere utilizzato in diversi scenari, compresi sistemi distribuiti.
- **Problemi:**
 - Messaggi persi dalla rete.
 - Necessità di *acknowledgment* per confermare la ricezione.
 - Gestione dei messaggi duplicati usando numeri sequenziali.
 - Autenticazione e denominazione dei processi.
- Malgrado l'inaffidabilità, lo scambio di messaggi è cruciale nello studio delle reti.
 - ... *ne parleremo al secondo semestre*



PROBLEMA PRODUTTORE-CONSUMATORE E MESSAGGI

- Soluzione **senza memoria condivisa usando solo messaggi.**
- Utilizza un totale di N messaggi, simili ai N posti del buffer nella memoria condivisa.
 - Il consumatore invia al produttore N messaggi vuoti.
 - Il produttore prende un messaggio vuoto, lo riempie e lo invia.
- Numero totale di messaggi rimane costante, gestito dal sistema operativo.
- Questa soluzione garantisce efficienza e memoria predeterminata.



MESSAGE PASSING: PRODUCER-CONSUMER

<pre>#define N 100 void producer() { int item; message msg; while(TRUE) { item = produce_item(); receive(consumer, &msg); build_message(&msg, item); send(consumer, &msg); } }</pre>	<pre>void consumer() { int item, i; message msg; for(i=0; i<N; i++) send(producer, &msg); while(TRUE) { receive(producer, &msg); item = extract_item(); send(producer, &msg); consume_item(item); } }</pre>
--	--



MESSAGE PASSING: PRODUCER-CONSUMER

N messages buffered
by the system, initially
all empty
(producer's queue)

```
#define N 100

void producer(){
    int item;
    message msg;

    while(TRUE) {
        item = produce_item();
        receive(consumer, &msg);
        build_message(&msg, item);
        send(consumer, &msg);
    }
}

void consumer(){
    int item, i;
    message msg;
    for(i=0; i<N; i++)
        send(producer, &msg);
    while(TRUE) {
        receive(producer, &msg);
        item = extract_item();
        send(producer, &msg);
        consume_item(item);
    }
}
```



MESSAGE PASSING: PRODUCER-CONSUMER

Producer receives
an *empty* message and
“replaces” it with a *full*
message

```
#define N 100

void producer() {
    int item;
    message msg;

    while(TRUE) {
        item = produce_item();
        receive(consumer, &msg);
        build_message(&msg, item);
        send(consumer, &msg);
    }
}

void consumer() {
    int item, i;
    message msg;
    for(i=0; i<N; i++)
        send(producer, &msg);
    while(TRUE) {
        receive(producer, &msg);
        item = extract_item();
        send(producer, &msg);
        consume_item(item);
    }
}
```



MESSAGE PASSING: PRODUCER-CONSUMER

Consumer receives
a *full* message and
“replaces” it with an
empty message

```
#define N 100

void producer() {
    int item;
    message msg;

    while(TRUE) {
        item = produce_item();
        receive(consumer, &msg);
        build_message(&msg, item);
        send(consumer, &msg);
    }
}
```

```
void consumer() {
    int item, i;
    message msg;
    for(i=0; i<N; i++)
        send(producer, &msg);
    while(TRUE) {
        receive(producer, &msg);
        item = extract_item();
        send(producer, &msg);
        consume_item(item);
    }
}
```



MESSAGE PASSING: PRODUCER-CONSUMER

receive blocks
producer / consumer
when N messages are
full / empty

```
#define N 100

void producer() {
    int item;
    message msg;

    while(TRUE) {
        item = produce_item();
        receive(consumer, &msg);
        build_message(&msg, item);
        send(consumer, &msg);
    }
}

void consumer() {
    int item, i;
    message msg;
    for(i=0; i<N; i++)
        send(producer, &msg);
    while(TRUE) {
        receive(producer, &msg);
        item = extract_item();
        send(producer, &msg);
        consume_item(item);
    }
}
```



MECCANISMO DI SCAMBIO DI MESSAGGI E PROBLEMATICHE

- **Dinamica Produttore-Consumatore:**

- Se il produttore è più veloce, tutti i messaggi saranno pieni, costringendo il produttore ad attendere.
- Se il consumatore è più veloce, tutti i messaggi saranno vuoti, e il consumatore attende un messaggio pieno.

- **Indirizzamento dei Messaggi:**

- Ogni processo può avere un indirizzo univoco.
- Introduzione di "mailbox" come buffer per i messaggi.
- Send e receive fanno riferimento alle mailbox, non ai processi.

- **Applicazioni:**

- Scambio di messaggi usato in programmazione parallela.
- **MPI** (Message Passing Interface) è un esempio ben conosciuto usato in elaborazioni scientifiche.



MECCANISMI DI SINCRONIZZAZIONE E ARRIERE

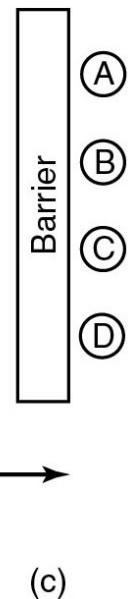
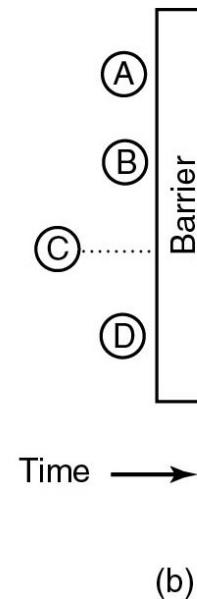
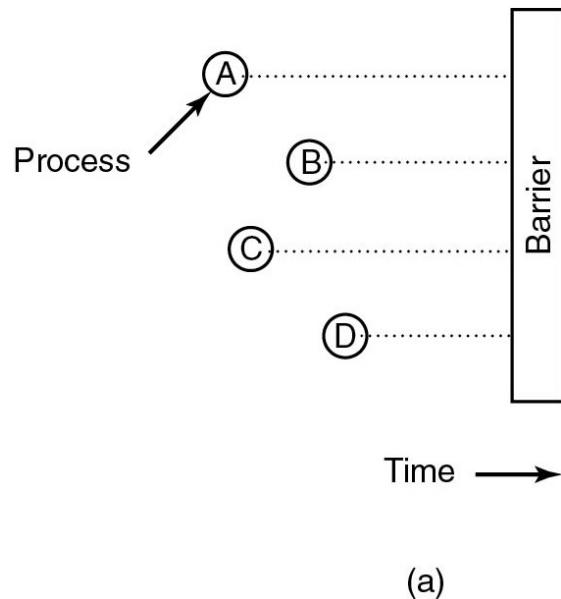
- Le **barriere** sono utilizzate per sincronizzare processi in fasi diverse.
 - Quando un processo raggiunge una barriera, attende fino a quando tutti gli altri processi la raggiungono.
 - Es.: in calcoli paralleli su matrici, i processi non possono avanzare a un'iterazione successiva finché tutti non hanno terminato l'iterazione attuale.

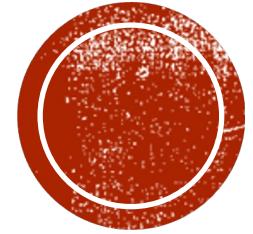


BARRIERE

- Le **barriere** sono utilizzate per **sincronizzare processi in fasi diverse**.
 - Quando un processo raggiunge una barriera, attende fino a quando tutti gli altri processi la raggiungono (ad esempio usando `pthread_join`).
 - **Esempio:** in calcoli paralleli su matrici, i processi non possono avanzare a un'iterazione successiva finché tutti non hanno terminato l'iterazione attuale.

- (a) Processi che si avvicinano a una barriera.
- (b) Tutti i processi tranne uno vengono bloccati alla barriera.
- (c) Quando l'ultimo processo arriva alla barriera, tutti vengono lasciati passare.





INVERSIONE DELLE PRIORITÀ

READ-COPY-UPDATE

PROBLEMA DEL MARS PATHFINDER:

- Il rover **Sojourner**, parte della missione **Mars Pathfinder**, usava un sistema operativo real-time per gestire diverse attività:
 - **Thread di alta priorità**: Analizzava dati scientifici importanti.
 - **Thread di bassa priorità**: Gestiva compiti meno importanti, come la registrazione di diagnostiche.
 - **Thread di priorità media**: Eseguiva altre operazioni di routine, come le comunicazioni.



NASA/JPL-Caltech



PROBLEMA DEL MARS PATHFINDER:

- Il problema si è verificato quando:
 - Un **thread di bassa priorità** stava usando una risorsa condivisa (bloccata da un mutex).
 - Un **thread di alta priorità** ha dovuto aspettare perché la risorsa era bloccata.
 - Nel frattempo, un **thread di priorità media** continuava a lavorare e impediva al thread di bassa priorità di liberare la risorsa.
- Questo ha causato un **blocco del thread di alta priorità**, che non poteva lavorare e quindi attivava un riavvio del sistema.



NASA/JPL-Caltech



IL PROBLEMA: PRIORITY INVERSION (INVERSIONE DELLE PRIORITÀ)

- L'inversione delle priorità si verifica quando:
 - Un thread di **alta priorità** aspetta una risorsa bloccata da un thread di **bassa priorità**.
 - Un thread di **priorità media**, che non ha nulla a che fare con la risorsa, impedisce al thread di bassa priorità di completare il suo lavoro.
- **Risultato:** il thread di alta priorità non riesce a lavorare, anche se teoricamente dovrebbe avere la precedenza su tutti.
- **Soluzione:** La NASA ha risolto il problema introducendo una tecnica chiamata **Priority Inheritance Protocol**. Con questo protocollo:
 - Quando un **thread di alta priorità** attende una risorsa bloccata da un **thread di bassa priorità**, il sistema “eleva” temporaneamente la priorità del thread di bassa priorità al livello del thread bloccante.
 - Questo permette al thread di bassa priorità di completare rapidamente il lavoro e rilasciare il mutex, consentendo al thread di alta priorità di procedere.
 - Una volta completato il lavoro, il thread di bassa priorità torna alla sua priorità originale.



UNA VERSIONE SEMPLIFICATA

- Immagina tre persone:
 - **Anna** (alta priorità) vuole usare una fotocopiatrice.
 - **Luca** (bassa priorità) sta usando la fotocopiatrice.
 - **Marco** (priorità media) sta facendo altro, ma sta continuamente interrompendo Luca con domande.
- **Risultato:**
 - Luca non riesce a finire il lavoro e liberare la fotocopiatrice.
 - Anna, che avrebbe la precedenza, deve aspettare inutilmente.
- Per risolvere il problema, è stato applicato il Priority Inheritance Protocol:
 - Quando Anna (alta priorità) aspetta Luca (bassa priorità), Luca **eredita temporaneamente la priorità di Anna**.
 - Questo gli permette di completare velocemente il suo lavoro, ignorando Marco (priorità media).
 - Una volta liberata la risorsa, Luca torna alla sua priorità normale.



READ-COPY-UPDATE

I migliori lock sono quelli che non si usano.

- **Obiettivo:** accessi concorrenti senza lock.
- **Problema:** possibile inconsistenza dei dati
 - (es. calcolo della media mentre si riordina un array).
- **Principio Base di Read-Copy-Update**
 - Aggiornare strutture dati consentendo letture simultanee senza incappare in versioni inconsistenti dei dati.
 - Lettori vedono: o la versione vecchia o la nuova, mai un mix delle due.

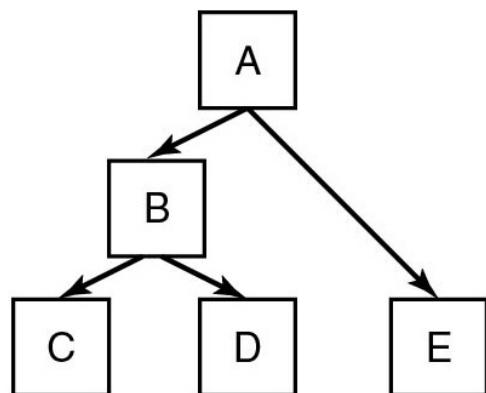


AVOIDING LOCKS: READ-COPY-UPDATE (2)

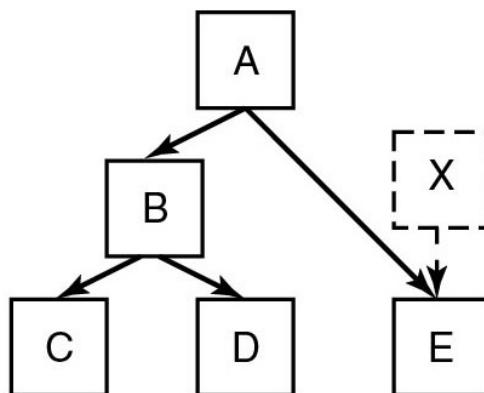
▪ Inserimento:

- Nodo X preparato e reso visibile in modo atomico.
- Nessuna versione non coerente letta.

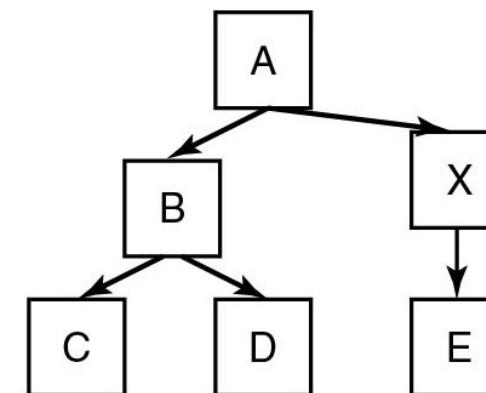
Adding a node:



(a) Original tree.



(b) Initialize node X and connect E to X. Any readers in A and E are not affected.



(c) When X is completely initialized, connect X to A. Readers currently in E will have read the old version, while readers in A will pick up the new version of the tree.

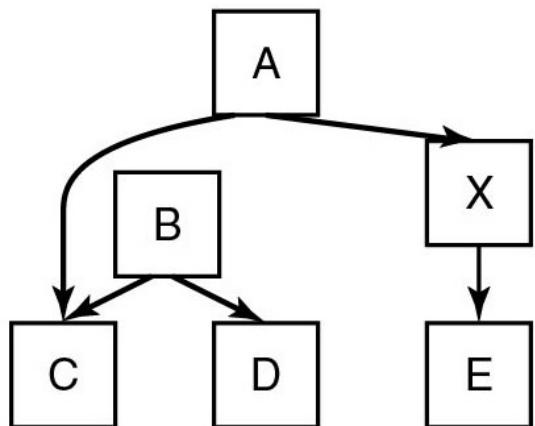


AVOIDING LOCKS: READ-COPY-UPDATE (3)

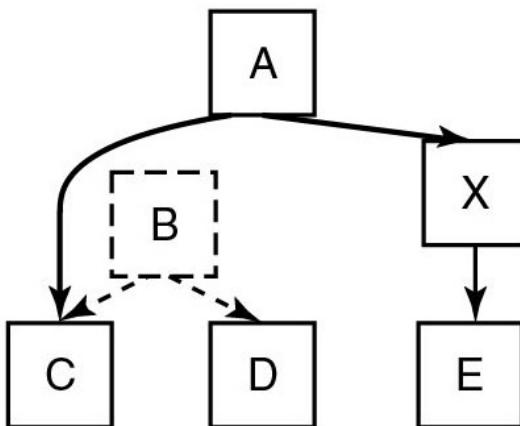
- **Rimozione:**

- Nodo B e D eliminati senza bisogno di lock.
- Lettori vedono o la nuova o la vecchia struttura, mai entrambe.

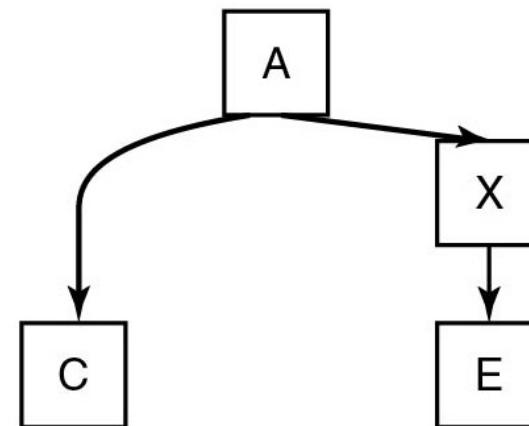
Removing nodes:



(d) Decouple B from A. Note that there may still be readers in B. All readers in B will see the old version of the tree, while all readers currently in A will see the new version.



(e) Wait until we are sure that all readers have left B and C. These nodes cannot be accessed any more.



(f) Now we can safely remove B and D



READ-COPY-UPDATE

▪ Problema e Soluzione:

- **Quando liberare B e D?** Finché ci sono lettori, non si possono liberare.
- **Operazione RCU:** determina il tempo massimo per trattenere un riferimento.
- **Grace Period:** tempo in cui ogni thread esce almeno una volta dalla sezione critica.
 - Aspetta un periodo \geq grace period prima di liberare la memoria.
 - I thread nella sezione critica non si bloccano né vanno in sleep, quindi si aspetta un cambio di contesto.

▪ Applicazione nell'Informatica:

- RCU non comune per processi utente.
- Diffuso nei kernel dei sistemi operativi.
- Kernel Linux: API RCU usata in molti sottosistemi
 - rete, file system, driver, gestione della memoria

