

# **SINCRONIZZAZIONE DEI PROCESSI NEI S.O.**

**Danilo Croce**

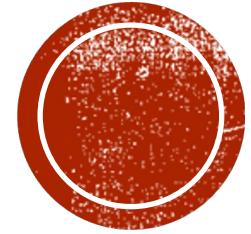
Ottobre 2023



# 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)

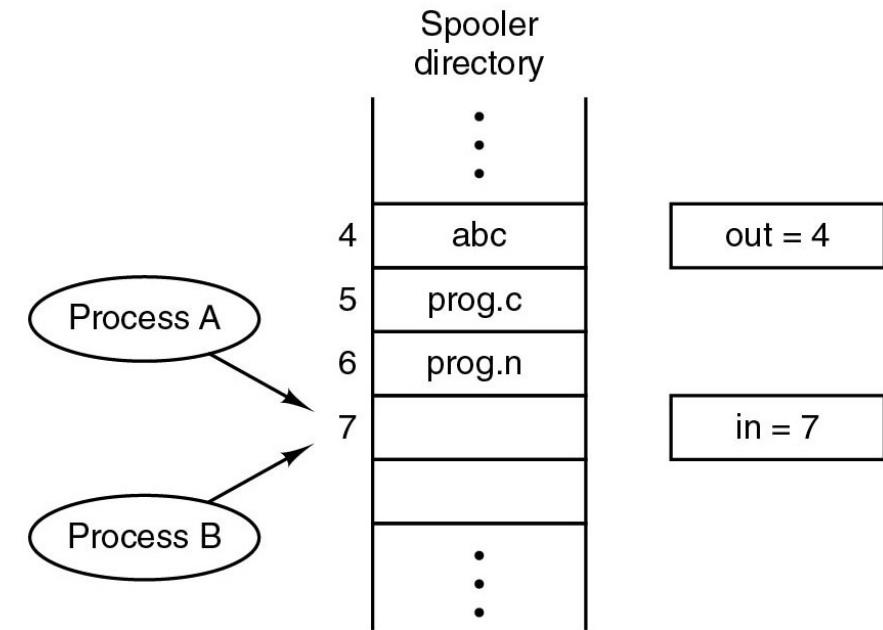
- I processi hanno bisogno di un modo per **comunicare**:
  - Condividere i dati durante l'esecuzione
- Nessuna condivisione esplicita tra processi:  
=> I dati devono essere scambiati normalmente tra i processi
- I processi hanno bisogno di un modo per **sincronizzarsi**:
  - Per tenere conto delle 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”:

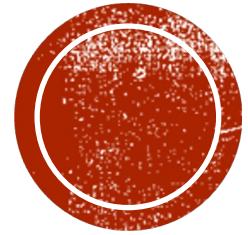
1. Due processi non possono trovarsi contemporaneamente all'interno delle rispettive regioni critiche.
2. Non si possono fare ipotesi sulla velocità o sul numero di CPU.
3. Nessun processo in esecuzione al di fuori della propria regione critica può bloccare altri processi.
4. Nessun processo deve aspettare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



# CRITICAL REGION (2)

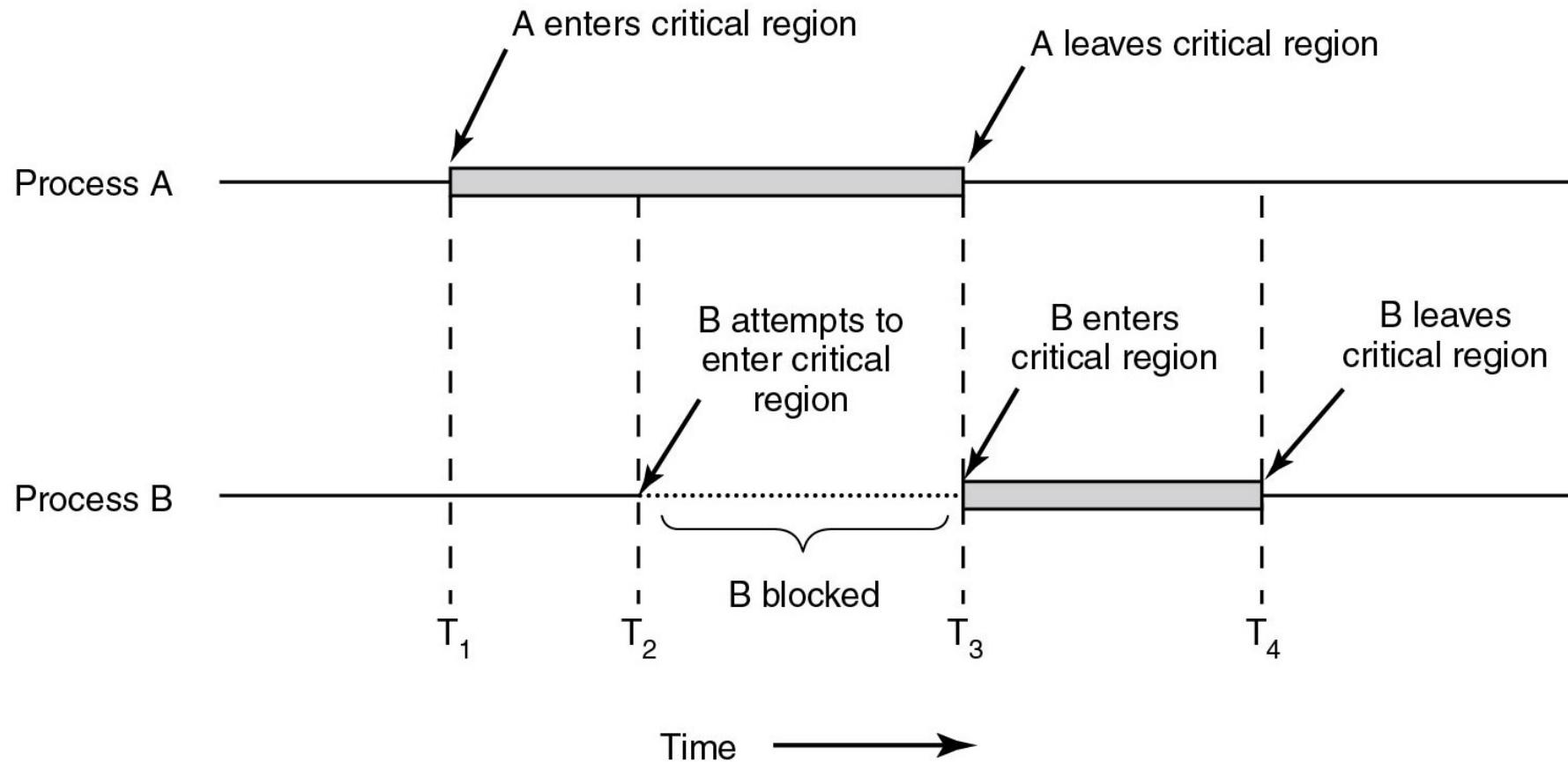
- (NON) soluzioni:
  - **Disabilitare gli interrupt:** impedisce semplicemente che la CPU possa essere riallocata. Funziona solo per sistemi a CPU singola
  - **Bloccare le variabili:** proteggere le regioni critiche con variabili 0/1. Le “corse” si verificano ora sulle variabili di blocco.





# 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 E' REALIZZATA LA MUTUA ESCLUSIONE NELLA CPU?

- E' possibile avvalersi di due istruzioni assembly
- **Istruzione TSL (Test and Set Lock):**
  - Presente in computer con più processori.
  - Legge il contenuto della memoria "lock", salva un valore non zero, e blocca altre CPU da accesso alla memoria.
  - Purtroppo anche disabilitando gli interrupt su un processore non c'è garanzia che un processo «faccia danni» da un'altra CPU e quindi, blocchiamo tutti fino al termine dell'esecuzione di TSL
- **Istruzione XCHG:**
  - Scambia i contenuti di due posizioni atomicamente.
  - Usata in tutte le CPU x86 Intel per sincronizzazione di basso livello.



# FUNZIONAMENTO E USO DEL TSL

- Quando `lock` è 0:
  - Un processo può impostare `lock` a 1 con TSL e accedere alla memoria condivisa.
  - Al termine, il processo resetta `lock` a 0.
- **Metodo per gestire Regioni Critiche:**
  - Processi chiamano `enter_region` prima di entrare nella regione critica e `leave_region` dopo.
  - Se chiamati correttamente, garantisce la mutua esclusione.
  - Se usati in modo errato, la mutua esclusione fallisce.



# UTILIZZO DELL'ISTRUZIONE TLS:

enter\_region:

```
TSL REGISTER,LOCK      | copia il lock nel registro e lo imposta a 1  
CMP REGISTER,#0        | il lock era zero?  
JNE enter_region       | se non era zero, il lock era stato impostato, per cui  
                        | ri-esegui il ciclo  
RET                    | torna al chiamante; si è entrati nella regione critica
```

leave\_region:

```
MOVE LOCK,#0           | memorizza 0 in lock  
RET                    | torna al chiamante
```



# UTILIZZO DELL'ISTRUZIONE XCHG:

```
enter_region:  
    MOVE REGISTER,#1           | mette un 1 nel registro  
    XCHG REGISTER,LOCK         | scambia il contenuto del registro e della variabile  
                                lock  
    CMP REGISTER,#0            | il lock era zero?  
    JNE enter_region          | se non era zero il lock era stato impostato, per  
                                cui ri-esegui il ciclo  
    RET                        | torna al chiamante;  
                                si è entrati nella regione critica  
  
leave_region:  
    MOVE LOCK,#0               | memorizza 0 in lock  
    RET                        | torna al chiamante
```

Entrambe le istruzioni sono essenziali per garantire la sicurezza nelle operazioni condivise tra più processori.



# ISTRUZIONE TSL E XCHG

## ■ Istruzione TSL (Test and Set Lock)

- Legge il contenuto della parola di memoria lock in un registro e salva un valore non zero in lock.
- Operazione è indivisibile: nessun altro processore può accedere finché TSL non è completata.
- Bloccare il bus della memoria impedisce altri accessi alla memoria da altre CPU.

## ■ Istruzione XCHG (Exchange)

- Scambia i contenuti di due posizioni (es. un registro e una parola di memoria) atomicamente.
- Utilizzata dalle CPU x86 Intel per sincronizzazione di basso livello.



# 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:**

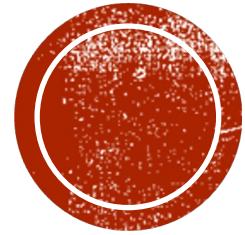
- Il consumatore potrebbe essere risvegliato un attimo prima di andare a dormire
- Bit di Attesa può essere aggiunto come rimedio per segnali di risveglio persi.

- **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 (1/5)

```
#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);
    }
}
```



# SEMAFORI: PRODUTTORE-CONSUMATORE (2/5)

```
#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 (3/5)

```
#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 (4/5)

```
#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);
    }
}
```

full semaphore  
blocks the consumer  
when the buffer is  
*empty*

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



# SEMAFORI: PRODUTTORE-CONSUMATORE (5/5)

```
#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);
    }
}
```

3 semaphores used  
in our solution.  
→ Can we use 2?  
→ Lost wakeups?



# 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);  
    }  
}
```

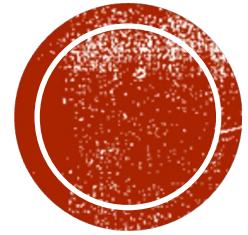
First / last reader  
issues down / up  
operations on db



# 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.



# PARALLELISMO E PRESTAZIONI

- L'efficienza nella sincronizzazione diventa cruciale con l'aumento del parallelismo.
- Spin lock e mutex con busy waiting: efficaci per attese brevi, ma sprecano CPU per attese lunghe.
- Passaggio al kernel per bloccare processi è oneroso se le contese sono poche.
- Soluzione: **Futex** (Fast User Space Mutex) – combina il meglio di entrambi gli approcci.



# COS'È UN FUTEX?

- Caratteristica di Linux per **implementare lock elementari evitando il kernel finché non è necessario**.
  - ☺ Migliora le prestazioni riducendo il passaggio al kernel.
  - ☹ Non standard (serve `#include <linux/futex.h>`)
- **Due parti:**
  - servizio kernel
  - libreria utente
- **Operazione:**
  - Variabile condivisa nello spazio utente usata come lock.
  - Il passaggio al kernel avviene solo quando un *thread* è bloccato da un altro.
- Quando un lock è rilasciato, il kernel può essere chiamato per svegliare altri processi in attesa.



# 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
<code>pthread_mutex_init</code>	Create a mutex
<code>pthread_mutex_destroy</code>	Destroy an existing mutex
<code>pthread_mutex_lock</code>	Acquire a lock or block
<code>pthread_mutex_trylock</code>	Acquire a lock or fail
<code>pthread_mutex_unlock</code>	Release a lock

- Some of the Pthreads calls relating to mutexes.



# MUTEXES IN PTHREADS (2)

Thread Call	Description
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Block waiting for a signal
pthread_cond_signal	Signal another thread and wake it up
pthread_cond_broadcast	Signal multiple threads and wake all of them

- Some of the Pthreads calls relating to condition variables.



# 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.



# NOTA ESEMPIO:

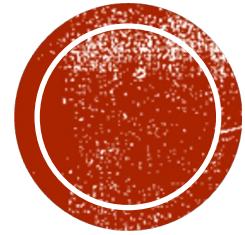
## 6.4\_producer\_consumer\_pthread.c

...

```
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

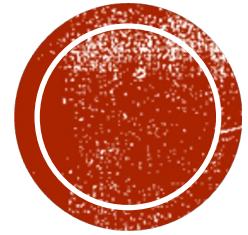
- **sleep/wakeup:**
  - meccanismi più primitivi utilizzati per mettere un processo/thread in attesa (sleep) e poi sveglierlo (wakeup).
  - **Problema:** possono portare a delle *race condition*).
    - Immagina che il processo A voglia svegliare il processo B.
    - Se il processo A chiama wakeup per 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, &amp;msg);         build_message(&amp;msg, item);         send(consumer, &amp;msg);     } }</pre>	<pre>void consumer() {     int item, i;     message msg;     for(i=0; i&lt;N; i++)         send(producer, &amp;msg);     while(TRUE) {         receive(producer, &amp;msg);         item = extract_item();         send(producer, &amp;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

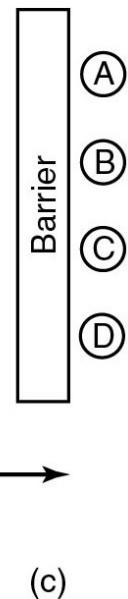
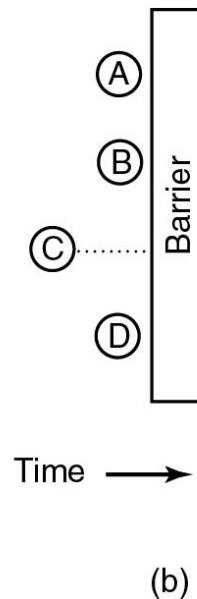
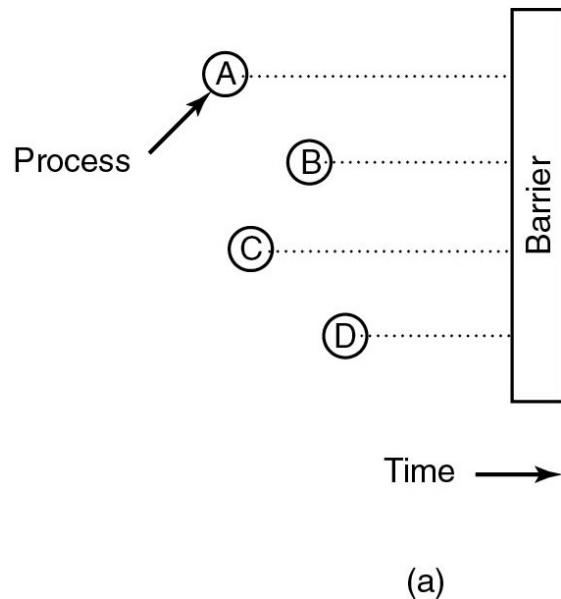
- Le **barriere** sono utilizzate per sincronizzare processi in fasi diverse.
  - Quando un processo raggiunge una barriera, attende fino a quando tutti gli altri processi la raggiungono.
  - Es.: in calcoli paralleli su matrici, i processi non possono avanzare a un'iterazione successiva finché tutti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

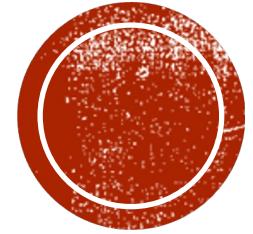


# BARRIERE

- 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.
  - 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:

- **Veicolo robotico su Marte.**
- **3 thread:**
  - Dati meteorologici (bassa priorità),
  - Gestione bus informativo (alta priorità),
  - Comunicazioni (media priorità).
- **Bus informativo:**
  - memoria condivisa utilizzata per passare informazioni.
- **Mutex:**
  - utilizzato per controllare l'accesso al bus informativo.



NASA/JPL-Caltech



# CHE COSA È SUCCESSO?

- Il thread dei dati meteorologici (bassa priorità) acquisisce il mutex.
- Il thread di comunicazione (media priorità) prenota il thread dei dati meteorologici.
- Il thread di gestione bus (alta priorità) cerca di eseguire, ma si blocca perché non può acquisire il mutex.
- **Risultato: Inversione delle priorità**
  - il thread di comunicazione funziona come se avesse alta priorità, causando interruzioni nelle trasmissioni.



NASA/JPL-Caltech



# POSSIBILI SOLUZIONI ALL'INVERSIONE DELLE PRIORITÀ

## ▪ **Disattivazione degli interrupt**

- Semplice ma rischioso;
- gli interrupt potrebbero non essere riattivati.

## ▪ **Priority Ceiling (Limite della Priorità)**

- Assegnare **una priorità al mutex**.
- La priorità viene assegnata al processo che detiene il mutex.
- Fintanto che nessun processo che deve acquisire il mutex ha una priorità superiore al limite, l'inversione non è più possibile.

## ▪ **Priority Inheritance (Ereditarietà della Priorità):**

- Task a bassa priorità che detiene il mutex eredita temporaneamente la priorità del task ad alta priorità.
- Adottato per risolvere i problemi del Mars Pathfinder.

## ▪ **Random Boosting (Potenziamento Casuale):**

- Aumenta la priorità di thread casuali che detengono un mutex.



# 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.

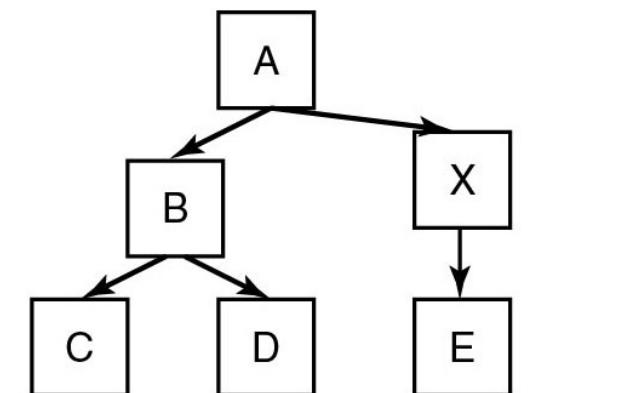
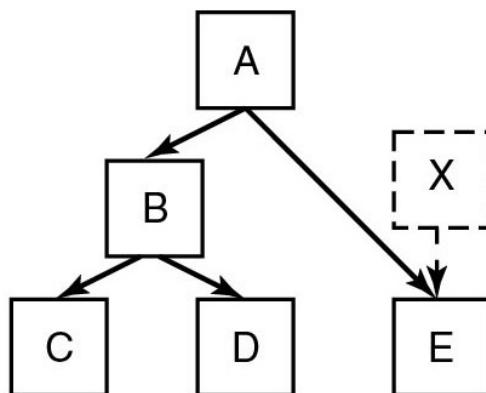
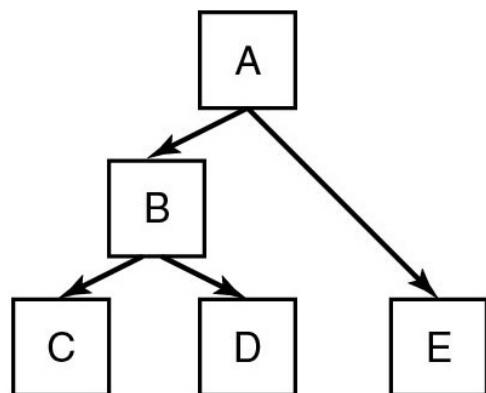


# AVOIDING LOCKS: READ-COPY-UPDATE (2)

## ▪ Inserimento:

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

Adding a node:

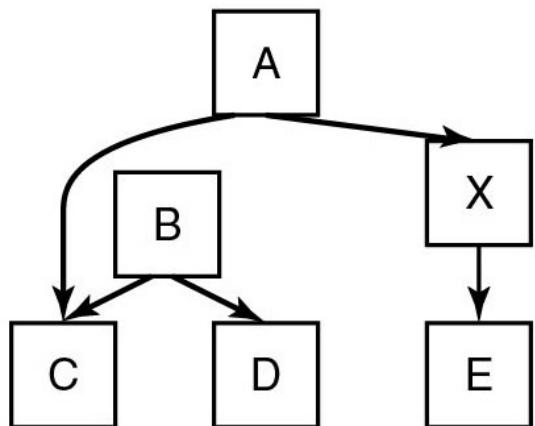


# AVOIDING LOCKS: READ-COPY-UPDATE (3)

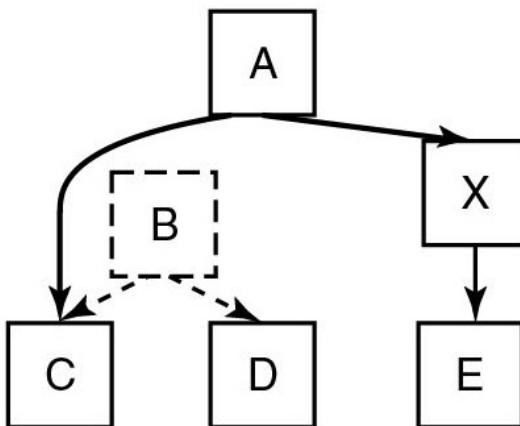
- **Rimozione:**

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

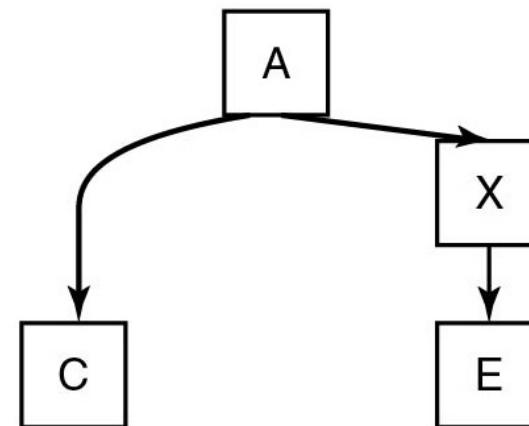
**Removing nodes:**



(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

