

Sistemi Operativi (M-Z)

C.d.L. in Informatica
(Laurea Triennale)
A.A. 2020-2021

Prof. Mario F. Pavone

Dipartimento di Matematica e Informatica
Università degli Studi di Catania
mario.pavone@unict.it
mpavone@dmi.unict.it



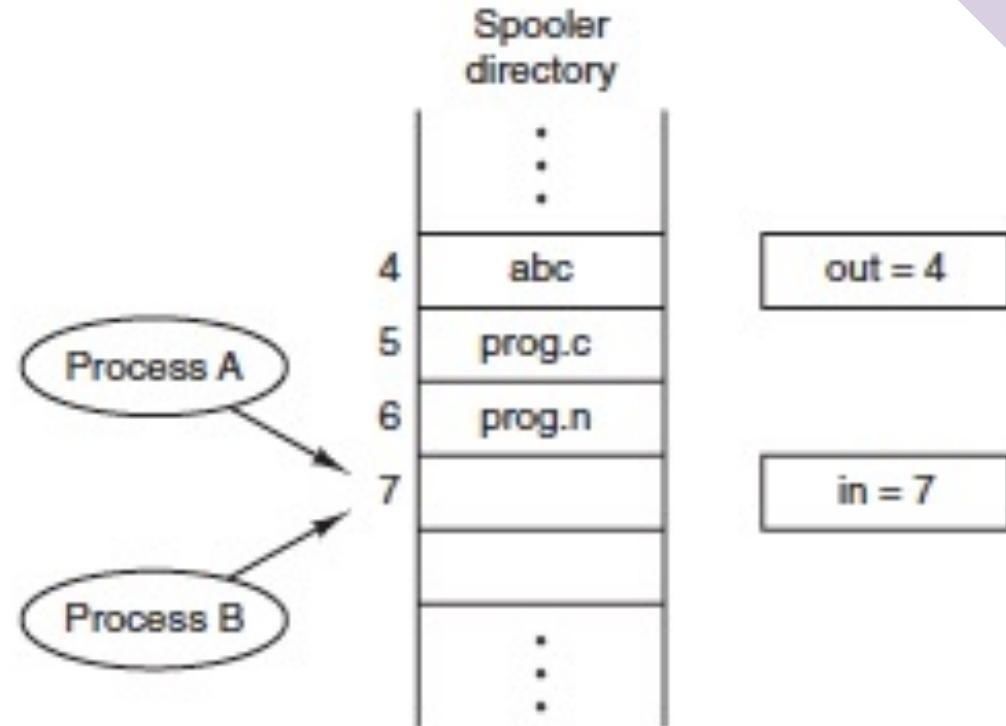
Comunicazione
fra Processi (IPC)

Comunicazione fra processi

- Spesso i processi hanno bisogno di **cooperare**:
 - collegamento I/O tra processi (**pipe**);
 - **InterProcess Communication (IPC)**;
 - possibili **problematiche**:
 - come scambiarsi i dati;
 - accavallamento delle operazioni su dati comuni;
 - coordinamento tra le operazioni (o sincronizzazione).
- **Corse critiche** (race conditions);
 - esempio: versamenti su conto-corrente;
 - corse critiche nel codice del **kernel**;
 - kernel **preemptive** vs. **non-preemptive**;
 - soluzione: **mutua esclusione** nell'accesso ai dati condivisi.

Race Conditions

- **Spool di Stampa:**
demone di stampa &
directory di spool

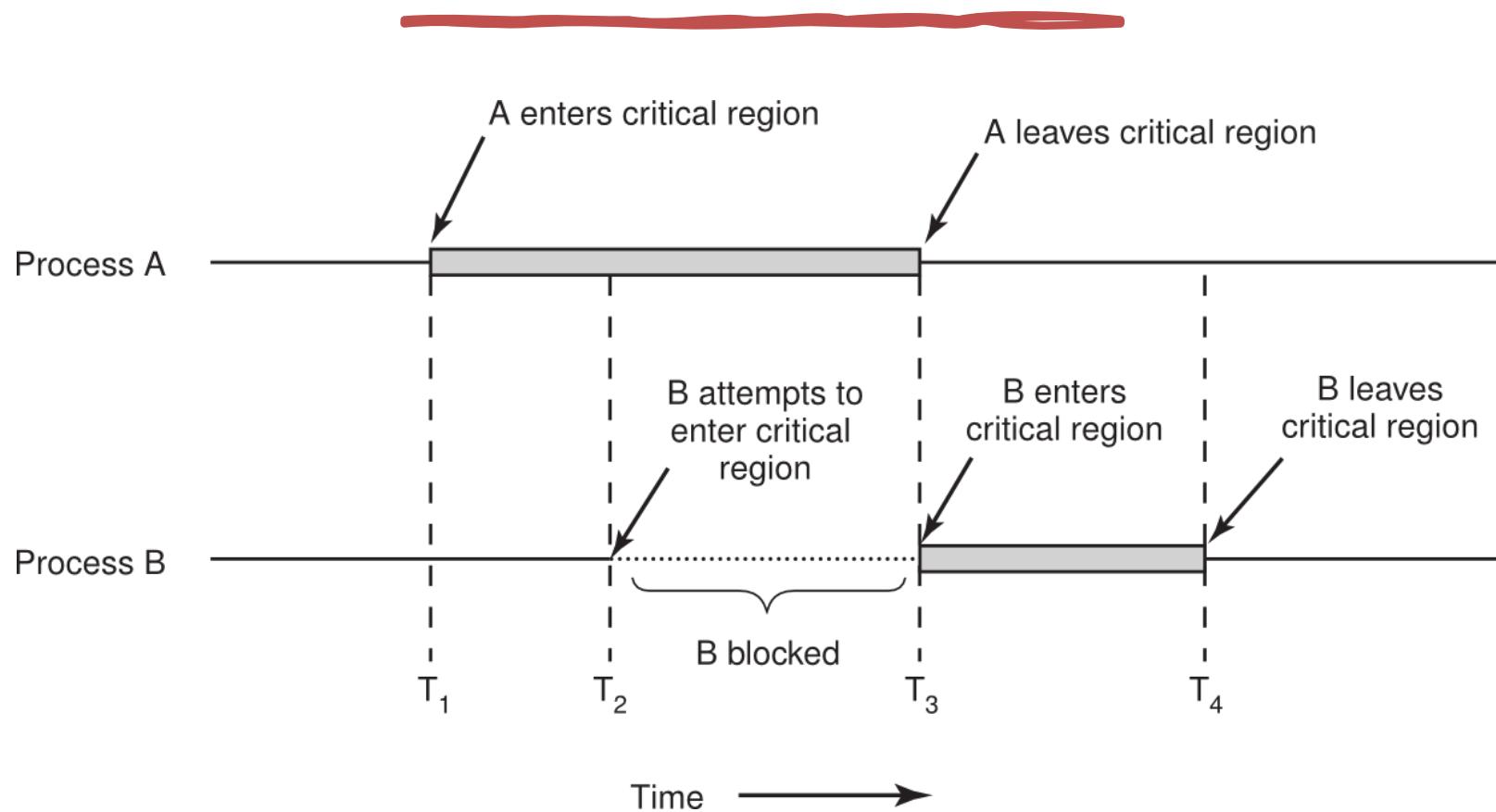


Evitare che più processi **accedino a dati condivisi contemporaneamente**

Regioni Critiche

- **Mutua Esclusione:** garantire che il processo *B* utilizzi le risorse condivise **solo dopo** che il processo *A* termini il suo completo utilizzo
- **Regione/Sezione Critica:** parte di programma in cui si accede alla memoria condivisa

Sezioni critiche



Sezioni critiche

- Astrazione del problema: **sezioni critiche** e **sezioni non critiche**.
- **Quattro condizioni** per avere una buona soluzione:
 1. **mutua esclusione** nell'accesso alle rispettive sezioni critiche;
 2. **nessuna assunzione** sulla velocità di esecuzione o sul numero di CPU;
 3. nessun processo fuori dalla propria sezione critica può **bloccare un altro processo**;
 4. nessun processo dovrebbe **restare all'infinito in attesa** di entrare nella propria sezione critica.

Come realizzare la mutua esclusione

- **Disabilitare gli interrupt**
 - scelta non saggia => blocco del sistema
- **Variabili di lock**
 - soluzione software
 - Variabile codivisa:
 - 0 -> nessun processo è nella regione critica
 - 1 -> qualche processo è nella sua regione critica

Come realizzare la mutua esclusione

- **Alternanza stretta:**

```
int N=2
int turn

function enter_region(int process)
    while (turn != process) do
        nothing

function leave_region(int process)
    turn = 1 - process
```

- può essere facilmente generalizzato al caso N;
- fa **busy waiting** (si parla di **spin lock**);
- implica **rigidi turni** tra le parti (viola condizione 3).

Soluzione di Peterson

```
int N=2
int turn
int interested[N]

function enter_region(int process)
    other = 1 - process
    interested[process] = true
    turn = process
    while (interested[other] = true and turn = process) do
        nothing

function leave_region(int process)
    interested[process] = false
```

- ancora **busy waiting**;
- può essere generalizzato al caso N;
- può avere problemi sui moderni multi-processori a causa del riordino degli accessi alla memoria centrale.

Istruzioni TSL e XCHG

- Molte architetture (soprattutto multi-processore) offrono specifiche istruzioni:
 - **TSL (Test and Set Lock);**
 - uso: **TSL registro, lock**
 - operazione atomica e blocca il bus di memoria;

```
enter_region:  
    TSL REGISTER,LOCK  
    CMP REGISTER,#0  
    JNE enter_region  
    RET
```

```
leave_region:  
    MOVE LOCK,#0  
    RET
```

- **XCHG (eXCHanGe);**
 - disponibile in tutte le CPU Intel X86;
- ancora **busy waiting**.

Istruzioni TSL e XCHG

- **TSL (Test-and-Swap Lock):** queste istruzioni (presenti solo nei processori Intel x86) offrono una sincronizzazione semplice e veloce.
- **enter_region:**

```
MOVE REGISTER,#1  
XCHG REGISTER,LOCK  
CMP REGISTER,#0  
JNE enter_region  
RET
```
- **leave_region:**

```
TSL REGISTER,LOCK  
CMP REGISTER,#0  
JNE enter_region  
RET
```

```
MOVE LOCK,#0
```
- **XCHG (eXCHanGe);**
 - disponibile in tutte le CPU Intel x86;
- ancora **busy waiting**.

Sleep e wakeup

- Tutte le soluzioni viste fino ad ora fanno **spin lock**;
 - **problema dell'inversione di priorità.**
- **Soluzione:** dare la possibilità al processo di bloccarsi in modo passivo (rimozione dai processi pronti);
 - primitive: **sleep** e **wakeup**.
- **Problema del produttore-consumatore (buffer limitato – N):**
 - variabile condivisa count inizialmente posta a 0;

```
function producer()
    while (true) do
        item = produce_item()
        if (count = N) sleep()
        insert_item(item)
        count = count + 1
        if (count = 1)
            wakeup(consumer)
```

```
function consumer()
    while (true) do
        if (count = 0) sleep()
        item = remove_item()
        count = count - 1
        if (count = N - 1)
            wakeup(producer)
        consume_item(item)
```

- questa soluzione **non funziona bene**: usiamo un **bit di attesa** **wakeup**.

Semafori

- Generalizziamo il concetto di sleep e wakeup – **semaforo**:
 - variabile intera condivisa **S**;
 - operazioni: **down** e **up** (dette anche **wait** e **signal**);
 - **operazioni atomiche**:
 - gruppo di operazioni eseguite insieme senza interruzioni
 - disabilitazione interrupt o spin lock TSL/XCHG;
 - tipicamente implementato **senza busy waiting** con una **lista di processi bloccati**.

Produttore-consumatore con i semafori

- Utilizzo di 3 semafori:
 - **full, empty & mutex**

```
function producer()
    while (true) do
        item = produce_item()
        down(empty)
        down(mutex)
        insert_item(item)
        up(mutex)
        up(full)
```

```
int N=100
semaphore mutex = 1
semaphore empty = N
semaphore full = 0
```

```
function consumer()
    while (true) do
        down(full)
        down(mutex)
        item = remove_item()
        up(mutex)
        up(empty)
        consume_item(item)
```

- **Mutex**: mutua esclusione
- L'ordine delle operazioni sui semafori è fondamentale...

- Diverso utilizzo dei Semafori:
 - **semaforo mutex**: mutua esclusione;
 - **semafori full & empty**: sincronizzazione.

```
function producer()
  while (true) do
    item = produce_item()
    down(empty)
    down(mutex)
    insert_item(item)
    up(mutex)
    up(full)
```

```
function consumer()
  while (true) do
    down(full)
    down(mutex)
    item = remove_item()
    up(mutex)
    up(empty)
    consume_item(item)
```

Semafori



- Semaforo per la **gestione** della Mutua Esclusione
 - particolarmente valido in thread spazio utente
- Due stati: **Locked & unlocked**

```
mutex_lock:
    TSL REGISTER,MUTEX
    CMP REGISTER,#0
    JZE ok
    CALL thread_yield
    JMP mutex_lock
ok:RET
```

```
mutex_unlock:
    MOVE MUXTEX,#0
    RET
```

Mutex e thread utente

- simili a **enter_region/leave_region**
ma:
 - senza **spin lock**;
 - il **busy waiting** sarebbe problematico con i thread utente:
 - no **clock** per i thread

Mutex e thread utente

- Qual'è la differenza tra `enter_region` & `mutex_lock`

```
enter_region:
```

```
    TSL REGISTER,LOCK  
    CMP REGISTER,#0  
    JNE enter_region  
    RET
```

```
mutex_lock:
```

```
    TSL REGISTER,MUTEX  
    CMP REGISTER,#0  
    JZE ok  
    CALL thread_yield  
    JMP mutex_lock
```

```
ok:RET
```



Cede la CPU ad un altro thread

- `Thread_yield` è molto veloce (chiamata scheduler thread spazio utente)

Futex

- Osservazione: i mutex in user-space sono molto efficienti ma lo spin lock può essere lungo!
 - Spin lock: spreco cicli CPU
 - Blocco del processo e gestione al Kernel
- → **futex** = *fast user space mutex* (Linux)
- due componenti:
 - **servizio kernel**
 - coda di thread bloccati
 - **libreria utente**
 - variabile di **lock**
 - contesa in modalità utente (tipo con TSL/XCHG)
 - richiamo kernel solo in caso di bloccaggio

Produttore/Consumatore con i Semafori

- Attenzione nell'uso dei semafori

```
function producer()
    while (true) do
        item = produce_item()
        down(empty)
        down(mutex)
        insert_item(item)
        up(mutex)
        up(full)
```

```
function producer()
    while (true) do
        item = produce_item()
        down(mutex)
        down(empty)
        insert_item(item)
        up(mutex)
        up(full)
```

- I processi potrebbero rimanere bloccati per sempre (deadlock)

I Monitor

- primitiva di sincronizzazione
- costrutto ad alto-livello disponibile su alcuni linguaggi
- **tipo astratto di dato**: raccolta di variabili, strutture dati & procedure
 - tipo di dati astratto (ADT – Abstract Data Type): dati privati con metodi pubblici
- processi: chiamare procedure ma no accesso strutture dati
- garanzia **mutua esclusione**: 1 processo attivo alla volta
 - organizzazione al compilatore
 - convertire regioni critiche in procedure monitor
- vincolo di **accesso ai dati** (interni ed esterni)
- **come bloccare un processo che non può proseguire?**

I Monitor

- Meccanismo di sincronizzazione:
variabili condizione
 - operazioni **wait** e **signal**
 - **wait**: blocca processo chiamante
 - **signal**: sveglia il processo in sleep
 - variabili condizione: non sono contatori => non accumulano segnali
 - cosa accade dopo signal? (due processi attivi nel monitor)
 - Hoare: eseguire il processo svegliato;
 - Brinch-Hansen: signal come ultima istruzione di una procedura monitor
 - continuare l'esecuzione del segnalatore

I Monitor

- . Processo **P** chiama signal(x) & processo **Q** è in sleep su x
 - . Nota: entrambi i processi posso continurare l'esecuzione
- . la signal può avere **diverse semantiche**:
 - . **monitor Hoare** (teorico): **signal & wait**;
 - . **monitor Mesa** (Java): **signal & continue**;
 - . **compromesso** (concurrent Pascal): **signal & return**.
- . Ex. JAVA: **synchronized** keyword
- . **Meno errori in programmazione parallela** rispetto ai semafori
- . Svantaggio:
 - . Molti linguaggio non hanno i monitor
 - . semafori => più semplice da aggiungere
 - . Sistema distribuito con più CPU aventi propria memoria privata

Produttore-consumatore con i monitor

```
monitor pc_monitor
    condition full, empty;
    integer count = 0;

function insert(item)
    if count = N then wait(full);
    insert_item(item);
    count = count + 1;
    if count = 1 then signal(empty)
```

```
function producer()
    while (true) do
        item = produce_item()
        pc_monitor.insert(item)
```

```
function remove()
    if count = 0 then
        wait(empty);
    remove = remove_item();
    count = count - 1;
    if count = N-1 then signal(full)
```

```
function consumer()
    while (true) do
        item = pc_monitor.remove()
        consume_item(item)
```

Scambio messaggi tra processi

- Primitive più ad alto livello:
 - `send(destinazione, messaggio)`
 - `receive(sorgente, messaggio)`
 - bloccante per il chiamante (o può restituire un errore);
 - estendibile al caso di più macchine (es., libreria MPI);
 - metodi di indirizzamento: **diretto** o tramite **mailbox**;
 - assumendo realisticamente l'**esistenza di un buffer** per i messaggi:
 - capienza finita N
 - la mailbox contiene i messaggi spediti ma non ancora accettati
 - la send può essere bloccante
- Strategia **Rendezvous**: eliminazione uso buffer
 - meno flessibile

Produttore- Consumatore con Scambio di Messaggi



```
#define N 100                                     /* number of slots in the buffer */

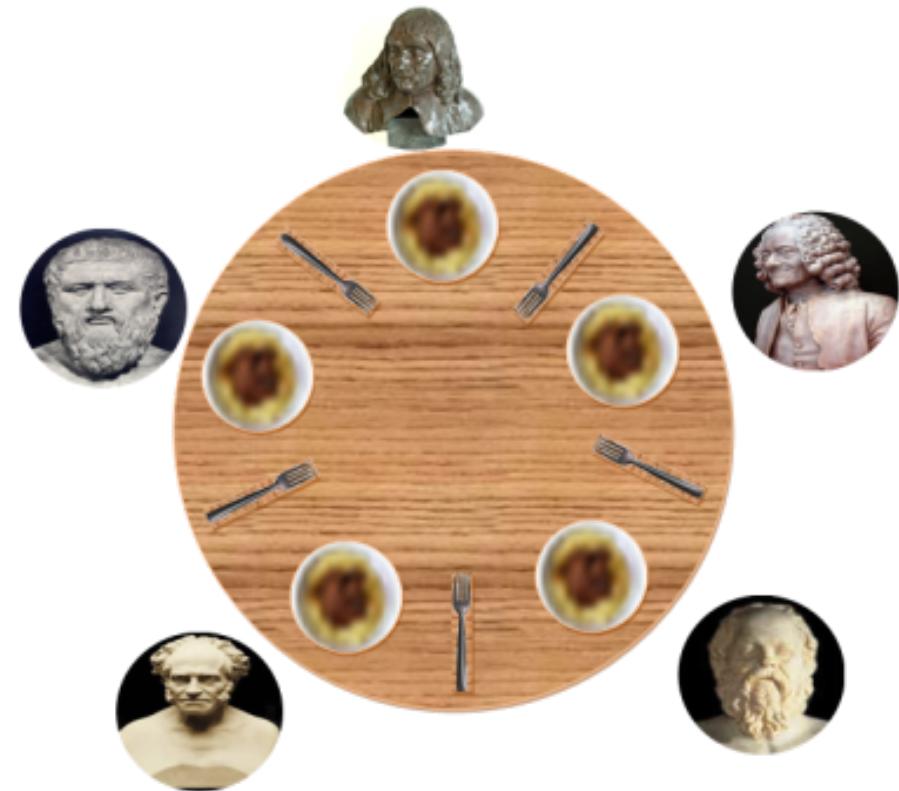
void producer(void)
{
    int item;
    message m;                                    /* message buffer */

    while (TRUE) {
        item = produce_item();                    /* generate something to put in buffer */
        receive(consumer, &m);                  /* wait for an empty to arrive */
        build_message(&m, item);                /* construct a message to send */
        send(consumer, &m);                     /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);                 /* get message containing item */
        item = extract_item(&m);              /* extract item from message */
        send(producer, &m);                   /* send back empty reply */
        consume_item(item);                  /* do something with the item */
    }
}
```

Problema dei 5 filosofi a cena

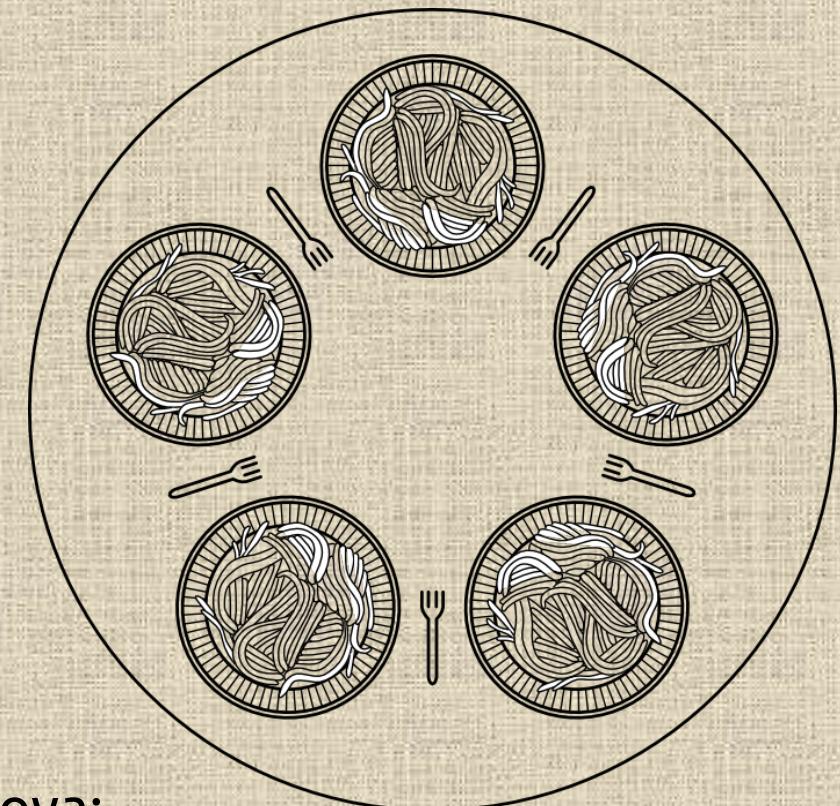


- Problema di **Sincronizzazione**
- Ogni filosofo: **pensa e mangia**
- Per mangiare **deve avere entrambe le forchette**: destra & sinistra
- **DOMANDA**: è possibile scrivere un programma per ogni filosofo in modo che non si blocchi mai?

Problema dei 5 filosofi

- Problema classico che modella l'**accesso esclusivo** ad un **numero limitato di risorse** da parte di processi in concorrenza.
- **soluzione 1:**

```
int N=5
function philosopher(int i)
    think()
    take_fork(i)
    take_fork((i+1) mod N)
    eat()
    put_fork(i)
    put_fork((i+1) mod N)
```

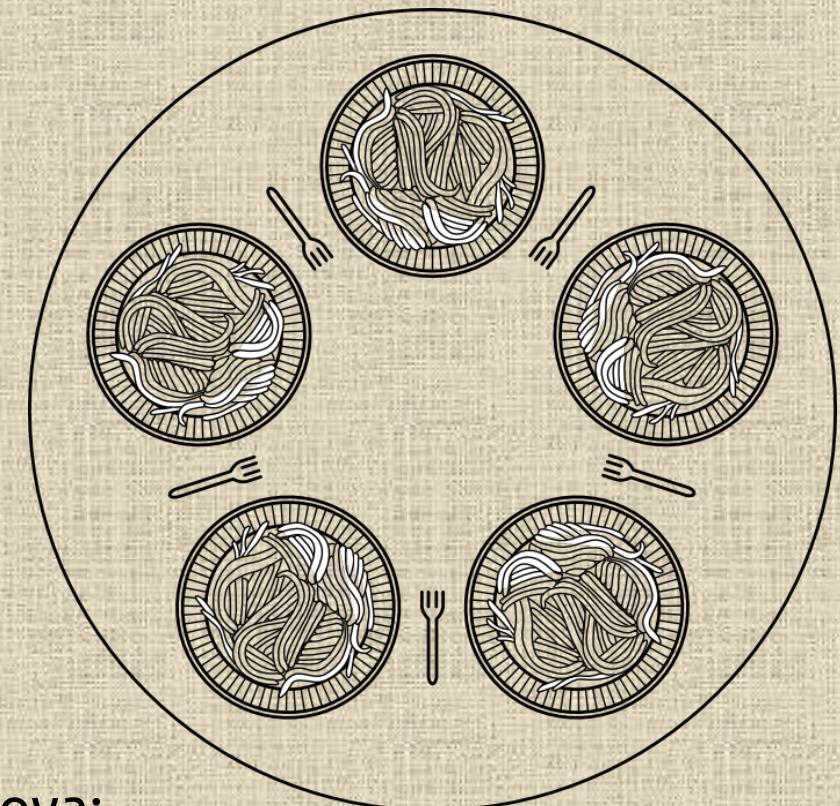


- **soluzione 2:** controlla con rilascio, riprova;
- Soluzioni **non corrette!!!!**
- **STARVATION:** programmi eseguiti indefinitamente senza avanzamento

Problema dei 5 filosofi

- Problema classico che modella l'**accesso esclusivo** ad un **numero limitato di risorse** da parte di processi in concorrenza.
- **soluzione 1:**

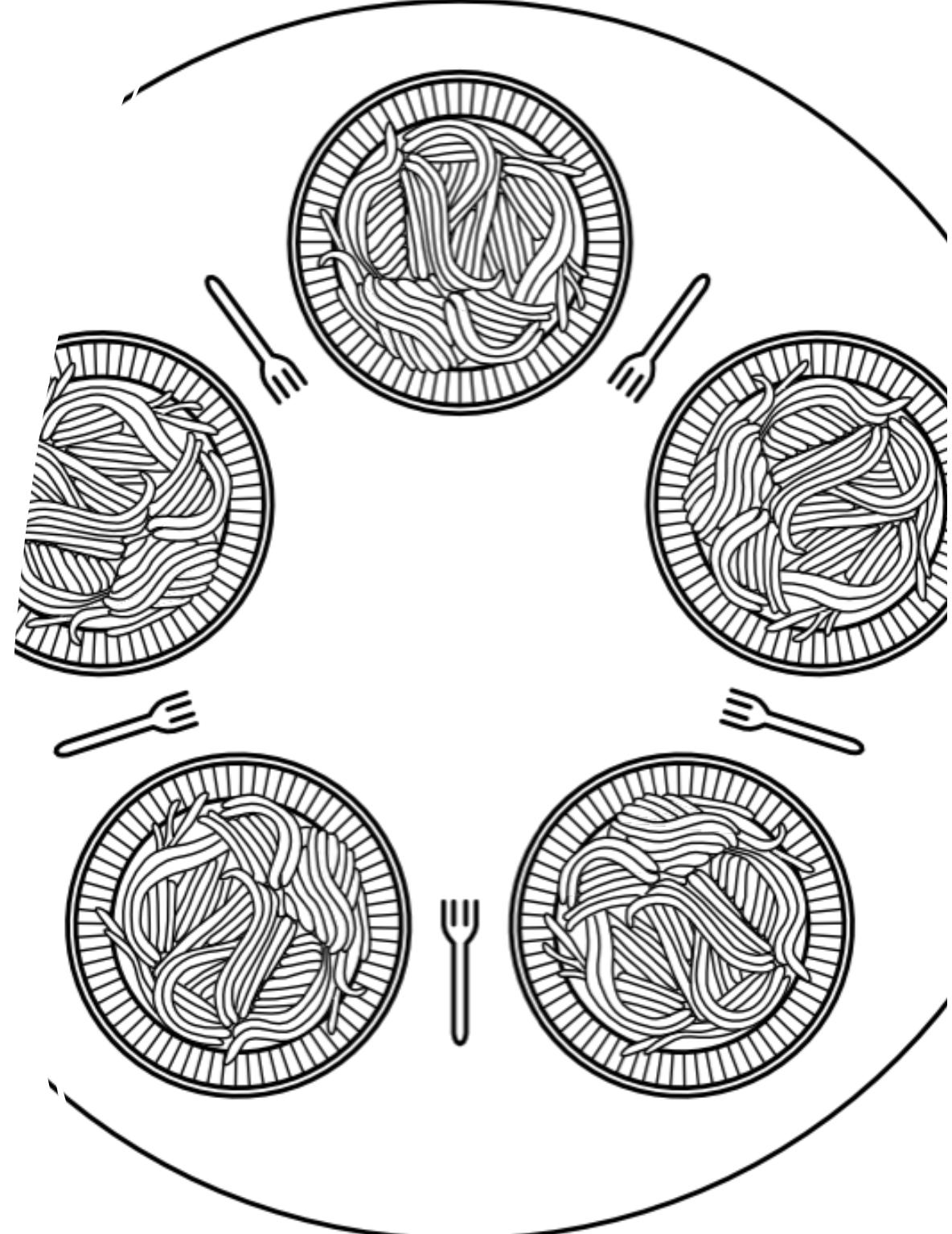
```
int N=5
function philosopher(int i)
    think()
    take_fork(i)
    take_fork((i+1) mod N)
    eat()
    put_fork(i)
    put_fork((i+1) mod N)
```



- **soluzione 2:** controlla con rilascio, riprova;
- Soluzioni **non corrette!!!!**
- **STARVATION:** programmi eseguiti indefinitamente senza avanzamento

Problema dei 5 filosofi

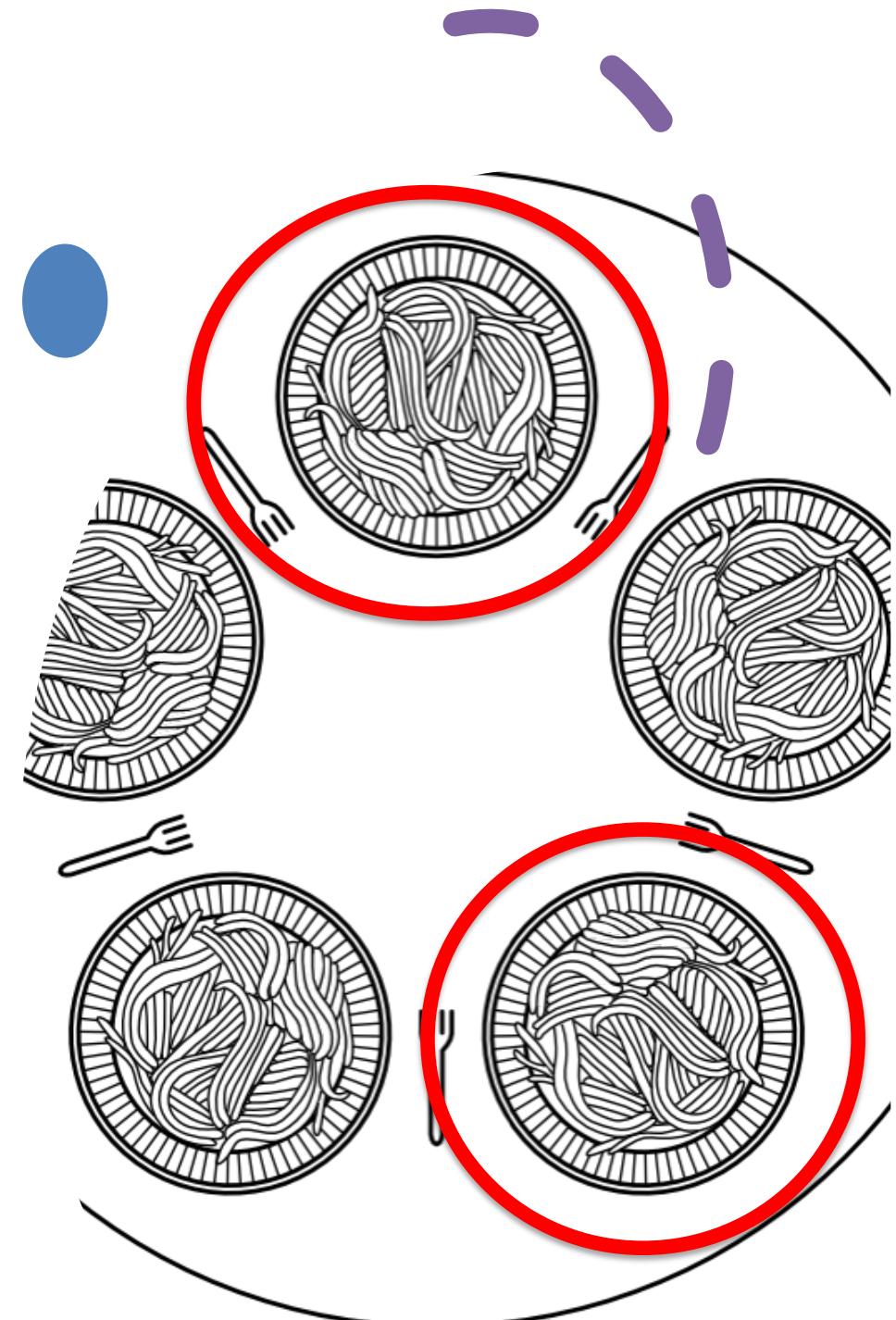
- ... altre soluzioni.....
- **soluzione 3:** controlla con **rilascio e riprova** aspettando un **tempo random**.
 - si vuole una funzione che funzioni sempre
- **soluzione 4:** utilizzo di un semaforo ***mutex***
 - Non del tutto efficiente



Problema dei 5 filosofi

- ... altre soluzioni.....
- . **soluzione 3:** controlla con **rilascio e riprova** aspettando un **tempo random**.
 - . si vuole una funzione che funzioni sempre
- . **soluzione 4:** utilizzo di un semaforo **mutex**
 - . Non del tutto efficiente

Con 5 forchette riescono
a mangiare 2 filosofi
contemporaneamente



Problema dei 5 filosofi: soluzione basata sui semafori

```
int N=5; int THINKING=0  
int HUNGRY=1; int EATING=2  
int state[N]  
semaphore mutex=1  
semaphore s[N]={0,...,0}
```

```
function philosopher(int i)  
    while (true) do  
        think()  
        take_forks(i)  
        eat()  
        put_forks(i)
```

```
function left(int i) = i-1 mod N  
function right(int i) = i+1 mod N
```

```
function test(int i)  
    if state[i]=HUNGRY and state[left(i)]!=EATING and  
    state[right(i)]!=EATING  
        state[i]=EATING  
        up(s[i])
```

```
function take_forks(int i)  
    down(mutex)  
    state[i]=HUNGRY  
    test(i)  
    up(mutex)  
    down(s[i])
```

```
function put_forks(int i)  
    down(mutex)  
    state[i]=THINKING  
    test(left(i))  
    test(right(i))  
    up(mutex)
```

Problema dei 5 filosofi: soluzione basata sui monitor

```
int N=5; int THINKING=0; int HUNGRY=1; int EATING=2
```

```
monitor dp_monitor
    int state[N]
    condition self[N]

    function take_forks(int i)
        state[i] = HUNGRY
        test(i)
        if state[i] != EATING
            wait(self[i])

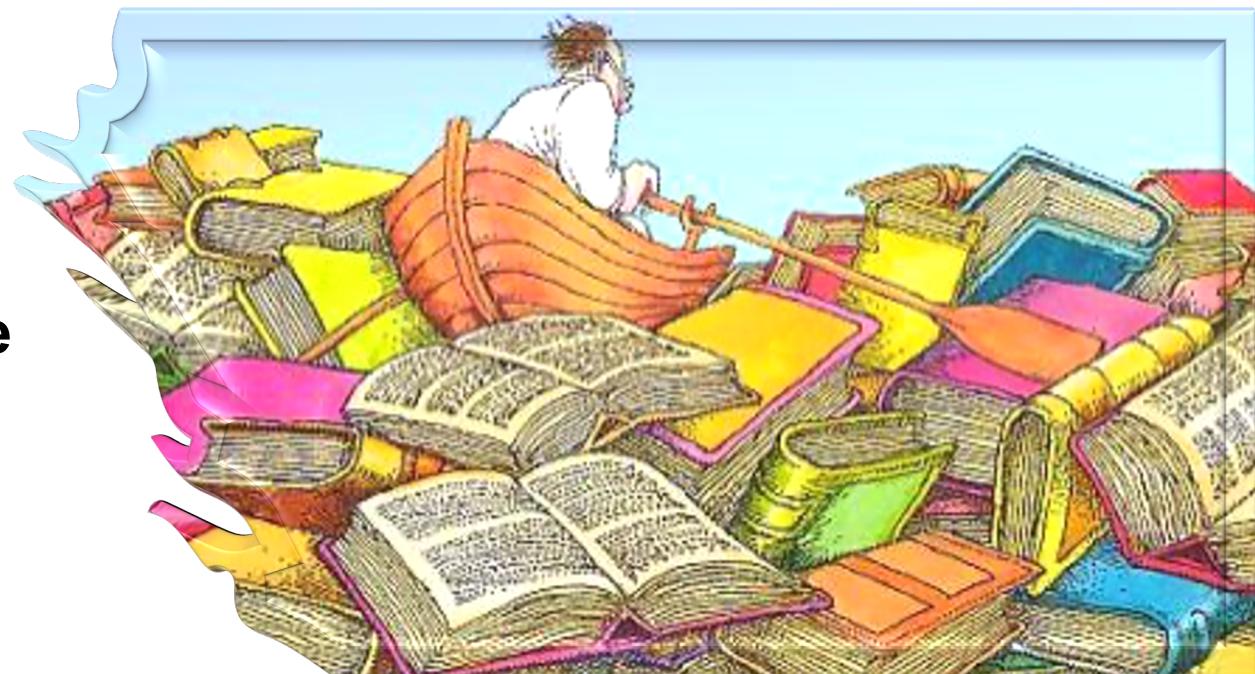
    function put_forks(int i)
        state[i] = THINKING;
        test(left(i));
        test(right(i));

    function test(int i)
        if ( state[left(i)] != EATING and state[i] = HUNGRY
        and state[right(i)] != EATING )
            state[i] = EATING
            signal(self[i])
```

```
function philosopher(int i)
    while (true) do
        think()
        dp_monitor.take_forks(i)
        eat()
        dp_monitor.put_forks(i)
```

Problema dei lettori e scrittori

- Problema classico che **modella l'accesso ad un data-base**
- Più processi leggono il DB => NO se si aggiorna il DB
- DOMANDA: come programmare i lettori e gli scrittori?



Problema dei lettori e scrittori: soluzione basata sui semafori

```
function reader()
    while true do
        down(mutex)
        rc = rc+1
        if (rc = 1) down(db)
        up(mutex)
        read_database()
        down(mutex)
        rc = rc-1
        if (rc = 0) up(db)
        up(mutex)
        use_data_read()
```

```
semaphore mutex = 1
semaphore db = 1
int rc = 0
```

```
function writer()
    while true do
        think_up_data()
        down(db)
        write_database()
        up(db)
```

- **problema:** lo scrittore potrebbe attendere per un tempo indefinito
- **Alternativa:** lo scrittore attende solo i lettori che lo precedono
 - **svantaggio:** minori prestazioni

Problema dei lettori e scrittori: soluzione n.1 basata sui monitor

```
monitor rw_monitor
    int rc = 0; boolean busy_on_write = false
    condition read,write

    function start_read()
        if (busy_on_write) wait(read)
        rc = rc+1
        signal(read)

    function end_read()
        rc = rc-1
        if (rc = 0) signal(write)

    function start_write()
        if (rc > 0 OR busy_on_write) wait(write)
        busy_on_write = true

    function end_write()
        busy_on_write = false
        if (in_queue(read))
            signal(read)
        else
            signal(write)
```

```
function reader()
    while true do
        rw_monitor.start_read()
        read_database()
        rw_monitor.end_read()
        use_data_read()
```

```
function writer()
    while true do
        think_up_data()
        rw_monitor.start_write()
        write_database()
        rw_monitor.end_write()
```

Problema dei lettori e scrittori: soluzione n.2 basata sui monitor

```
monitor rw_monitor
    int rc = 0; boolean busy_on_write = false
    condition read,write

    function start_read()
        if (busy_on_write OR in_queue(write)) wait(read)
        rc = rc+1
        signal(read)

    function end_read()
        rc = rc-1
        if (rc = 0) signal(write)

    function start_write()
        if (rc > 0 OR busy_on_write) wait(write)
        busy_on_write = true

    function end_write()
        busy_on_write = false
        if (in_queue(read))
            signal(read)
        else
            signal(write)
```

```
function reader()
    while true do
        rw_monitor.start_read()
        read_database()
        rw_monitor.end_read()
        use_data_read()
```

```
function writer()
    while true do
        think_up_data()
        rw_monitor.start_write()
        write_database()
        rw_monitor.end_write()
```

Problema dei lettori e scrittori: soluzione n.3 basata sui monitor

```
monitor rw_monitor
    int rc = 0; boolean busy_on_write = false
    condition read,write

    function start_read()
        if (busy_on_write OR in_queue(write)) wait(read)
        rc = rc+1
        signal(read)

    function end_read()
        rc = rc-1
        if (rc = 0) signal(write)

    function start_write()
        if (rc > 0 OR busy_on_write) wait(write)
        busy_on_write = true

    function end_write()
        busy_on_write = false
        if (in_queue(write))
            signal(write)
        else
            signal(read)
```

```
function reader()
    while true do
        rw_monitor.start_read()
        read_database()
        rw_monitor.end_read()
        use_data_read()
```

```
function writer()
    while true do
        think_up_data()
        rw_monitor.start_write()
        write_database()
        rw_monitor.end_write()
```