Sistemi Operativi (M-Z)

Comunicazione fra Processi (InterProcess Communication)

C.d.L. in Informatica
(laurea triennale)
A.A. 2023-2024

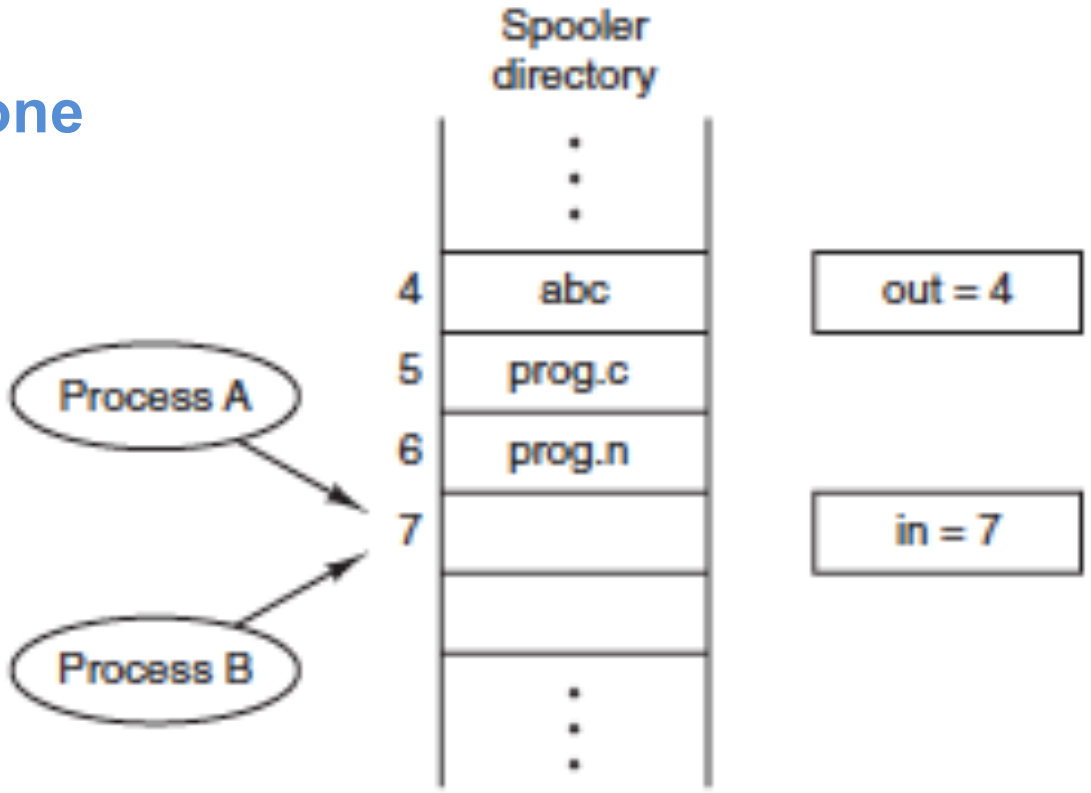
Prof. Mario F. Pavone
Dipartimento di Matematica e Informatica
Università degli Studi di Catania
mario.pavone@unict.it

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



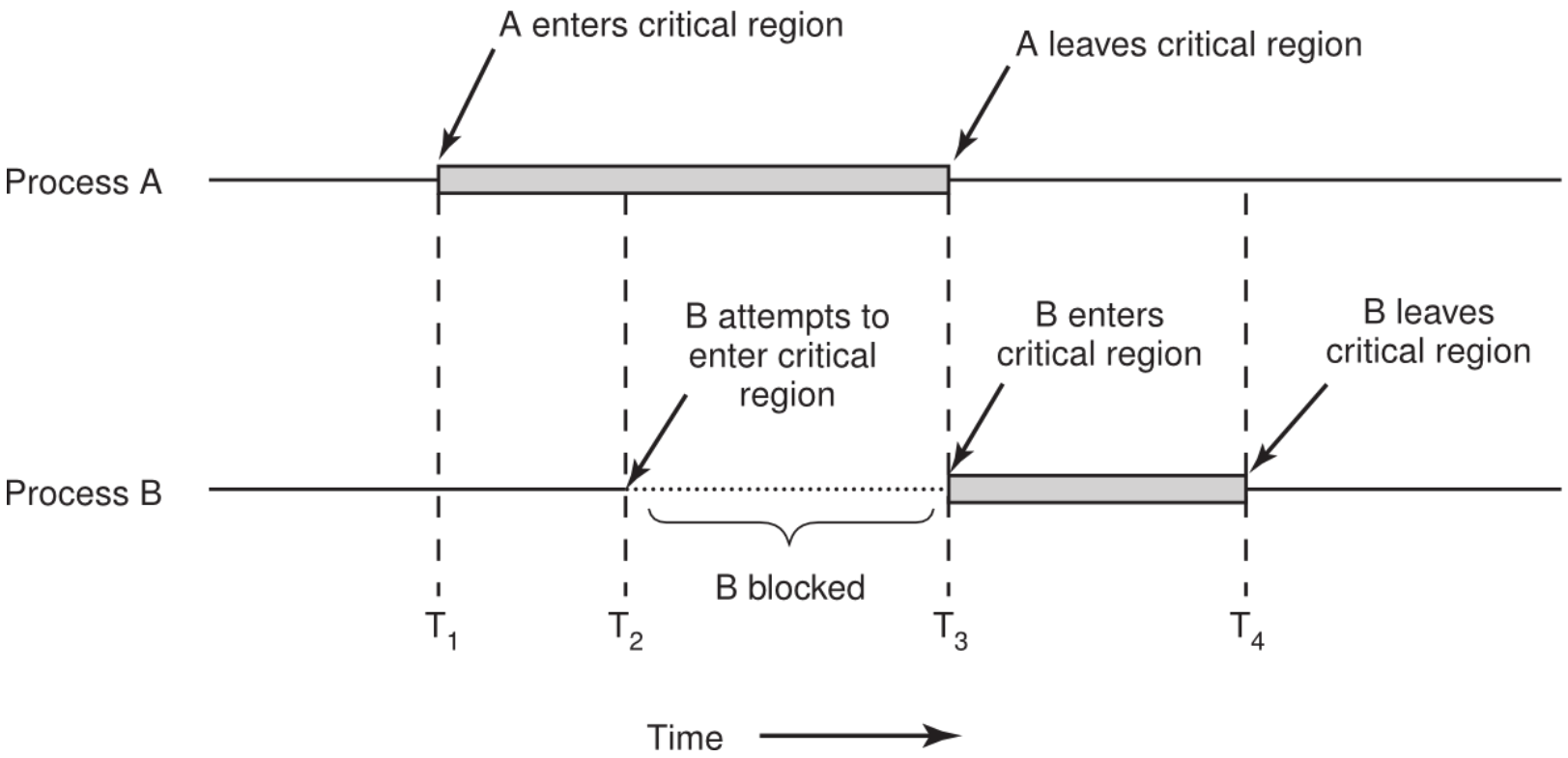
Evitare che più processi **accedino a dati condivisi contemporaneamente**

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
  - **stesso problema spool di stampa**

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

- M… …essore) offrono sp…
  - …emoria;

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

```
…gion:
MOVE LOCK,#0
```

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

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

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

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

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

```
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)
```

- Mutex: mutua esclusione

- 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

ATTENZIONE: L'ordine delle operazioni sui semafori è fondamentale…

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**)

Mutex e thread utente

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

1 ← locked

0 ← unlocked

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

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

- **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

NOTA: no contese lock => no Kernel call => veloce esecuzione

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

```
monitor tipo_risorsa {
      <dichiarazioni variabili locali>;
      <inizializzazione variabili locali>;

      public void op1 ( ) {
      <corpo della operazione op1 >;
      }
      …
      public void opn ( ) {
      <corpo della operazione opn>;
      }

      <eventuali operazioni non public>
}
```

Le **operazioni public** (o entry) sono **le sole operazioni** che possono essere utilizzate dai processi per accedere alle variabili locali. L'accesso avviene in **modo mutuamente esclusivo**.

Le operazioni **non** dichiarate **public** non sono accessibili dall'esterno. Sono invocabili solo all'interno del monitor (dalle funzioni public e da quelle non public).

Le variabili locali sono **accessibili solo all'interno del monitor**.

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, sospendendosi;
    - **Brinch-Hansen**: signal come ultima istruzione di una procedura monitor
    - continuare l'esecuzione del segnalatore

I Monitor

La dichiarazione di una variabile cond di tipo condizione ha la forma:

```
condition cond;
```

Operazioni sulle variabili condition:

- Le operazioni del monitor agiscono su tali variabili mediante le operazioni:

```
wait(cond);
signal(cond);
```

wait:

- L'esecuzione dell'operazione **wait(cond)** sospende il processo, introducendolo nella coda individuata dalla variabile cond, e il monitor viene liberato. Al risveglio, il processo riprende l'esecuzione mutamente esclusiva all'interno del monitor

signal:

- L'esecuzione dell'operazione **signal(cond)** rende attivo un processo in attesa nella coda individuata dalla variabile cond; se non vi sono processi in coda, non produce effetti.

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

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

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