Programmazione Concorrente: Richiami

Advanced Computer Programming

Prof. Luigi De Simone

Sommario

Argomenti

- Richiami su Processi e Thread
- Programmazione concorrente
- Richiami sui problemi produttore/consumatore e lettore/scrittore

Riferimenti

- P. Ancilotti, M.Boari “Programmazione concorrente e distribuita”, Mc-Graw-Hill (Cap. 2 e Cap. 3)
- Dispensa su problemi di programmazione concorrente (Ancillotti – Boari, Principi e Tecniche di Programmazione Concorrente)

Processi e Thread

Programma e Processo

Programma: codifica di un algoritmo in un linguaggio di programmazione

- *descrizione* statica delle operazioni da eseguire

```
void X (int b) {
  if(b == 1) {
  …

int main() {
  int a = 2;
  X(a);
}
```

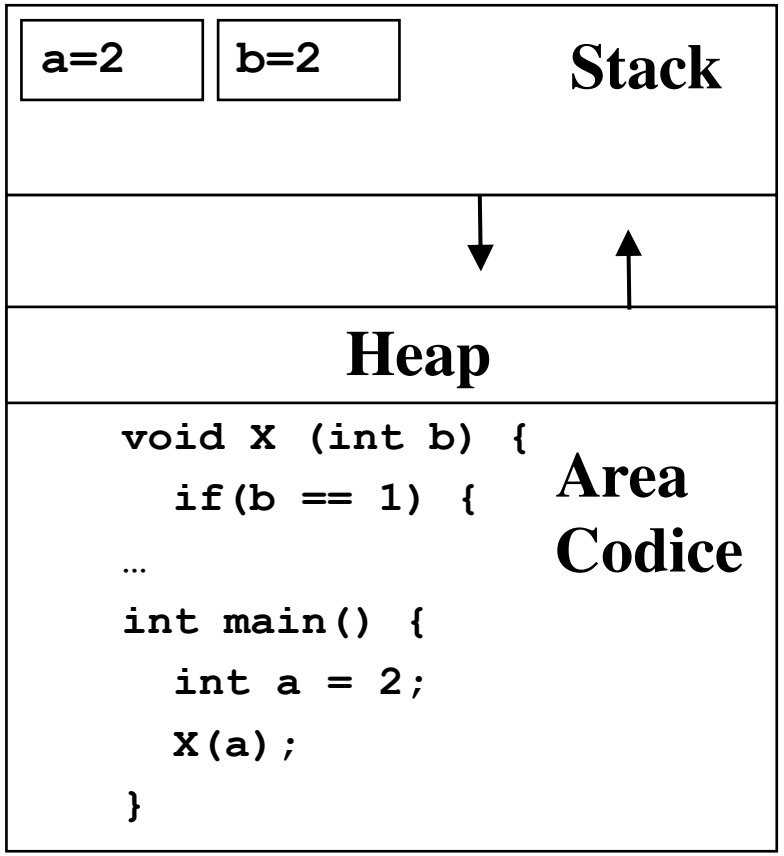
Programma e Processo

Processo: è un programma in esecuzione

- unità di esecuzione del SO che identifica le attività dell'elaboratore relative all'esecuzione del programma

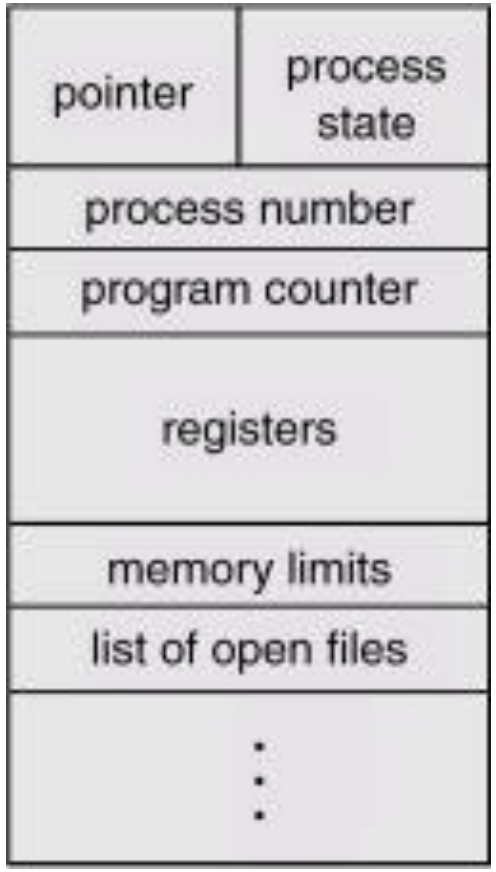
Area di memoria

allocata per un processo



Descrittore di processo

Il SO gestisce i processi associando ad ognuno di essi un descrittore (detto Process Control Block)



- identificazione del processo (PID, propretario, …)
- contesto del processore (registri CPU, stack pointer, PC, …)
- informazioni di controllo del processo
  - risorse possedute (memoria centrale, file aperti, …)
  - parametri schedulazione
  - stato del processo

Esecuzione dei processi

In generale il numero di processori è inferiore al numero dei processi, ciononostante si ha la percezione che più processi evolvano contemporaneamente

- nel caso *semplificativo* di sistema monoprocessore un solo processo alla volta può utilizzare l'unità di elaborazione

Il SO alterna l'utilizzo della CPU tra i vari processi al fine di sfruttare i *tempi morti* del processore o attribuire le risorse di calcolo in base a determinati criteri.

Context switch

L'insieme delle operazione eseguite per il prerilascio di un processo in esecuzione a favore di un nuovo processo si chiama context switch

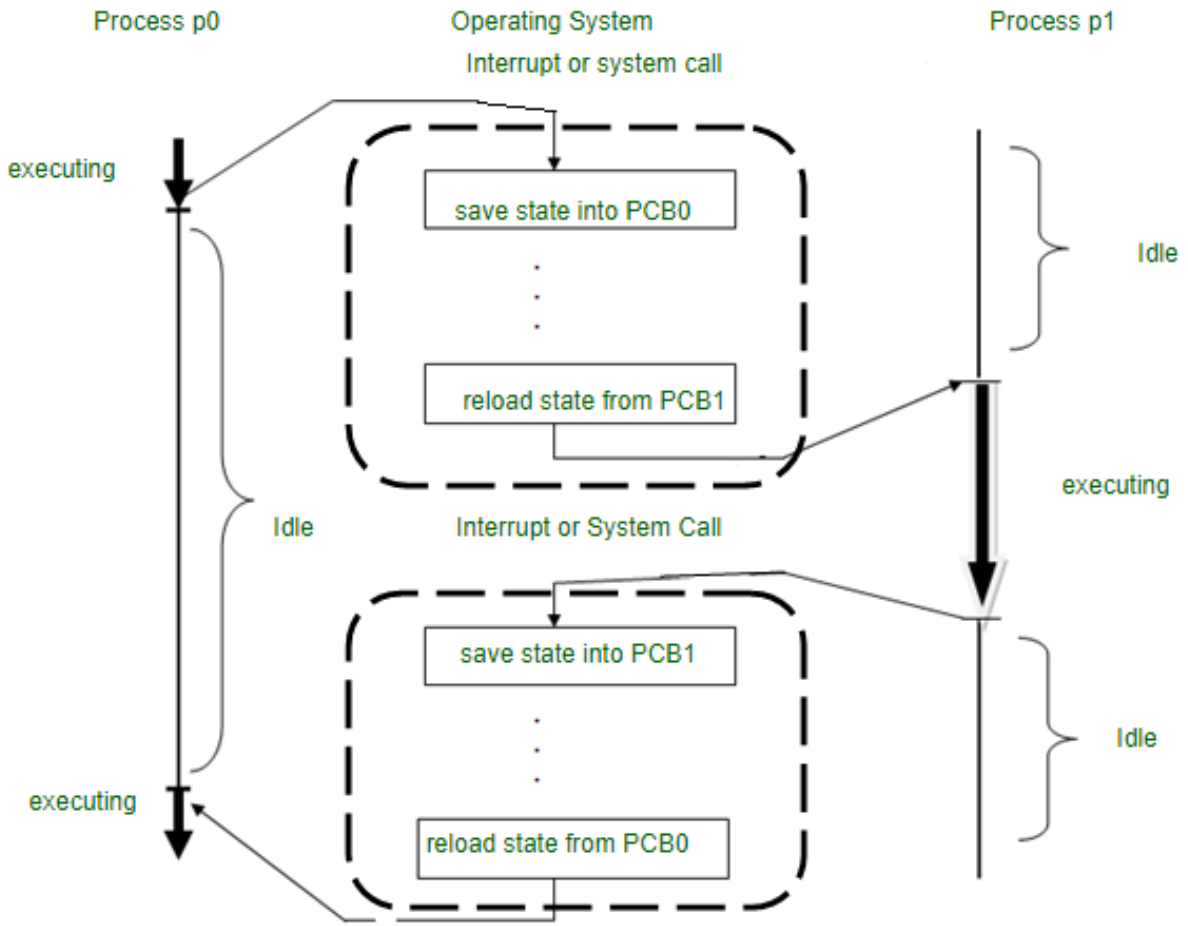
Per esempio, un context switch può avvenire a seguito di:

- Timeout: il tempo assegnato al processo è scaduto;
- Interruzioni di I/O;
- System call: il processo richiede un servizio al SO;
- …

Context switch

Come avviene un context switch

```
Procedure Context_Switch
begin
        Salvataggio_stato
        Scheduling_CPU
        Ripristino_stato
end
```



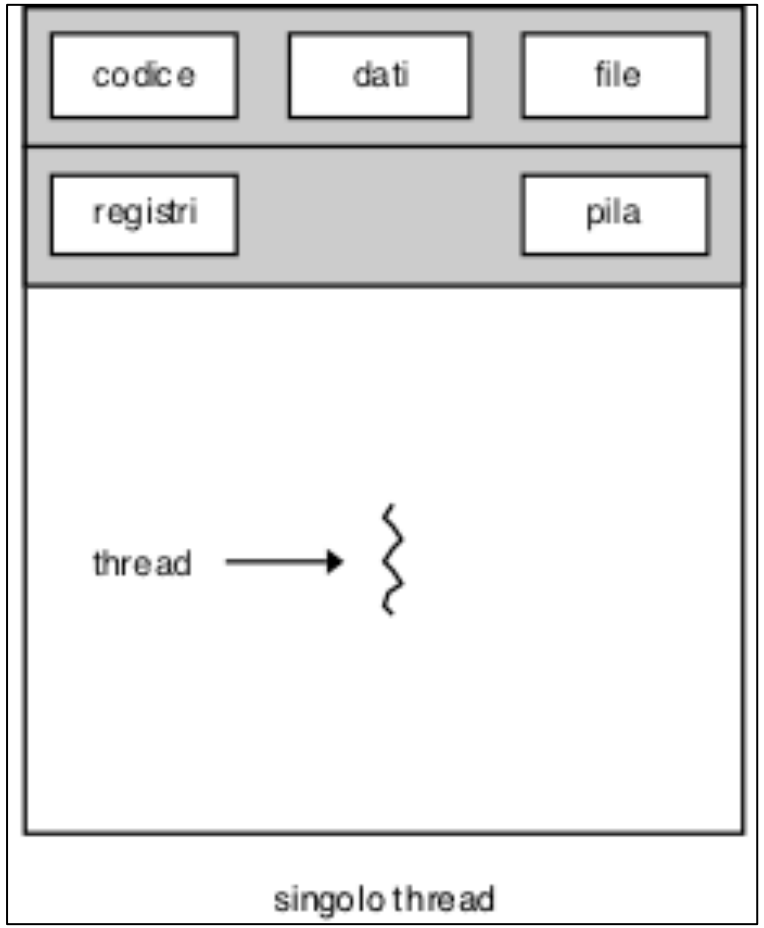
Processo e thread

- **Un processo incapsula due concetti importanti:**
  - **esecuzione** (per es., flusso di controllo e stato di esecuzione)
  - **possesso di risorse** (per es., spazio di indirizzamento, risorse di I/O)

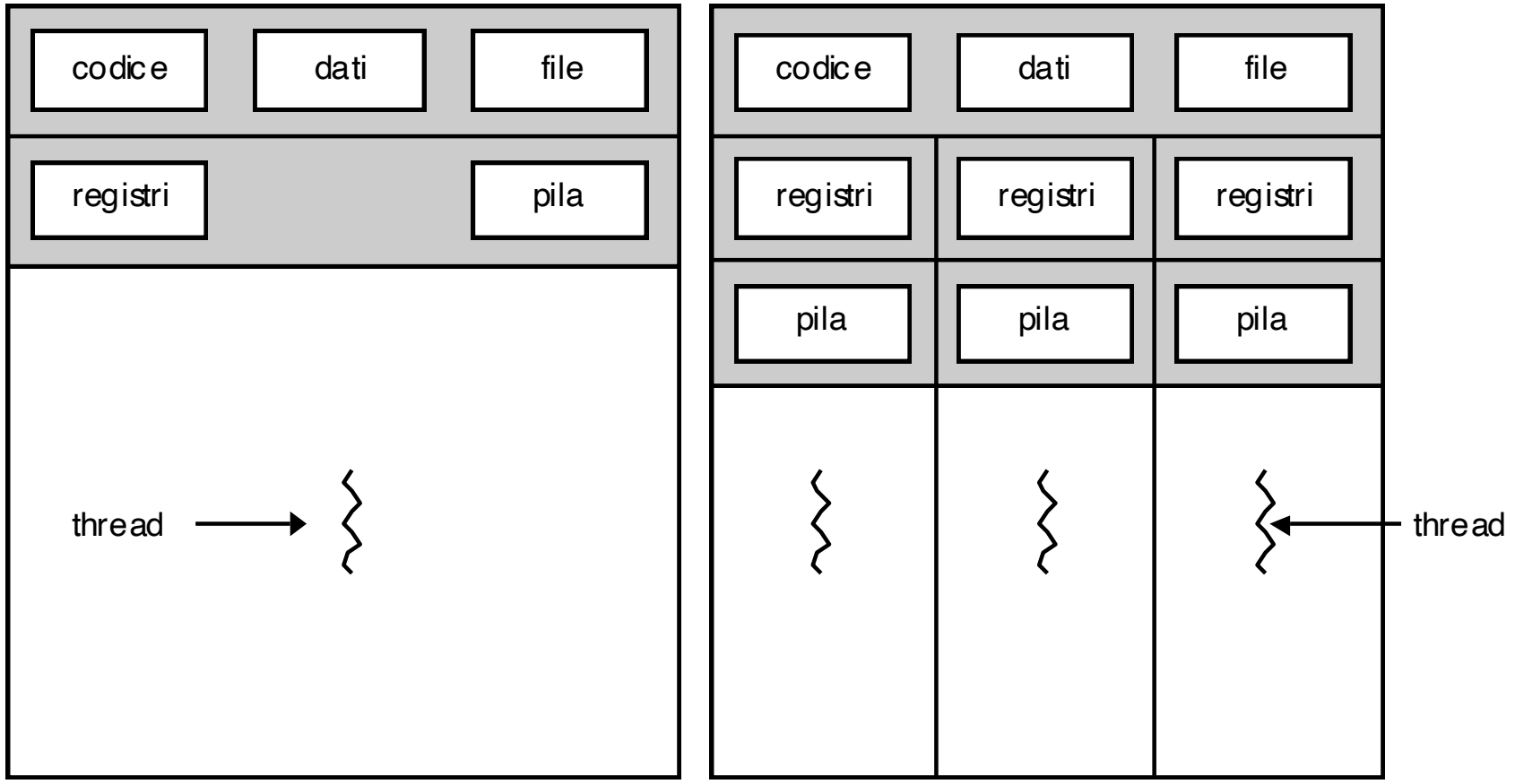
- **Nozione di Thread**
  - Separazione tra esecuzione e possesso di risorse
  - Un thread è un flusso di controllo sequenziale in un processo
  - Un processo definisce lo spazio di indirizzamento e le risorse che sono **condivise** tra i thread

Thread

- **Un thread è un flusso di controllo sequenziale in un processo**
  - Ha il suo insieme di registri;
  - Ha il suo stack;
  - **Non** ha una propria area heap e/o area dati statici (a differenza dei processi).



Processo multithread



- **Un programma multi-threaded si configura come un unico processo di elaborazione in grado di eseguire più task concorrentemente (ha più punti di esecuzione)**

Processi e Threads

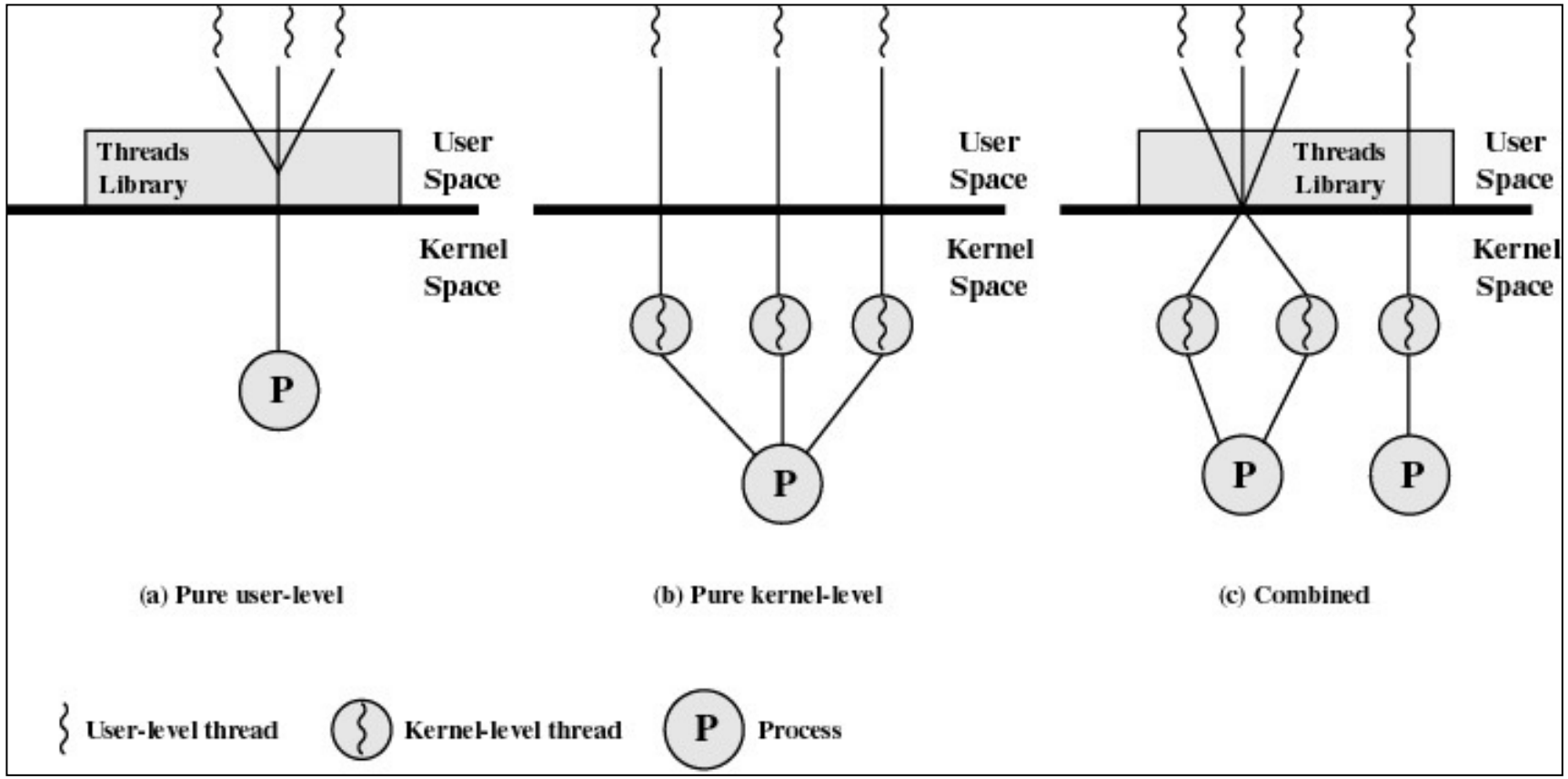
- **Si può dire che:**
  - Un processo è detto anche "processo pesante", in riferimento al contesto (spazio di indirizzamento, stato) che si porta dietro.
  - Un thread è detto anche "processo leggero", perché ha un contesto più semplice.
  - Un thread è contenuto all'interno di un processo.
  - Diversi thread contenuti nello stesso processo condividono lo stesso spazio di indirizzamento.
  - Processi differenti non condividono le proprie risorse.

Vantaggi nell'utilizzo dei thread

- **Vantaggi nell'uso dei thread:**
  - La creazione/terminazione di un thread è molto più efficiente della creazione di un processo;
  - La comunicazione tra threads è molto più semplice ed efficiente di quella tra processi poiché non coinvolge il kernel;
  - Il context switch tra threads ha decisamente un minor overhead di quello tra processi.

- **Perché realizzare programmi multithread**
  - Guadagno in performance;
  - Bloccare parte del processo non implica bloccare tutto il processo;
  - Rispecchia la natura intrinseca di molti programmi (task indipendenti, applicazioni server, calcoli numerici);
  - …

- Thread a livello utente (ULT) e a livello kernel (KLT)



Aumentare le prestazioni

- **Concorrenza: sfrutta i tempi morti del processore**
  - Multitasking: capacità di eseguire più processi contemporaneamente
  - Multithreading: più flussi di controllo nello stesso processo
- **Parallelismo**
  - Implicito: Instruction Level Parallelism (ILP), sfrutta il parallelismo intrinseco delle istruzioni
  - Esplicito: architetture multiprocessore e multicomputer

Nelle architetture parallele, più thread possono eseguire su processori diversi, ognuno gestito in concorrenza.

Speed-up

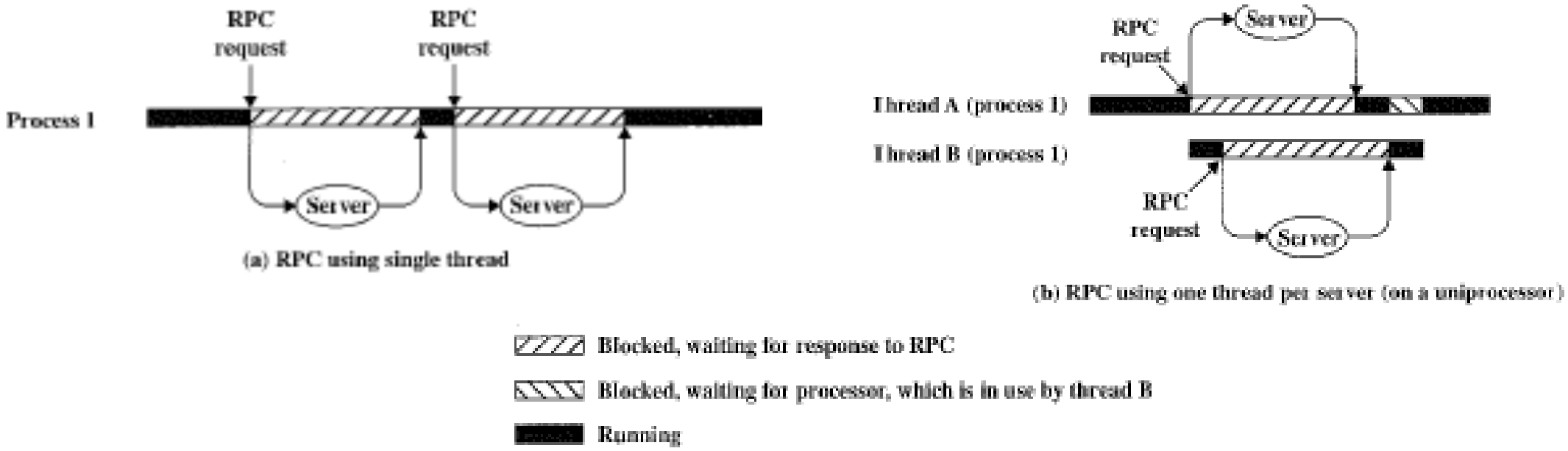
- **Concorrenza e parallelismo sono introdotte al fine di velocizzare l'esecuzione dei programmi**

$$\text{Speed-up } S = T_S / T_P$$

  - T_S: tempo di esecuzione sequenziale del processo
  - T_P: tempo di esecuzione parallela o concorrente

Speed-up nei sistemi monoprocessore

- Ottimizzazione dei tempi di attesa



Nel caso in esempio, l'attesa di un thread non deve precludere l'esecuzione dell'altro thread (occorrono KLT, no ULT)

Speed-up: Ts/Tp > 1

Limiti del multithreading

- L'uso indiscriminato dei thread può **aumentare l'overhead** dovuto allo scheduling e al context-switch:
  - più thread implicano più context-switch, lasciando quindi meno tempo utile all'elaborazione;
- Inoltre, il tempo di esecuzione concorrente è compromesso dalla presenza di **punti di sincronizzazione** tra i thread (ad es., accesso in mutua esclusione a risorse condivise)
- **Quando è vantaggioso utilizzare il multithreading?**
  - Quando il tempo di esecuzione di ciascun thread è largamente superiore al tempo di context-switch tra i thread
  - Quando l'esecuzione concorrente non è troppo vincolata dalla presenza di punti di sincronizzazione (codice non parallelizzabile)

Speed-up nelle architetture parallele

- Molti problemi hanno una soluzione che è naturale ottenere con un insieme di **programmi paralleli**
- Tuttavia, (eccetto che in casi molto molto particolari) lo speed-up che si può ottenere è meno che lineare rispetto al numero di CPU disponibili
  - analogamente ai thread, **i programmi che girano in parallelo dovranno prima o poi sincronizzarsi** per mettere in comune i dati elaborati da ciascuno;
  - sarà necessaria una **fase comune di inizializzazione** prima di far partire i vari programmi in parallelo.
- C'è comunque sempre una parte di lavoro che **non** può essere svolta in parallelo a tutte le altre operazioni

Esempio

- Consideriamo ad esempio il seguente comando:

  `gcc main.c function1.c function2.c –o output`

- Supponiamo che su una macchina **monoprocessore** ci vogliano:
  - 3 secondi per compilare main.c
  - 2 secondi per compilare function1.c
  - 1 secondo per compilare function2.c
  - 1 secondo per linkare gli oggetti
- Se avessimo 3 CPU, i tre sorgenti potrebbero essere compilati in parallelo, e poi linkati assieme usando una delle tre CPU.
- Ma l'operazione di *linking* può essere eseguita solo dopo che tutti e tre i file oggetto sono stati prodotti, ossia dopo 3 secondi
- Il tempo necessario per generare output sarà quindi 3+1=4 secondi, per uno speed-up di 7/4 = 1.75, pur avendo usato il triplo dei processori.

Legge di Amdahl

- Sia **P un programma** che gira in un **tempo T su un processore**, con
  - **f** la frazione di T dovuta a **codice sequenziale**
  - **1-f** la frazione di T dovuta a **codice parallelizzabile**
- Allora, il tempo di esecuzione dovuto alla parte parallelizzabile, passa da (1-f)T a (1-f)T/n se sono disponibili **n processori**, lo **speed-up che si ottiene è allora:**

$$S = \frac{fT + (1-f)T}{fT + (1-f)T/n} = \frac{1}{f + 1/n - f/n} = \frac{n}{1 + (n-1)f}$$

S = n solo se f = 0 → assenza di codice sequenziale!

Programmazione Concorrente

Programmazione concorrente

- Intesa come l'insieme delle tecniche, delle metodologie e degli strumenti necessari per fornire il supporto **all'esecuzione di applicazioni software come un insieme di attività svolte simultaneamente**

Sviluppo di programmi concorrenti

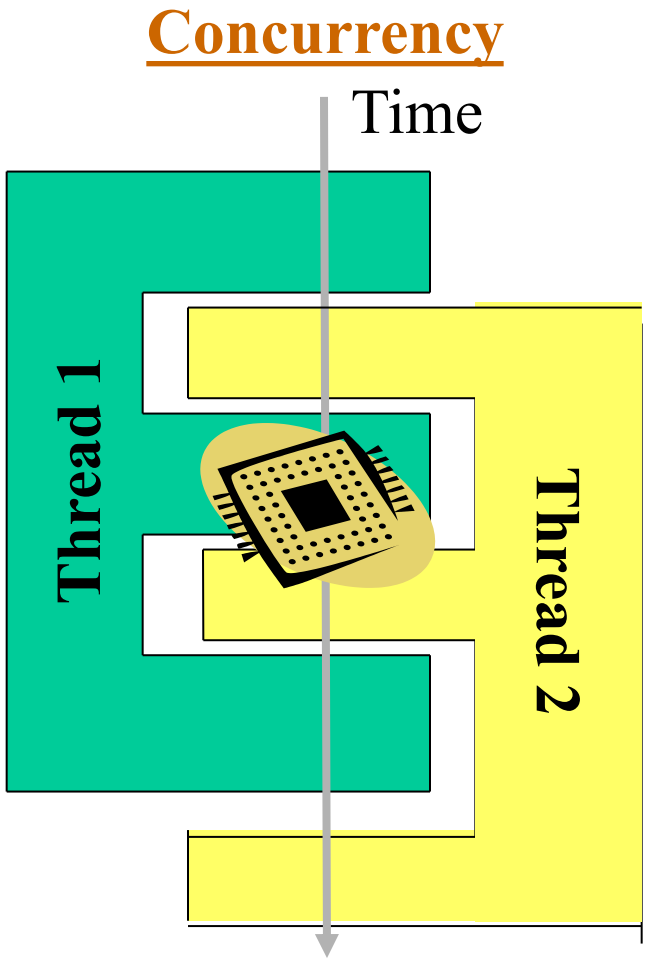
- Primitive per definire attività indipendenti (processi, threads)
- Primitive per la comunicazione e sincronizzazione tra attività eseguite in modo concorrente (concorrenza non significa parallelismo)

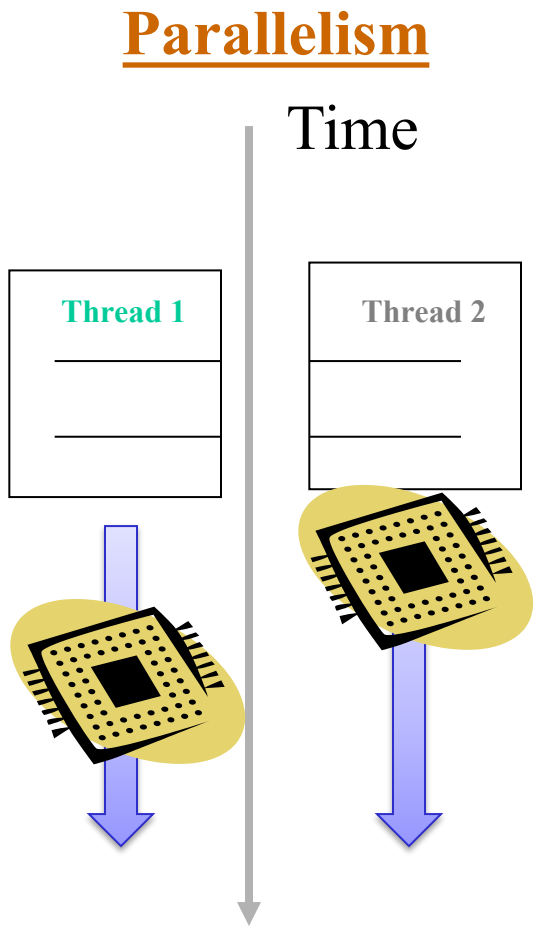
Concorrenza

- **Processi concorrenti**
  - Insieme di processi la cui esecuzione si sovrappone nel tempo.
  - Più in generale:
    - in un sistema monoprocessore due processi si dicono concorrenti se la prima operazione di uno comincia prima dell'ultima dell'altro

- **Le problematiche**
  - Assegnazione del tempo di CPU
  - Presenza di risorse condivise
  - Comunicazione tra i processi
  - Sincronizzazione

Concorrenza e Parallelismo





Programmazione concorrente

- Le due problematiche fondamentali nello sviluppo di programmi concorrenti sono:
- **Safety** – assicurare la **consistenza**:
  - **Mutua esclusione** – le risorse condivise sono aggiornate in maniera atomica
  - **Condition synchronization** – alcune operazioni potrebbero richiedere di essere differite se le risorse condivise non sono in uno stato “appropriato” (es. lettura da un buffer vuoto)
- **Liveness** – assicurare **l’avanzamento** dell’elaborazione:
  - **No Deadlock** – Alcuni processi possono sempre accedere a una risorsa condivisa
  - **No Starvation** – Tutti i processi, prima o poi (**eventually**), possono accedere alle risorse condivise.

Processi concorrenti

- **Processi indipendenti:**
  - due processi P1 e P2 sono **indipendenti** se l'esecuzione di P1 non è influenzata da P2, e viceversa (Proprietà della riproducibilità)
- **Processi interagenti:**
  - due processi P1 e P2 sono **interagenti** se l'esecuzione di P1 è influenzata da P2, e viceversa.
    - Effetto dell'interazione dipende dalla **velocità relativa** dei processi
    - Comportamento non riproducibile

Tipologie di interazione

- **Competizione:** per l'uso di risorse comuni che non possono essere utilizzate contemporaneamente (**mutua esclusione**).

- **Cooperazione:** nell'eseguire un'attività comune mediante scambio di informazioni (**comunicazione**).

- **Interferenza:**
  - **Dovuta a competizione** tra processi per uso non autorizzato di risorse comuni, oppure ad un **erronea soluzione di problemi di competizione e di cooperazione**.
  - Si manifesta spesso in modo ***non deterministico*** in quanto dovuta alla differente velocità di esecuzione dei processi.

Sincronizzazione

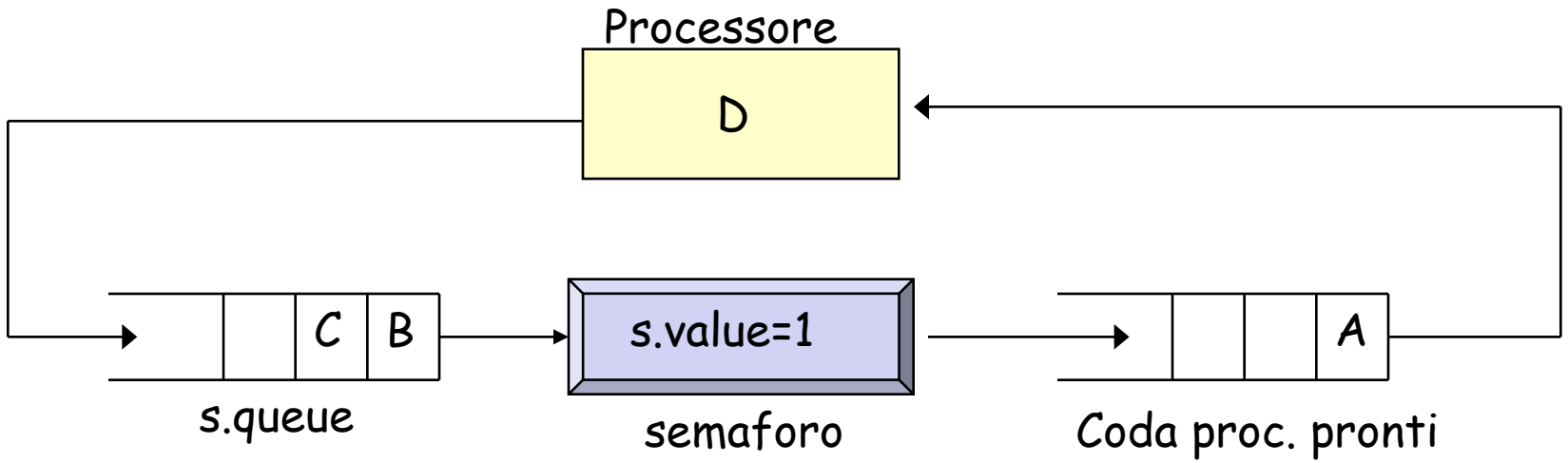
- Nei casi citati, per un corretto funzionamento, è necessario imporre dei **vincoli** nell'esecuzione delle operazioni dei processi.

- **Vincoli per la sincronizzazione**
  - **Competizione**: un solo processo alla volta deve avere accesso alla risorsa comune (**sincronizzazione indiretta o implicita**)
    - Problemi: mutua esclusione, deadlock, starvation,
  - **Cooperazione**: le operazioni eseguite dai processi concorrenti devono seguire una sequenza prefissata (**sincronizzazione diretta o esplicita**)
    - Es: problema del Produttore/Consumatore

Semaforo

- Un tipo di dato astratto **s** che incapsula:
  - **una variabile** di tipo intero (**s.value**);
  - **una coda** (**s.queue**), per tenere traccia dei processi che si sono sospesi nell'attesa di una *signal*.
- Sono definite le seguenti **operazioni**:
  - Inizializzazione della variabile ad un valore non negativo;
  - Operazione di **wait**, che ha l'effetto di **decrementare il valore del semaforo**. Se il valore del semaforo diventa negativo, il processo viene bloccato;
  - L'operazione di **signal** che ha l'effetto di **incrementare il valore del semaforo**. Se il valore del semaforo diventa minore o uguale a zero viene "sbloccato" un processo che si era sospeso durante l'esecuzione della Wait;

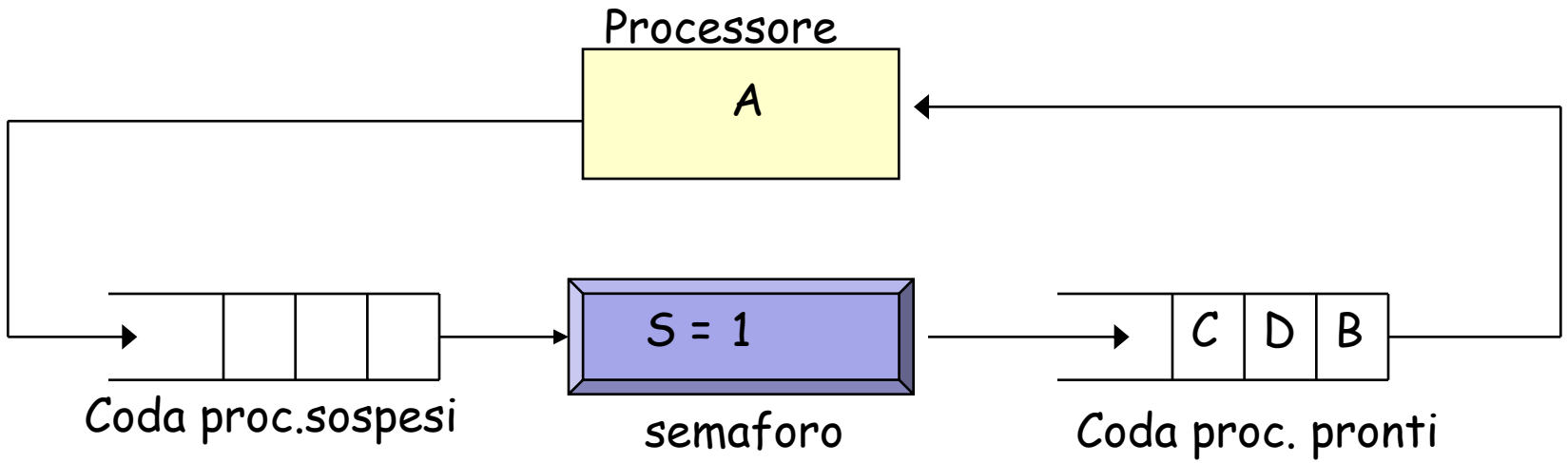
Semaforo: modello concettuale




```
void wait(sempahore s) {
s.value--;
if (s.value<0){
        s.queue.insert(Process);
        suspend(Process);
        }
}
```

```
void signal(sempahore s) {
s.value++;
if (s.value<=0){
        s.queue.remove(Process);
        wake-up(Process);
        }
}
```

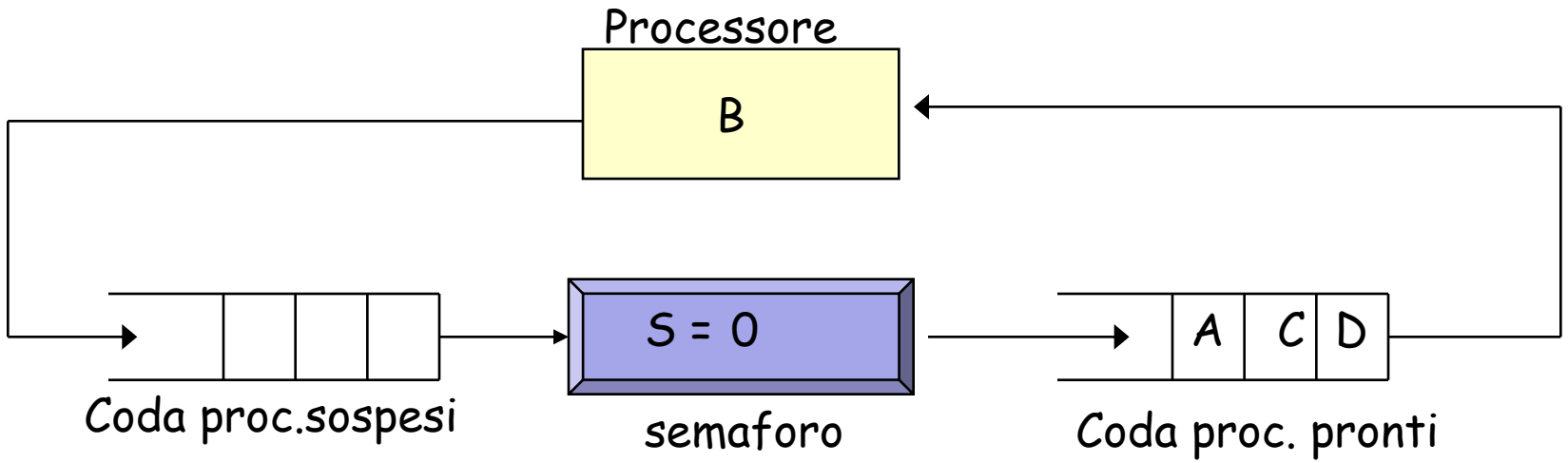
Semaforo: modello concettuale



1) Inizialmente A è in esecuzione, B, C e D sono ready, il valore del semaforo è 1

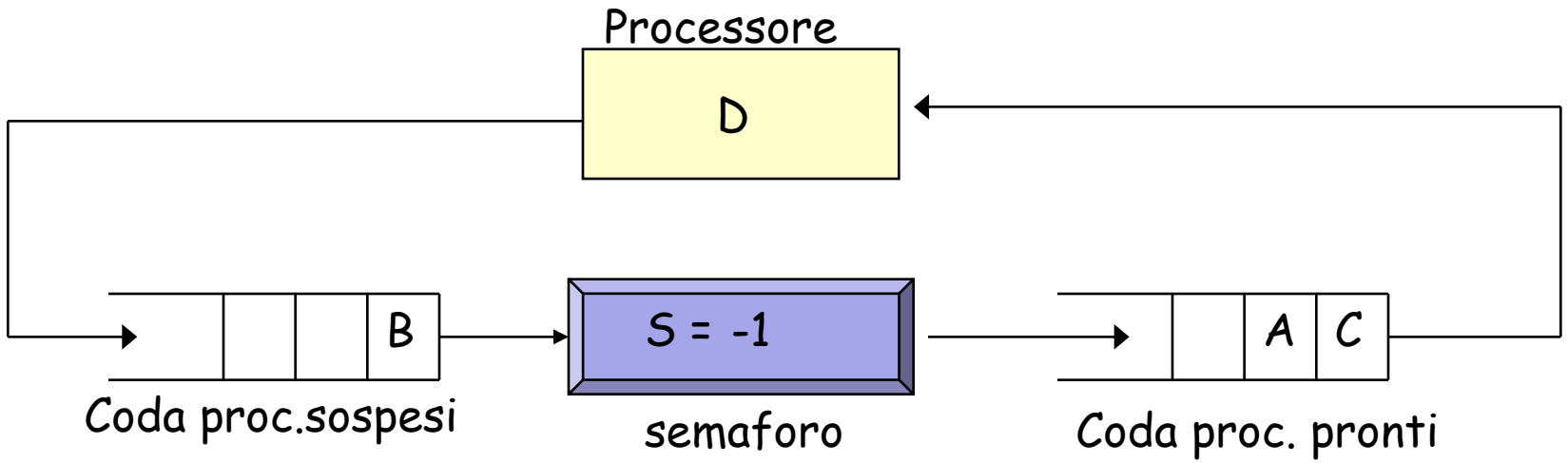
A esegue una wait

Semaforo: modello concettuale



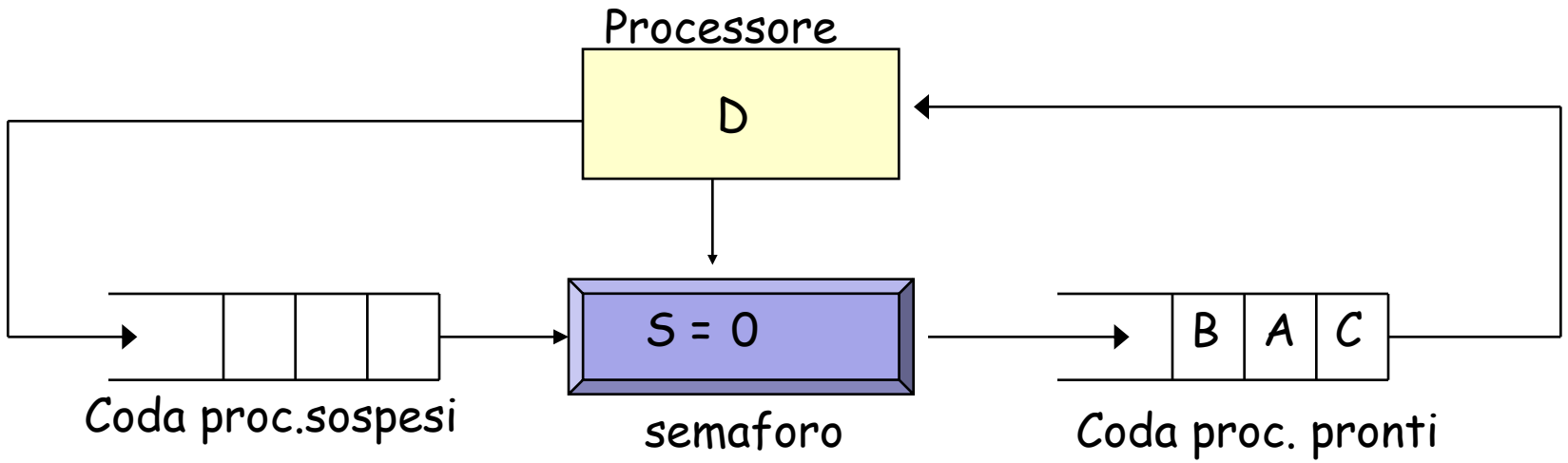
2) A esegue una wait, il valore di S viene decrementato. A non viene sospeso

Semaforo: modello concettuale



3) B esegue una wait, il valore del semaforo è decrementato e B viene sospeso

Semaforo: modello concettuale



4) D esegue una signal, che permette a B di essere inserito nella coda dei proc pronti. Si noti che in questo caso D non viene sospeso

Mutua esclusione

- Due o più processi (thread) che vogliono utilizzare una risorsa ad uso esclusivo (come ad esempio una stampante), che chiameremo **Risorsa Critica**
- La porzione del codice che utilizza la risorsa è denominata **Sezione Critica**
- Un solo processo alla volta può accedere alla sezione critica

Mutua esclusione con semaforo

- Si utilizza un semaforo, chiamato mutex, contrazione di mutual exclusion, inizializzato a 1.

```
const int n = /* numero di processi */
semaphore mutex;
…
void P(int i) { /* il codice del processo i-mo*/
        …
        wait(mutex);
        /* sezione critica */
        signal(mutex);
        …
}
…
int main() {
        mutex.value = 1;
        /* esecuzione concorrente di P(1), …, P(n) */
}
```

Costrutto Monitor

- Un **monitor** è un costrutto sintattico che associa un insieme di operazioni ad una **risorsa** condivisa tra più processi (thread)
- Garantisce la **mutua esclusione:**
  - un solo thread per volta può eseguire una procedura definita nel monitor ("il thread è nel monitor")
  - un thread che richiama una procedura di monitor, quando un altro thread è già nel monitor, viene bloccato su una coda associata al monitor
  - quando un thread **all'interno del monitor** si blocca, un altro thread deve potervi accedere

Strategie di controllo

- La sospensione di un thread –nel caso in cui la condizione logica non sia verificata– avviene utilizzando una **variabile di tipo condition.**
- Principali operazioni sulle **variabili condition**
  - wait – il thread che la esegue rilascia il lock sul monitor e si mette in attesa di essere risvegliato da una signal sulla stessa condizione (*per ogni variabile condition è implementata una coda*);
  - signal - risveglia un thread in attesa di una determinata condizione.

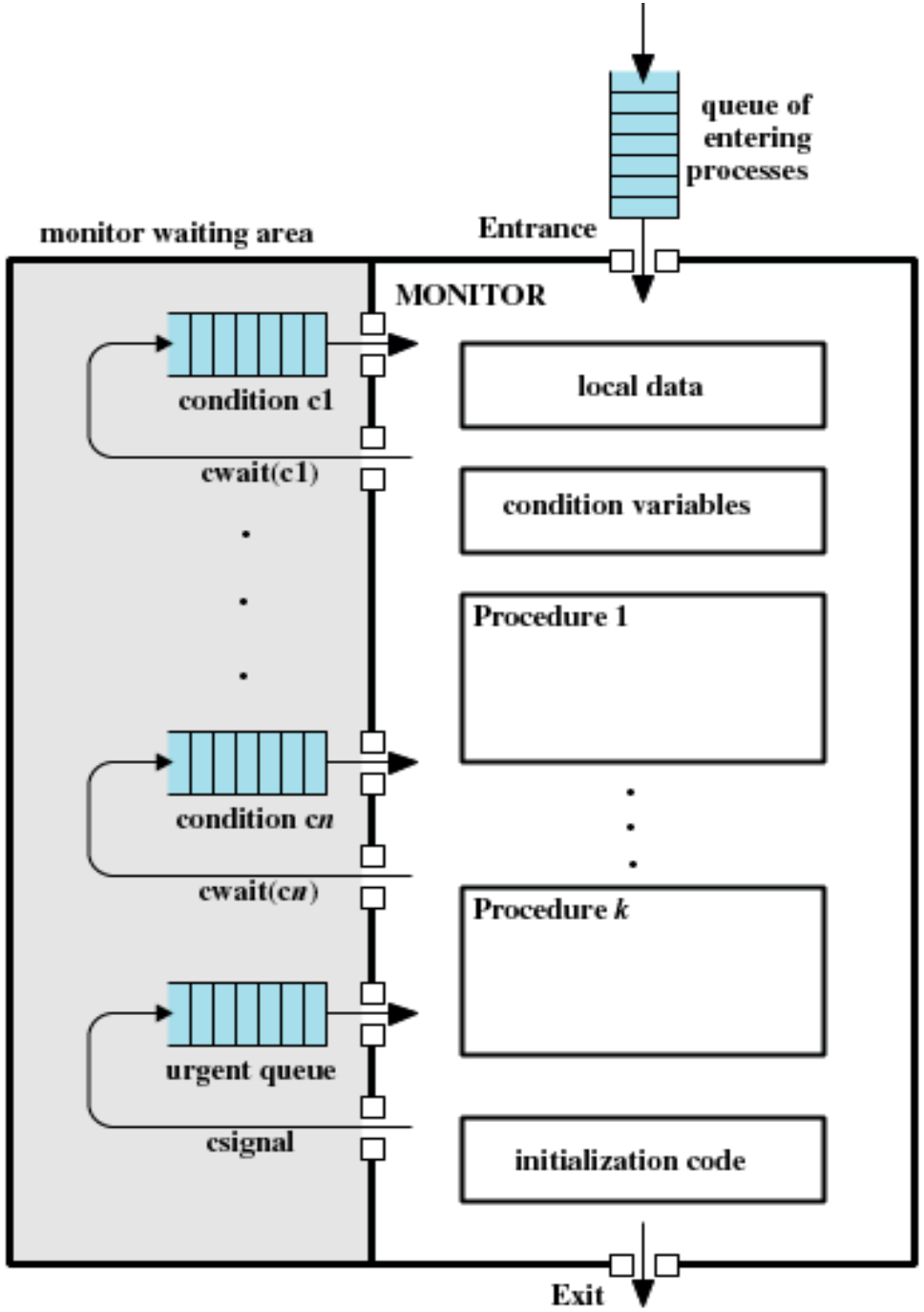
Strategie di controllo

Esempio:

var_cond x; // variabile condition

- **x.wait** provoca la sospensione del processo fino a che un altro processo non esegue una **x.signal**
- **NB: se non vi è alcun processo in attesa sulla variabile x, la signal_cond() non ha alcun effetto**

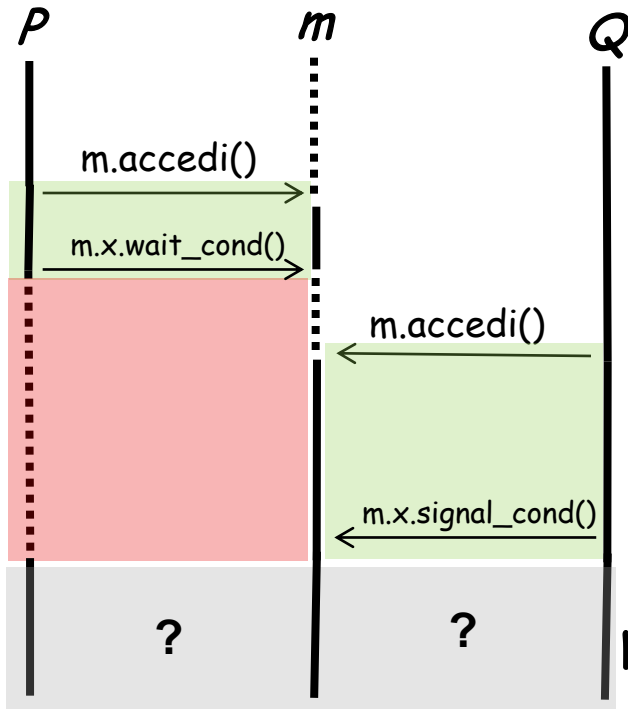
Monitor


queue of entering processes
monitor waiting area
Entrance
MONITOR
condition c1
cwait(c1)
local data
condition variables
Procedure 1
condition cn
cwait(cn)
Procedure k
urgent queue
csignal
initialization code
Exit

Semafori vs var. condition

Semafori	Varabili condition
Possono essere utilizzati ovunque, ma non in un monitor	Possono essere utilizzate solo nei monitor
wait() non blocca sempre il suo chiamante	**wait()** blocca sempre il suo chiamante
signal() può risvegliare un processo e incrementa il contatore del semaforo	**signal()** può risvegliare un processo (il «segnale» non ha alcun effetto se nessun processo è sospeso)
Se la **signal()** risveglia un processo, sia il chiamante sia il processo risvegliato riprendono l'esecuzione	Se la **signal()** risveglia un processo, solo un processo tra il chiamante ed il processo risvegliato riprende l'esecuzione

Cooperazione nei monitor



P ha accesso per primo al monitor...

...ma la risorsa **non è ancora pronta**. P quindi **si sospende**.

P ritorna **pronto**.

Q ha accesso al monitor, e può **aggiornare lo stato della risorsa**

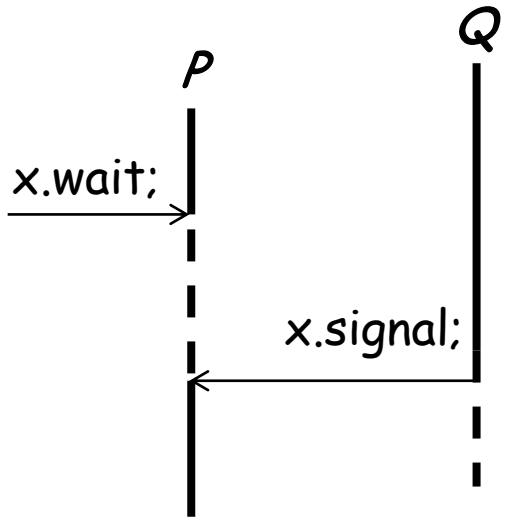
Quando la risorsa diventa **pronta** per essere usata da P, il processo **Q riattiva P**.

PROBLEMA: In linea di principio, sia P sia Q potrebbero essere eseguiti (entrambi sono pronti), ma si violerebbe la **mutual esclusione**! Occorre che solo uno dei due processi possa eseguire, e l'altro sia sospeso.

Non esiste una soluzione univoca. Diversi sistemi attribuiscono comportamenti (**semantica**) diversi alle primitive wait_cond()/signal_cond().

Prima soluzione: “signal and wait”

- *Signal_and_wait* prevede che il processo P risvegliato riprenda immediatamente l’esecuzione e che Q venga sospeso;

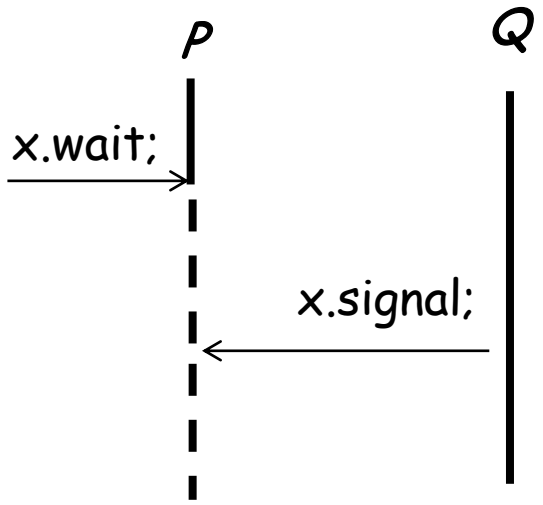


Q viene sospeso per evitare che possa modificare nuovamente la condizione di sincronizzazione.

- Un caso particolare di questo schema è la ***signal_and_urgent_wait***

Seconda soluzione: “signal and continue”

- *Signal_and_continue* (detto anche *wait and notify*) privilegia il processo segnalante rispetto al segnalato. Il processo Q segnalante prosegue la propria esecuzione, mantenendo l’accesso esclusivo al monitor, dopo aver risvegliato P



Q prosegue l’esecuzione dopo aver risvegliato P

Il processo P segnalato viene trasferito alla coda associata all’ingresso del monitor. *Poiché in questa coda possono esistere altri processi, questi possono precedere l’esecuzione di P e quindi modificare il valore della condizione di sincronizzazione*.

Produttore/Consumatore e Lettore/Scrittore: Richiami

Il problema produttore/consumatore

Due categorie di processi:

- **Produttori**, che depositano un messaggio su di una risorsa condivisa
- **Consumatori**, che prelevano il messaggio dalla risorsa condivisa

Vincoli:

- Il produttore **non può produrre** un messaggio prima che qualche consumatore abbia **prelevato** il messaggio precedente
- Il consumatore **non può prelevare** alcun messaggio fino a che un produttore non l'abbia **depositato**

Differenze con il problema della mutua esclusione

- Pur esistendo un problema (potenziale) di mutua esclusione nell'utilizzo del buffer comune...
- ...la soluzione impone un **ordinamento** nelle operazioni dei due processi

È necessario che produttori e consumatori si coordinino per indicare rispettivamente l'**avvenuto deposito e prelievo**

Problema dei Lettori/Scrittori

Due categorie di processi:

- **Lettori**, che leggono un messaggio su di una risorsa condivisa
- **Scrittori**, che scrivono il messaggio dalla risorsa condivisa

Vincoli:

1. i **processi lettori** possono accedere contemporaneamente alla risorsa
2. i **processi scrittori** hanno accesso esclusivo alla risorsa
3. i **lettori** e **scrittori** si **escludono mutuamente** dall'uso della risorsa