# Programmazione Concorrente

# Dal libro di testo (pag 2)

- Nonostante l'evoluzione tecnologica abbia incrementato le capacità computazionali grazie all'aumento della velocità delle CPU, la gestione del processore ancora oggi deve essere ottimizzata perché spesso presenta situazioni di criticità: tutti i moderni SO cercano di sfruttare al massimo le potenzialità di parallelismo fisico dell'hardware per minimizzare i tempi di risposta e aumentare il throughput del sistema, ossia il numero di programmi eseguiti per unità di tempo.
- Il programma, composto da un insieme di byte contenente le istruzioni che dovranno essere eseguite, è un'entità passiva fi nché non viene caricato in memoria e mandato in esecuzione: diviene un processo che evolve man mano che le istruzioni vengono eseguite dalla CPU, cioè si trasforma in un'entità attiva,

# Dal libro di testo (pag 2)

- Nonostante l'evoluzione tecnologica abbia incrementato le capacità computazionali grazie all'aumento della velocità delle CPU, la **gestione del processore** ancora oggi deve essere ottimizzata perché spesso presenta situazioni di criticità: tutti i moderni SO cercano di sfruttare al massimo le **potenzialità di parallelismo fisico** dell'hardware per minimizzare i tempi di risposta e aumentare il **throughput** del sistema, ossia il numero di programmi eseguiti per unità di tempo.
- Il programma, composto da un **insieme di byte** contenente le istruzioni che dovranno essere eseguite, è un'**entità passiva** finché non viene caricato in memoria e mandato in esecuzione: **diviene un processo** che evolve man mano che le istruzioni vengono eseguite dalla CPU, cioè si trasforma in un'**entità attiva**,

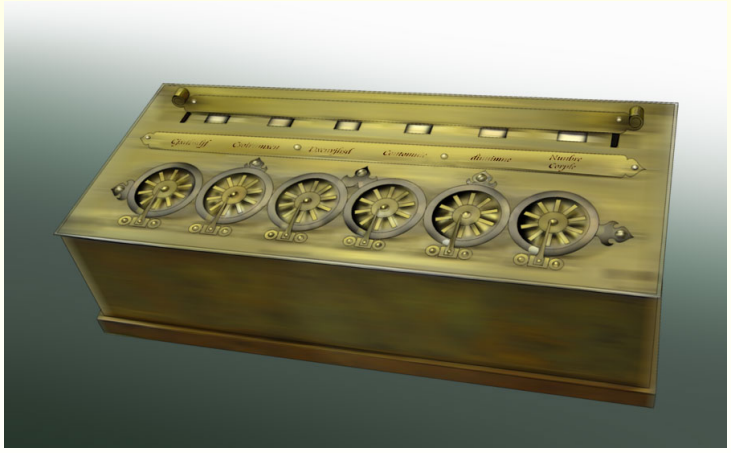
# CHE SIGNIFICA TUTTO CIÒ?!?!

# Ricominciamo da tre...

- Di cosa parleremo
  - Programmazione concorrente
- Nella realtà…
  - L’acqua è poca e la papera non galleggia
  - Usare risorse limitate
  - In tanti vogliono fare le cose
  - Bisogna coordinarsi
- Nella teoria…
  - Concorrenza
  - Parallelismo
  - Condivisione
- Esempi di concorrenza
  - Staffetta
  - Orchestra
- Esempi di condivisione
  - Incrocio stradale
  - Laboratorio scolastico
- Esempi di parallelismo
  - Autostrada
  - Compito in classe

# Un po’ di storia per capire

- Dallla pascalina alla programmazione
  - Pascalina/ babbage/ telaio jacquard
  - Monoprogrammato/ programmabile/ programmato



# Un po’ di storia per capire

- Dai sistemi batch ai sistemi time sharing
  - Gestione dell’esecuzione
  - Gestione delle risorse
  - Il ruolo del sistema operativo
  - sw → HW → sw

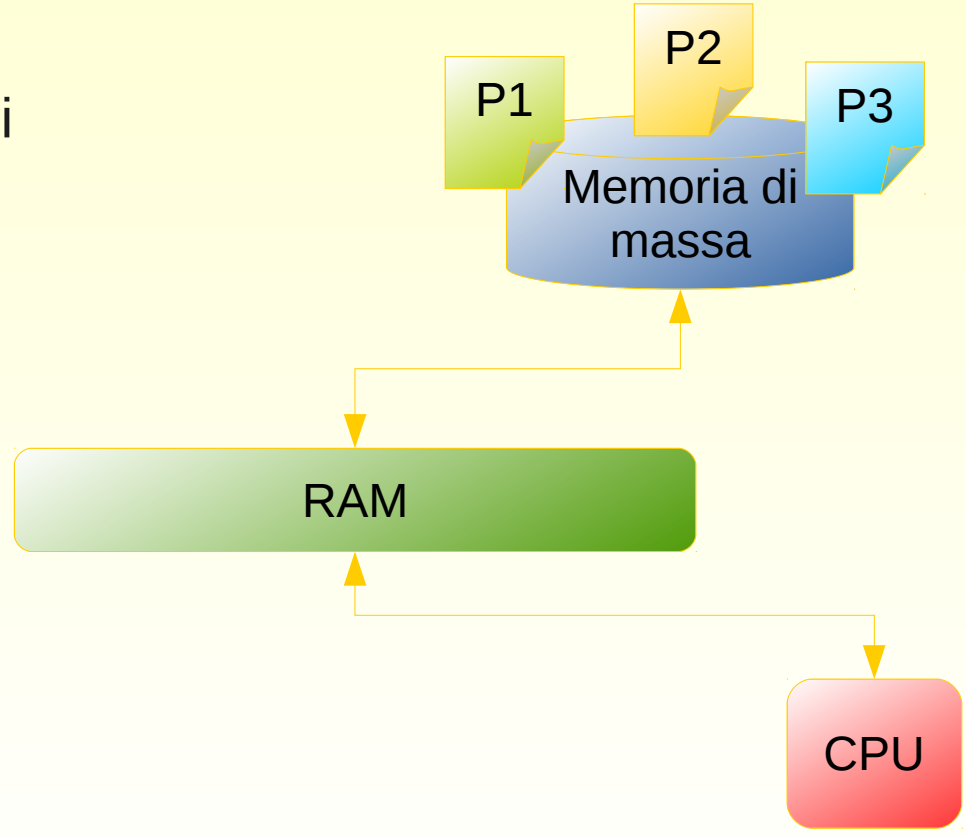

# Un po’ di storia per capire

- Dai sistemi batch ai sistemi time sharing
  - Gestione dell’esecuzione
  - Gestione delle risorse
  - Il ruolo del sistema operativo
  - sw → HW → sw

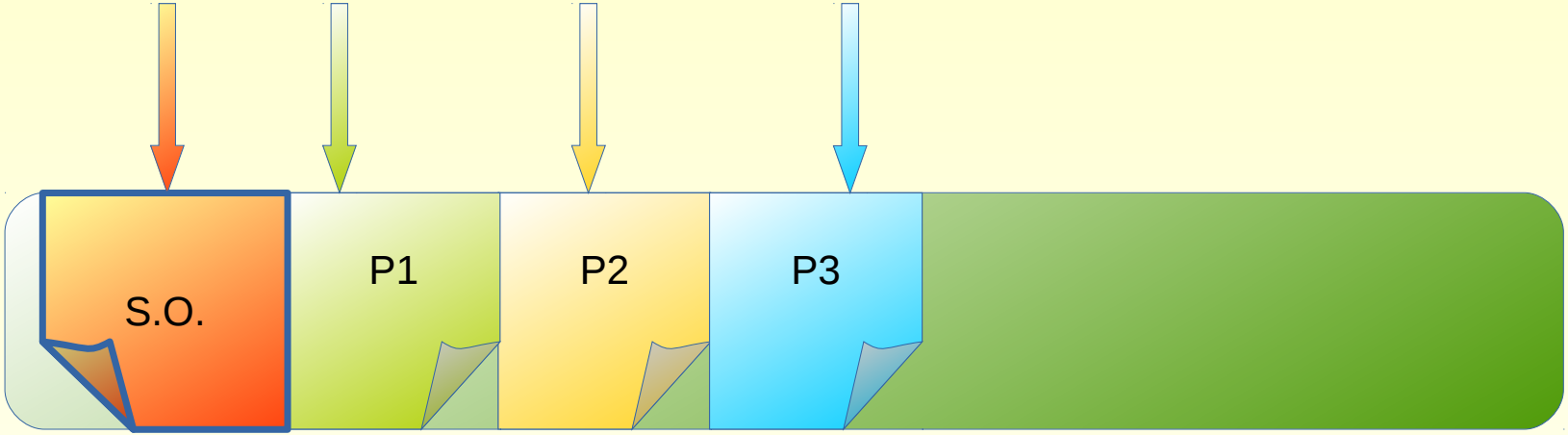


# Un po’ di storia per capire

- Dall’algoritmo al processo
  - Il process control block
  - Diagramma di stato dei processi
- Definizioni di
  - Algoritmo
  - Programma
  - Processo



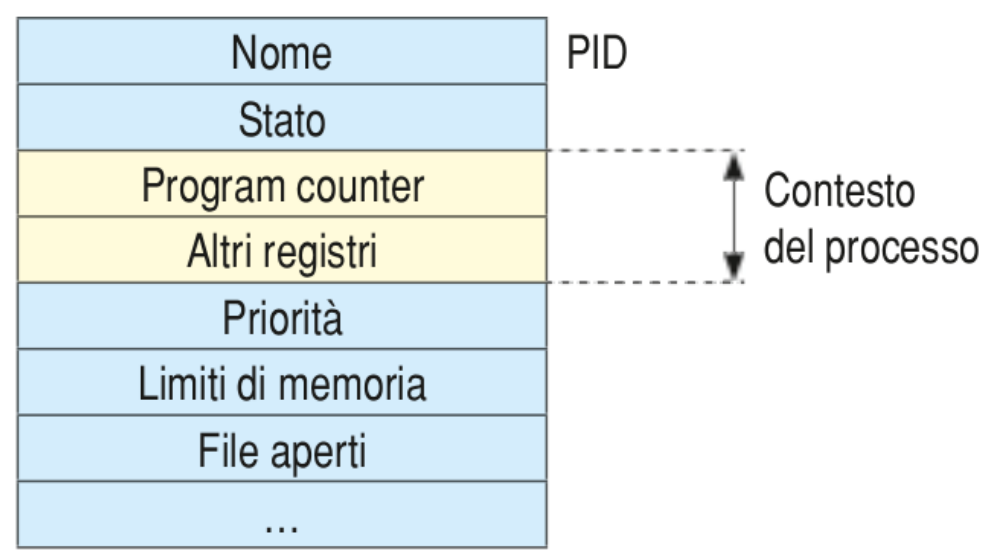
# Modello a processi



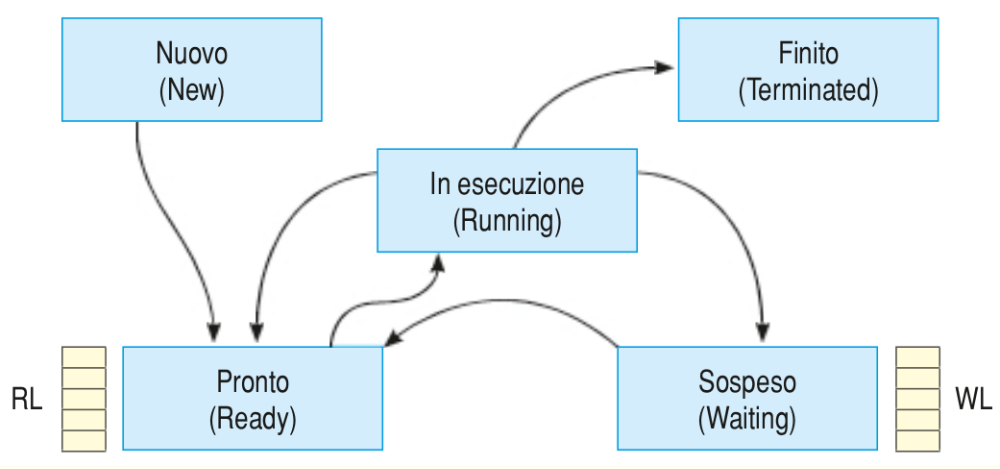

Osservazione: sono rappresentati i processi come entità logiche unitarie, senza tenere in considerazione elementi quali la memoria virtuale, la paginazione, la segmentazione etc.

# PCB

- Affinché il sistema operativo possa gestire più processi e affinché questi ultimi possano utilizzare le risorse HW in maniera consistente, sono necessarie informazioni aggiuntive oltre alle istruzioni del programma
- Tali informazioni aggiuntive sono condensate e descritte dal Process Control Block, che rappresenta un generico processo



# Diagramma degli stati di un processo




- **nuovo (new)**: è lo stato in cui si trova un processo appena è stato creato, cioè l'utente richiede l'esecuzione di un programma che risiede sul disco;
- **esecuzione (running)**: la CPU sta eseguendo le sue istruzioni, e quindi a esso è assegnato il processore.
- **attesa (waiting)**: quando gli manca una risorsa per poter evolvere (oltre alla CPU) e quindi sta aspettando che si verifichi un evento (per I/O);
- **pronto (ready-to-run)** se ha tutte le risorse necessarie alla sua evoluzione tranne la CPU (cioè è il caso in cui sta aspettando che gli venga assegnato il suo time-slice di CPU)
- **finito (terminated)** siamo nella situazione in cui tutto il codice del processo è stato eseguito e quindi ha determinato l'esecuzione; il sistema operativo deve ora rilasciare le risorse che utilizzava.

# Condivisione e Concorrenza

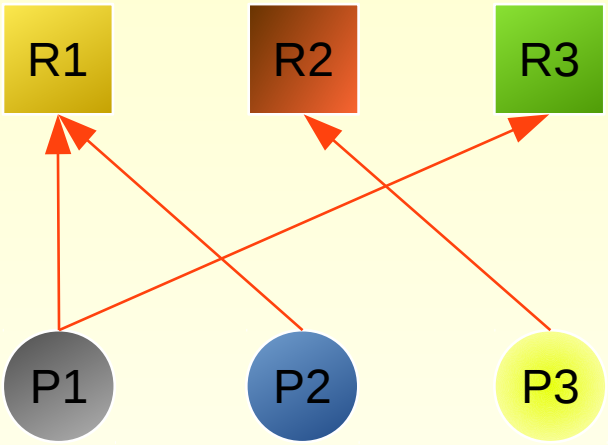
- Risorsa
  - Componente HW o SW che può essere riutilizzato. Esempi:
    - RAM, Hard disk, Stampante, Monitor
    - Variabile
- Condivisione
  - Utilizzo di una risorsa da parte di più agenti. Esempi:
    - Diversi processi vengono eseguiti sullo stesso HW
    - Un file viene aperto da più programmi

# Evoluzione dei processi

- Durante l'esecuzione, possono verificarsi essenzialmente due situazioni:
  - Un processo non ha bisogno di dati e/o risorse posseduti/e da altri processi (processi indipendenti)
  - Un processo ha bisogno di dati e/o risorse posseduti/e da altri processi (processi interagenti)
- Due processi si dicono concorrenti se la prima istruzione di uno viene eseguita quando ancora non è terminata l'esecuzione dell'altro

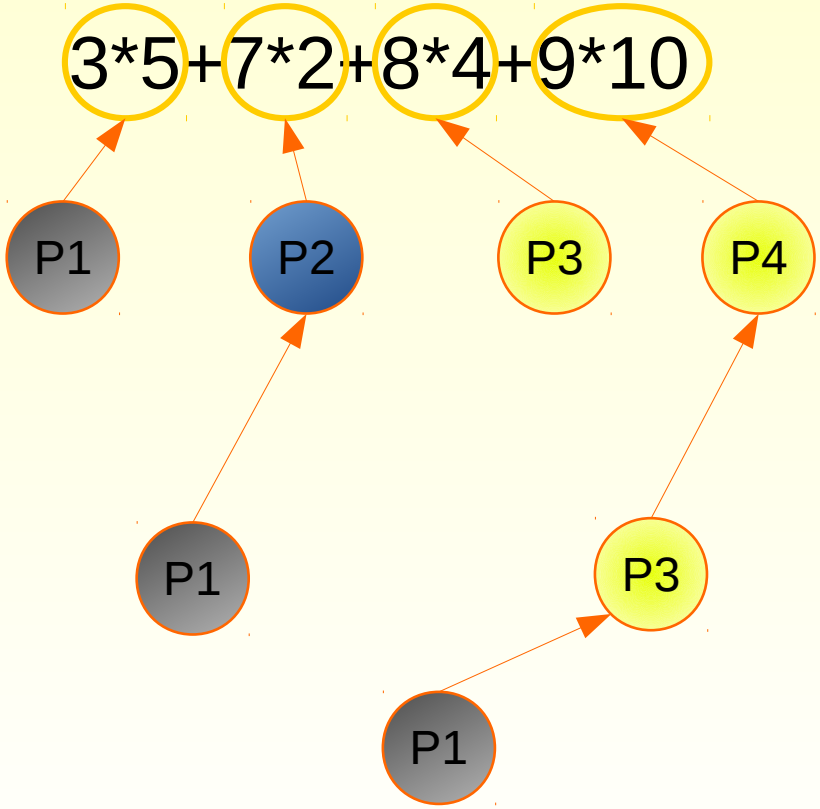
- Gestione delle risorse
  - Fare in modo che le risorse siano assegnate in maniera *consistente*
- Gestione della concorrenza
  - Fare in modo che i dati siano manipolati in maniera *consistente*

# Esempi di problemi legati alla concorrenza




- Affinché P1 e P2 possano proseguire nei loro calcoli, bisogna decidere a chi dei due assegnare la risorsa R1
- Quando questa risorsa sarà assegnata, uno dei due dovrà attendere che l'altro la rilasci per poterla utilizzare e proseguire a sua volta

# Esempi di problemi legati alla concorrenza




- Se voglio scrivere un programma che ottimizzi il calcolo di espressioni matematiche devo suddividere il lavoro fra diversi processi
- Affinché sia restituito il risultato finale, i processi devono cooperare
- Un processo non può proseguire se l'altro non gli ha fornito il risultato parziale

# Esempi di problemi legati alla concorrenza

```
procedura Scarica(x)
inizia
   TotS ← TANTI - x;
   TANTI ← TotS;
fine
```

```
procedura Carica(y)
inizia
   TotC ← TANTI + Y;
   TANTI ← TotC;
fine
```

- **Race conditions**
  - Il risultato dell'esecuzione dipende dal particolare ordine con cui viene effettuato l'accesso alla risorsa/variabile

| | | |
|---|---|---|
| TANTI=10<br>TotS ← TANTI - 3;<br>TotC ← TANTI + 5;<br>TANTI ← TotC;<br>TANTI ← TotS; | TANTI=10<br>TotS ← TANTI - 3;<br>TANTI ← TotC;<br>TotC ← TANTI + 5;<br>TANTI ← TotS; | TANTI=10<br>TotC ← TANTI + 5;<br>TotS ← TANTI - 3<br>TANTI ← TotS;<br>TANTI ← TotC; |
| (TANTI =7) | (TANTI =12)<br>ESECUZIONE SEQUENZIALE | (TANTI =15) |

# Meccanismi e Politiche

- Meccanismi
  - Forniscono gli strumenti mediante quali è possibile svolgere un determinato compito. Esempio
    - Semafori di un incrocio, Time out in una partita di basket, time sharing

- Politiche
  - Decidono come gli strumenti vengono utilizzati. Esempio
    - Durata del verde, numero e durata dei time out per partita, tempo disponibile per l'esecuzione

# Come risolvere il problema

- Meccanismi
  - Istruzioni atomiche (IA)
    - Istruzioni che sono eseguite a livello macchina senza la possibilità che vengano interrotte. Durante la loro esecuzione gli interrupt vengono disabilitati e la CPU non “accetta” altre richieste di esecuzione
  - Sezioni critiche
    - Estensione del concetto di IA. Sono porzioni di codice che manipolano variabili e/o risorse condivise e che vengono eseguite esclusivamente da un processo per volta
  - Semafori
    - Strumento SW supportato da apposito HW che permette di regolare l’accesso a variabili/risorse condivise
- Politiche
  - Mutua esclusione
  - Sincronizzazione

# Come risolvere il problema

- MUTUA ESCLUSIONE
  - Quando un processo sta eseguendo una sezione critica, nessun altro processo deve poter eseguire una sezione critica
- PROGRESSO
  - I processi che non sono in sezione critica non possono impedire ad altri processi di entrare in sezione critica
- ATTESA LIMITATA
  - Un processo che abbia fatto richiesta di eseguire in sezione critica deve avere la possibilità di esaudire la sua richiesta

# Appunti TPSIT 2016-17

Tratti da Operating System Concepts 9 edizione

## Processi

I primi computer permettevano l'esecuzione di un solo programma alla volta. Questo programma aveva il controllo completo del sistema e l'accesso a tutte le risorse del sistema stesso. Al contrario, i moderni sistemi di elaborazione permettono che più programmi vengano caricati in memoria ed eseguiti concorrentemente. Tale evoluzione richiede un controllo più restrittivo e più compartimentazione dei vari programmi; e queste necessità sfociano nella nozione di processo, che è un programma in esecuzione. Un processo è l'unità di lavoro in un moderno sistema time-sharing.

Più è complesso un sistema operativo, più ci si aspetta che si esegua compiti per conto dell'utente. Nonostante il suo compito principale sia l'esecuzione dei programmi utente, esso deve avere cura di tutti i task del sistema che sono lasciati fuori dal kernel stesso. Un sistema consiste quindi in una collezione di processi:

- processi del sistema operativo che eseguono codice di sistema
- processi dell'utente che eseguono codice utente

Potenzialmente, tutti questi processi possono eseguire concorrentemente, con la CPU (o le CPU) assegnate fra essi. Assegnando le CPU a diversi processi, il sistema operativo può rendere il computer più produttivo.

Anche su un sistema a singolo utente, un utente può essere in grado di far girare diversi programmi contemporaneamente: un word processor, un Web browser e un programma di posta elettronica. E anche se un utente può eseguire un solo programma alla volta, come per esempio su device embedded che non supporta il multitasking (Arduino, per esempio, ndt), il sistema operativo può avere bisogno del supporto per le proprie attività programmate internamente, come ad esempio la gestione della memoria. Sotto molti aspetti, tutte queste attività sono simili, quindi vengono tutte chiamate processi.

## Il concetto di processo

Informalmente, come detto precedentemente, un processo è un programma in esecuzione. Un processo è più che il codice del programma, che a volte è indicato come la sezione testo del processo. Esso include l'attività corrente, rappresentata dal valore del program counter (PC) e dal contenuto dei registri dei processori.

Si sottolinea che un programma di per sé non è un processo. Un programma è un'entità passiva, come un file contenente una lista di istruzioni memorizzata su un disco (spesso detto file eseguibile). Al contrario, un processo è un'entità attiva, con un PC che specifica la prossima istruzione da eseguire ed un insieme di risorse associate. Un programma diventa un processo quando un file eseguibile è caricato in memoria centrale.

Due tecniche comuni per il caricamento di file eseguibili sono

- il doppio click su un'icona che rappresenta il programma (nei sistemi dotati di GUI, ndt)
- la digitaizione del nome dell'eseguibile (nei sistemi dotati di CUI, ndt)

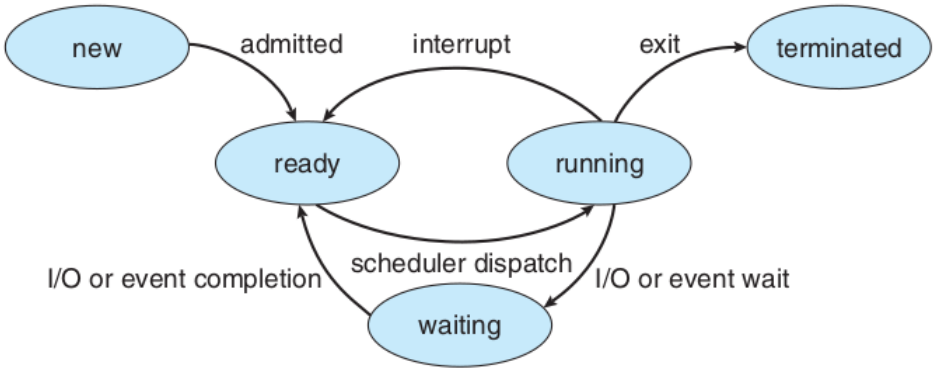
## Stati di un processo

Quando un processo è in esecuzione, cambia stato. Lo stato del processo è definito in parte dall'attività corrente del processo. Un processo può trovarsi in uno dei seguenti stati

| | |
|---|---|
| NEW (nuovo) | Il processo è in fase di creazione |
| RUNNING (in esecuzione) | Le istruzioni sono in esecuzione |
| WAITING (in attesa) | Il processo è in attesa che accada qualche evento (come il completamento di un I/O o la ricezione di un segnale) |
| READY (pronto) | Il processo è pronto per essere assegnato ad un processore |
| TERMINATED | Il processo ha terminato la sua |

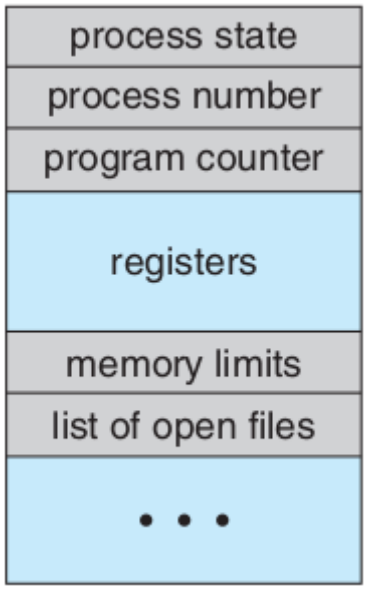
| | |
|---|---|
| (terminato) | esecuzione |

Questi nomi sono arbitrari e possono variare da un sistema operativo all'altro. Tuttavia (ndt), gli stati che rappresentano si possono trovare su tutti i sistemi. È importante capire che solo un processo può essere nello stato RUNNING su un processore ad un dato istante di tempo. Più processi possono comunque essere READY or WAITING.




## Il Process Control Block

Ogni processo è rappresentato nel sistema operativo mediante un Process Control Block (PCB), anche detto Task Control Block. Un PCB è una struttura dati (ndt) come quella mostrata di seguito




Esso contiene una serie di informazioni associate ad uno specifico processo, incluse le seguenti.

- Stato del processo (vedi diagramma degli stati, ndt)
- Valore del PC. Contatore che indica l'indirizzo della prossima istruzione che il processo deve eseguire
- Valori dei registri della CPU. Insieme al PC, questa informazione di stato deve essere salvata quando si genera un interrupt, per permettere al processo di essere ripreso correttamente in un momento successivo
- Informazioni sullo stato degli I/O. Queste informazioni include la lista dei dispositivi di I/O allocati al processo, una lista dei file aperti, e così via

In breve, il PCB serve da conteniore per le informazioni che permettono la gestione dell'esecuzione del programma (ndt).

## Creazione di un processo

Durante il corso dell'esecuzione, un processo può creare diversi altri processi. Come menzionato prima, il processo creatore viene chiamato processo padre e i nuovi processi vengono chiamati processi figli di questo processo. Ognuno di questi nuovi processi può creare altri processi, formando un albero dei processi.

Molti sistemi operativi (inclusi UNIX, Linux e Windows) identificano i processi in base ad un identificatore unico del processo (PID) che è tipicamente un nuero intero. Il PID fornisce un valore unico per ogni processo nel sistema e può essere usato come un indice per accedere a vari attributi di un processo.

Quando un processo crea un nuovo processo possono verificarsi due possibilità:

1) il padre continua l'esecuzione concorrentemente con il suo figlio
2) il padre attende fino a che alcuni o tutti i suoi figli abbiano terminato.

## Terminazione di un processo

Un processo termina quando finisce di eseguire la sua istruzione finale e chiede al sistema operativo di cancellarlo utilizzando la chiamata di sistema exit(). A questo punto, il processo può ritornare un valore di stato (tipicamente un intero) al padre (attraverso la chiamata di sistema wait() di quest'ultimo). Tutte le risorse del processo – incluse quelle fisiche e di memoria virtuale, file aperti e buffers di I/O – sono deallocate dal sistema operativo.

*Cosa succederebbe se un padre non utilizza la chiamata a wait() e al contrario termina la sua esecuzione, lasciando quindi orfano suo figlio? Linux e UNIX gestiscono questo scenario assegnando il processo orfano al processo init (il primo processo avviato dal sistema operativo, ndt)*

## Comunicazione fra processi

I processi che eseguono concorrentemente nel sistema operativo possono essere sia indipendenti che cooperanti.

- Un processo è indipendente se non può influire su o subire influinze da altri processi che stanno eseguendo nel sistema operativo. Qualunque processo che non condivide dati con altri processi è indipendente

- Un processo è cooperante se può influenzare o essere influenzato da altri processi. Chiaramente, qualunque processo che condivide dati con altri processi è un processo cooperante

Esistono diverse ragioni per fornire un ambiente (di esecuzione, ndt) che permette la cooperazione fra i processi:

- 🗝 Condivisione di informazioni
  - ✔ Poiché diversi utenti possono essere interessati alla stessa porzione di informazione (esempio: file condiviso), bisogna prevedere un ambiente che fornisca accesso concorrente a questa informazione
- 🗝 Aumento delle prestazioni computazionali
  - ✔ se si vuole che un certo compito venga eseguito più velocemente bisogna scomporlo in sotto-compiti ognuno dei quali deve essere svolto in parallelo con gli altri.
- 🗝 Modularità
  - ✔ si vuole costruire sistemi in maniera modulare, dividendo le funzioni del sistema in diversi processi (successivamente si nitrodurranno i thread, ndt)
- 🗝 Convenienza

- Anche un utente individuale può voler lavorare su più compiti allo stesso momento. Ad esempio, un utente potrebbe voler redattare un testo, ascoltare musica e compilare allo stesso tempo

I processi cooperanti hanno bisogno di meccanismi di intercomunicazione (IPC) che permettano loro di scambiare dati e informazioni. Esistono due modelli fondamentali di IPC:

- memoria condivisa
- scambio di messaggi

## Difficoltà di programmazione nei sistemi concorrenti

La scrittura di applicazioni concorrenti è più complessa rispetto a quella delle applicazioni puramente sequenziali. In genere, per progettare e sviluppare un'applicazione concorrente bisogna seguire un flusso di lavoro che prevede le seguenti fasi:

- Identificazione dei compiti
  - Implica l'analisi delle applicazioni per cercare aree che possono essere divise in compiti separati e concorrenti. Idealmente, i compiti sono indipendenti gli uni dagli altri e possono quindi essere eseguiti in parallelo su un signolo core
- Bilanciamento del carico
  - Nell'identificare i compiti che possono essere eseguiti in parallelo i programmatori devono anche assicurare che i compiti eseguano un lavoro equilibrato. In alcuni casi, un certo compito può non contribuire in modo significativo (si sprecano risorse, ndt)
- Suddividione dei dati
  - Così come le applicazioni sono suddivise per compiti, l'accesso e la manipolazione dei dati deve essere suddiviso per essere eseguito su diversi core
- Dipendenza dei dati
  - I dati su cui si effettua un accesso (condiviso, ndt) devono essere esaminati per individuare eventuali dipendenze tra due o più compiti.

Quando un compito dipende dai dati di un altro, i programmatori si devono assicurare che l'esecuzione del compito sia sincronizzata per soddisfare tale dipendenza

- Test e debug
  - ✔ Quando un programma è eseguito in parallelo su più processori, possono essere possibili diversi scenari di esecuzione. Testare e correggere errori in programmi concorrenti è intrinsecamente più difficile che in programmi sequenziali.

**Nota**

- Un sistema è parallelo se può eseguire più di un compito simultaneamente. Un sistema concorrente, invece, supporta più compiti permettendo al compito di fare progressi. Quindi è possilile avere concorrenza senza parallelismo

## Il problema della sezione critica

Un processo cooperante è un processo che può influenzare o essere influenzato da altri processi che sono in esecuzione nel sistema. L'accesso concorrente ai dati può dare luogo a inconsistenze.

Una situazione tipo la seguente

```
procedura Scarica(x)
inizia
   TotS ← TANTI - x;
   TANTI ← TotS;
fine
```

```
procedura Carica(y)
inizia
   TotC ← TANTI + Y;
   TANTI ← TotC;
fine
```

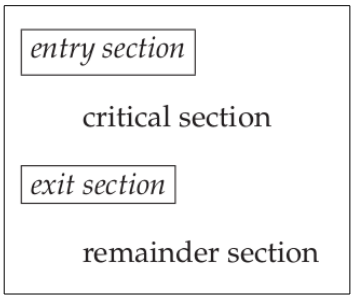
in cui

- diversi processi accedono e manipolano gli stessi dati
- il risultato dell'esecuzione dipende dal particolare ordine con cui l'accesso viene effettuato

prende il nome di race condition. Per proteggersi contro le race conditions, è necessario che solo un processo alla volta possa manipolare la variabile TANTI. Per dare questa garanzia bisogna che i processi siano in qualche modo sincronizzati

Per ottenere questo risultato si ricorre al concetto di sezione critica, ossia porzione di codice che deve essere eseguita senza che nessun altro processo possa interferire con le variabili manipolate in essa (ndt).

Il problema della sezione critica è quello di progettare un protocollo che il processo può utilizzare per coorperare. Ogni processo deve effettuare una *richiesta di permesso* per entrare in sezione critica. La sezione di codice che implementa tale richiesta viene chiamata ENTRY SECTION. La sezione critica deve essere seguita da una EXIT SECTION. La struttura generale del codice di un processo cooperante è mostrata di seguito




La ENTRY SECTION e la EXIT SECTION sono racchiuse in riquadri per evidenziare l'importanza di questa parte di codice.

Una soluzione al problema della sezione critica deve soddisfare i seguenti tre requisiti:

[I] Mutua esclusione

- Se un processo sta eseguendo la sua sezione critica, nessuna ltro processo può eseguire la sua sezione critica (relativamente ai dati in condivisione, ndt)

[II] Progresso

- Se non ci sono processi che stanno eseguendo la propria sezione critica e alcuni processi voglio eseguire una sezione critica, allora solo quei processi che non stanno eseguendo la loro la loro REMAINDER SECTION possono partecipare alla decisione su chi entrerà in sezione

critica e questa decisione non può essere rimandata indefinitamente (Se più processi vogliono eseguire la propria sezione critica, non deve esistere nessun processo che impedisca agli altri di entrare in sezione critica)

[III] Attesa limitata

- ✔ Esiste un limite sul numero di volte che ad altri processi è permesso entrare in sezione critica dopo che un processo ha fatto richiesta di entrare nella propria sezione critica.

**Nota**

- 👁 *Si assume che tutti i processi eseguano a velocità diversa da zero. Allo stesso tempo non facciamo nessuna ipotesi sulla velocità relativa dei processi.*

Il più semplice meccanismo per la risoluzione del problema della sezione critica è il lucchetto (MUTEX abbreviazione di MUTual Exclusion). Si usano i lucchetti per proteggere sezioni critiche e quindi prevenire le race conditions. Un processo deve acquisire il lucchetto prima di entrare in sezione critica; il processo rilascia poi il lucchetto quando esce dalla sezione critica.

**Nota**

- 👁 *Le istruzioni per l'acquisizione e il rilascio del lucchetto sono eseguite atomicamente, cioè senza possibilità di essere interrote. Così facendo ci si assicura che quando un processo sta facendo una richiesta di lucchetto, questa venga eseguita senza interferenze.*

```c
#include <stdio.h>
#include <assert.h>
#include <unistd.h>
#include <stdbool.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <pthread.h>
#include <math.h>

/*SEMPLICE ESEMPIO DI UTILIZZO DI fork(), getpid(), getppid()*/
/*-------------------------------------------------------------*/
int main1()
{
    printf("Programma avviato PID %d\n",getpid());
    pid_t id = fork();  //per il clonante, in id ci sarà il PID del processo figlio
                        //per il clonato, in id ci sarà 0
    printf("Clonazione eseguita PID %d\n",getpid());
    if(id == 0)
    {
        printf("Sono il clonato \tid:%d \t\tPID:%d \tPPID: %d\n", id, getpid(), getppid());
    }

    else
    {
        printf("Sono il clonante \tid:%d \tPID:%d \tPPID: %d\n", id, getpid(), getppid());

    }
}
/*-------------------------------------------------------------*/
/*-------------------------------------------------------------*/


/*ESEMPIO BASILARE DI VARIABILE CONDIVISA. L'UTILIZZO DELLA STRUCT NON È OBBLICATORIO */
/*MA RENDE IL CODICE PIÙ LEGGIBILE. VOLENDO USARE DIVERSE VARIABILI, SI DOVREBBERO    */
/*TUTTE CONDIVISE. IN QUESTO MODO, INVECE, VENGONO "RAGGRUPPATE" E GESTITE COME UNA   */
/*UNICA ENTITÀ. IN QUESTO ESEMPIO È POSSIBILE VEDERE GLI EFFETTI DELLE RACE CONDITIONS*/
/*-----------------------------------------------------------------------------------*/
typedef struct
{
    int valore;     //variabile condivisa
    int valori[5];  //array condiviso
    pthread_mutex_t lucchetto;
} risorsa;

static risorsa* r;
int main2()
{
    /*DICHIARO CONDIVISA LA VARIABILE*/
    r = mmap(NULL, sizeof(risorsa), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    assert(r);
    /*ASSEGNO UN VALORE ALLA VARIABILE*/
    r->valore = 6;

    printf("Programma avviato PID %d\n",getpid());
    pid_t id = fork();  //per il clonante, in id ci sarà il PID del processo figlio
                        //per il clonato, in id ci sarà 0
    printf("Clonazione eseguita PID %d\n",getpid());
    if(id == 0)
    {
        sleep(3);
        int tmp = r->valore + 1;
        printf("Sono il clonato \tid:%d \t\tPID:%d \tPPID: %d\n", id, getpid(), getppid());

        r->valore = tmp;
        printf("somma = [%d]\n", r->valore);

    }

    else
    {

        int tmp = r->valore + 3;
        printf("Sono il clonante \tid:%d \t\tPID:%d \tPPID: %d\n", id, getpid(), getppid());
        sleep(2);
        r->valore = tmp;
```

```c
        printf("Sono il clonante \tid:%d \tPID:%d \tPPID: %d\n", id, getpid(), getppid());
        printf("somma = [%d]\n", r->valore);
    }

    printf("Valore di r a fine esecuzione = [%d]\n", r->valore);

}
/*-------------------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------------------*/

/*ESEMPIO BASILARE DI ATTIVAZIONE DI PIÙ PROCESSI FIGLIO GESTITI DALLO STESSO PADRE. */
/*NELLA VARIABILE i SARÀ MEMORIZZATO L'IDENTIFICATIVO DEL FIGLIO. LA i NON VA        */
/*CONFUSA CON IL PID. IL PRIMO SERVE PER STABILIRE CHI FA COSA; IL SECONDO IDENTIFICA*/
/*IL PROCESSO IN MANIERA UNIVOCA A LIVELLO DI SISTEMA OPERATIVO.                     */
/*-------------------------------------------------------------------------------------*/
int main3()
{
    pid_t figli[10];
    int i = 0;
    for(i = 0; i < 10;i++)//ogni processo figlio "vedrà" in i un valore diverso
    {
        figli[i] = fork();
        if(figli[i] == 0) break;
    }
    printf("siamo in tanti...[%d]\n", getpid());
}
/*-------------------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------------------*/

/*ESEMPIO BASILARE DI ATTIVAZIONE DI PIÙ PROCESSI FIGLIO GESTITI DALLO STESSO PADRE. */
/*NELLA VARIABILE i SARÀ MEMORIZZATO L'IDENTIFICATIVO DEL FIGLIO. LA i NON VA        */
/*CONFUSA CON IL PID. IL PRIMO SERVE PER STABILIRE CHI FA COSA; IL SECONDO IDENTIFICA*/
/*IL PROCESSO IN MANIERA UNIVOCA A LIVELLO DI SISTEMA OPERATIVO. UTILIZZANDO i È     */
/*POSSIBILE FAR ESEGUIRE AI PROCESSI COMPITI DIVERSI.                                */
/*-------------------------------------------------------------------------------------*/
int main4()
{
    /*DICHIARO CONDIVISA LA VARIABILE*/
    r = mmap(NULL, sizeof(risorsa), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    assert(r);
    /*ASSEGNO UN VALORE ALLA VARIABILE*/
    r->valore = 0;
    pid_t figli[5];
    int i = 0;

    printf("Inizia tutto da qui [%d]\n", getpid());

    for(i = 0; i < 5;i++)
    {
        figli[i] = fork();
        if(figli[i] == 0) break;
    }

    printf("         tocca a me... %d [%d] <%d>\n", i, getpid(),r->valore);
    r->valore =r->valore + i;
    printf("                 %d [%d] <%d>\n", i, getpid(),r->valore);

//    int k;
//    for(k = 0; k < 5;k++)
//    {
//        waitpid(figli[i],NULL,NULL);
//    }
    if(i==5)
        printf("Solo il padre scrive dinuovo... %d [%d] <%d>\n", i, getpid(), r->valore);


}

/*ESEMPIO BASILARE DI UTILIZZO DELLA FUNZIONE wait() */
/*NELLA VARIABILE i SARÀ MEMORIZZATO L'IDENTIFICATIVO*/
/*DEL FIGLIO. IL PADRE ATTENDERÀ CHE TUTTI I FIGLI   */
/*ABBIANO TERMINATO LA LORO ESECUZIONE E POI ESEGUIRÀ*/
/*L'ULTIMA ISTRUZIONE                                */
/*-------------------------------------------------------------------------------------*/
```

```c
int main4a()
{
    /*DICHIARO CONDIVISA LA VARIABILE*/
    r = mmap(NULL, sizeof(risorsa), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    assert(r);
    /*ASSEGNO UN VALORE ALLA VARIABILE*/
    r->valore = 0;
    pid_t figli[5];
    int i = 0;

    printf("Inizia tutto da qui [%d]\n", getpid());

    for(i = 0; i < 5;i++)
    {
        figli[i] = fork();
        if(figli[i] == 0) break;
    }

    printf("         tocca a me... %d [%d] <%d>\n", i, getpid(),r->valore);
    r->valore =r->valore + i;
    printf("                 %d [%d] <%d>\n", i, getpid(),r->valore);

    if(i==5)
    {
        int k;
        for(k = 0; k < 5;k++)
        {
            waitpid(figli[i],NULL,NULL);
        }

        printf("Hanno finito tutti, ora finisco io!!! [%d]\n", getpid());

    }

}
/*-------------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------------*/

/*ESEMPIO BASILARE DI UTILIZZO DELLA FUNZIONE wait() PER LA SINCRONIZZAZIONE DI PROCESSI */
/*CHE COOPERANO PER ESEGUIRE UN COMPITO COMPLESSO. FA RIFERIMENTO ALL'ESEMPIO DELLE SLIDE*/
/*IN CUI BISOGNA IMPLEMENTARE UN SISTEMA DI CALCOLO PARALLELO PER LA SEMPLICE SOMMA      */
/* 3x5 + 7x2 + 8x4 + 9x10                                                              */
/*L'ARRAY CONDIVISO VIENE UTILIZZATO PER MEMORIZZARE I VALORI INTERMEDI DEL CALCOLO. OGNI*/
/*PROCESSO UTILIZZERÀ IL PROPRIO i PER FARE I CALCOLI. IL PROCESSO CHE DEVE UTILIZZARE UN*/
/*CALCOLO INTERMEDIO DOVRÀ ATTENDERE CHE UN ALTRO PROCESSO TERMINI PRIMA DI UTILIZZARE IL*/
/*RISULTATO DEL CALCOLO                                                               */
/*-------------------------------------------------------------------------------------*/
int main5()
{
    int N = 3;
    /*DICHIARO CONDIVISA LA VARIABILE*/
    r = mmap(NULL, sizeof(risorsa), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    assert(r);
    /*ASSEGNO UN VALORE ALLA VARIABILE*/
    r->valore = 0;
    pid_t figli[N];
    int i = 0;

    printf("Inizia tutto da qui [%d]\n", getpid());

    for(i = 0; i < N;i++)
    {
        figli[i] = fork();
        if(figli[i] == 0) break;
    }

    if (i == 3)
    {
        //esegui calcolo r = 3*5
        r->valori[3] = 3 * 5;
        printf("[%d] parziale <%d>\n", getpid(), r->valori[3]);

        //attendi risultato di i = 2 (r2)
        waitpid(figli[2], NULL, NULL);
```

```c
        //esegui calcolo r += r2
        r->valori[3] += r->valori[2];
        printf("[%d] parziale <%d>\n", getpid(), r->valori[3]);

        //attendi risultato di i = 1 (r1)
        waitpid(figli[1], NULL, NULL);

        //esegui calcolo r += r1
        r->valori[3] += r->valori[1];
        printf("[%d] definitivo <%d>\n", getpid(), r->valori[3]);


    }

    if (i == 2)
    {
        //esegui calcolo r = 7*2
        //condividi risultato
        r->valori[2] = 7*2;
        printf("    [%d] parziale <%d>\n", getpid(), r->valori[2]);

    }

    if (i == 1)
    {
        //esegui calcolo r = 7*2
        r->valori[1] = 8*4;
        printf("    [%d] parziale <%d>\n", getpid(), r->valori[1]);

        //attendi risultato di i = 0 (r0)
        waitpid(getpid()+1);

        //esegui calcolo r += r0
        //condividi risultato
        r->valori[1] += r->valori[0];
        printf("        [%d] parziale <%d>\n", getpid(), r->valori[1]);


    }

    if (i == 0)
    {
        //esegui calcolo r = 7*2
        //condividi risultato
        r->valori[0] += 9*10;
        printf("    [%d] parziale <%d>\n", getpid(), r->valori[0]);

    }
    return 0;

}
/*-------------------------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------------------------*/

/*ESEMPIO DI SEZIONE CRITICA*/
typedef struct
{
    int valore;     //variabile condivisa
    pthread_mutex_t lucchetto;
} risorsa_tipo_1;

typedef struct
{
    int valore;     //variabile condivisa
    pthread_mutex_t lucchetto;
} risorsa_tipo_2;

typedef struct
{
    int valore;     //variabile condivisa
    pthread_mutex_t lucchetto;
} risorsa_tipo_3;

static risorsa_tipo_1* r1;
static risorsa_tipo_2* r2;
```

```c
static risorsa_tipo_3* r3;
int main6()
{
    int N = 3;

    r1 = mmap(NULL, sizeof(risorsa_tipo_1), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1,
0);
    assert(r1);

    pthread_mutexattr_t attr1;
    pthread_mutexattr_init(&attr1);
    pthread_mutexattr_setpshared(&attr1, PTHREAD_PROCESS_SHARED);
    pthread_mutex_init(&r1->lucchetto, &attr1);

    int i = 0;
    pid_t id;
    for(;i < 2; i++)
    {
        id = fork();
        if(id==0) break;
    }

    printf("[%d] Cerco di ottere R1...\n",getpid());
    pthread_mutex_lock(&r1->lucchetto);
        printf("[%d] R1 ottenuta...\n",getpid());
        printf("    [%d] posso eseguire la sezione critica...\n",getpid());
        sleep(2);
        printf("[%d] rilascio R1...\n",getpid());
    pthread_mutex_unlock(&r1->lucchetto);

    if(i==2) //SE SONO IL PADRE ATTENDO LA TERMINAZIONE DEI FIGLI
    {

        waitpid(getpid()+1,NULL, NULL);
        waitpid(getpid()+2,NULL, NULL);
    }


}
/*-------------------------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------------------------*/


/*ESEMPIO DI SEZIONE CRITICA*/
int main6a()
{
    int N = 3;

    r1 = mmap(NULL, sizeof(risorsa_tipo_1), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1,
0);
    assert(r1);

    r2 = mmap(NULL, sizeof(risorsa_tipo_3), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1,
0);
    assert(r1);

    r3 = mmap(NULL, sizeof(risorsa_tipo_2), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1,
0);
    assert(r3);

    pthread_mutexattr_t attr1;
    pthread_mutexattr_init(&attr1);
    pthread_mutexattr_setpshared(&attr1, PTHREAD_PROCESS_SHARED);
    pthread_mutex_init(&r1->lucchetto, &attr1);

    pthread_mutexattr_t attr2;
    pthread_mutexattr_init(&attr2);
    pthread_mutexattr_setpshared(&attr2, PTHREAD_PROCESS_SHARED);
    pthread_mutex_init(&r2->lucchetto, &attr2);

    pthread_mutexattr_t attr3;
    pthread_mutexattr_init(&attr3);
    pthread_mutexattr_setpshared(&attr3, PTHREAD_PROCESS_SHARED);
    pthread_mutex_init(&r3->lucchetto, &attr3);
```

```
    int i = 0;
    pid_t id;
    for(;i < 2; i++)
    {
        id = fork();
        if(id==0) break;
    }

    if(i==2)
    {
//        printf("[%d] Cerco di ottere R3...\n",getpid());
//        pthread_mutex_lock(&r3->lucchetto);
//            printf("[%d] R3 ottenuta...\n",getpid());
//            printf("[%d] Cerco di ottere R1...\n",getpid());
//            pthread_mutex_lock(&r1->lucchetto);
//                printf("[%d] R1 ottenuta...\n",getpid());
//                printf("    [%d] posso eseguire la sezione critica...\n",getpid());
//                sleep(2);
//                printf("[%d] Rilascio R1...\n",getpid());
//            pthread_mutex_unlock(&r1->lucchetto);
//            printf("[%d] rilascio R3...\n",getpid());
//        pthread_mutex_unlock(&r3->lucchetto);

        waitpid(getpid()+1,NULL, NULL);
        waitpid(getpid()+2,NULL, NULL);

    }

//    if(i==1)
//    {
//        printf("[%d] Cerco di ottere R3...\n",getpid());
//        pthread_mutex_lock(&r3->lucchetto);
//            printf("[%d] R3 ottenuta...\n",getpid());
//            printf("    [%d] posso eseguire la sezione critica...\n",getpid());
//            sleep(2);
//            printf("[%d] rilascio R3...\n",getpid());
//        pthread_mutex_unlock(&r3->lucchetto);
//
//
//    }
//
//    if(i==0)
    {
        printf("[%d] Cerco di ottere R2...\n",getpid());
        pthread_mutex_lock(&r2->lucchetto);
            printf("[%d] R2 ottenuta...\n",getpid());
            printf("    [%d] posso eseguire la sezione critica...\n",getpid());
            sleep(2);
            printf("[%d] rilascio R2...\n",getpid());
        pthread_mutex_unlock(&r2->lucchetto);


    }
}
/*-------------------------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------------------------*/



int main()
{
//    main1();
//    main2();
//    main3();
//    main4a();
//    main5();
    main6();
    return 0;
}
```