Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2018-2019

Pietro Frasca

Lezione 12

Giovedì 15-11-2018

Un esempio di sincronizzazione tra thread

- Risolviamo il classico problema del produttore-consumatore.
- Nel caso più generale, più produttori e consumatori possono utilizzare un buffer in grado di contenere, al massimo, *N* messaggi.
- Ricordiamo che i vincoli per accedere al buffer sono due:
  - il produttore non può inserire un messaggio nel buffer se è pieno;
  - il consumatore non può prelevare un messaggio dal buffer vuoto.
- Supponendo, ad esempio, che i messaggi siano dei valori interi, realizziamo il buffer come un vettore di interi, gestendolo in modo circolare. Per gestire il buffer associamo ad esso le seguenti variabili:

- **cont**, il numero dei messaggi contenuti nel buffer;
- **scrivi**, indice del prossimo elemento da scrivere;
- **leggi**, indice del prossimo elemento da leggere.

Essendo il buffer una risorsa condivisa è necessario associare ad esso un **mutex *M*** per controllarne l'accesso in mutua esclusione.

Oltre al vincolo di mutua esclusione, i thread produttori e consumatori devono rispettivamente sospendersi nel caso in cui il buffer sia pieno o sia vuoto. Per realizzare tale sospensione associamo al buffer due variabili condizione di nome **PIENO,** per la sospensione dei produttori se il buffer è pieno, e di nome **VUOTO,** per la sospensione dei consumatori se il buffer è vuoto.

Da quanto detto rappresentiamo il buffer con un tipo dato struttura che chiameremo **buffer_t**:

```
typedef struct {
   int messaggio[DIM];
   pthread_mutex_t M;
   int leggi, scrivi;
   int cont;
   pthread_cond_t PIENO;
   pthread_cond_t VUOTO;
} buffer_t ;
```

- Per gestire una struttura di tipo buffer, definiamo inoltre tre funzioni:
  - **Init,** per inizializzare la struttura tipo *buffer_t*,
  - **produci**, funzione eseguita da un thread produttore per inserire un messaggio nel buffer
  - **consuma**, funzione eseguita da un thread consumatore per prelevare un messaggio dal buffer.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define FINE (-1)
#define MAX 20
#define DIM 10

typedef struct {
   pthread_mutex_t M;
   pthread_cond_t PIENO;
   pthread_cond_t VUOTO;
   int messaggio [DIM];
   int leggi, scrivi;
   int cont ;
} buffer_t;

buffer_t buf;
```

```
void init (buffer_t *buf);
void produci (buffer_t *buf, int mes);
int consuma (buffer_t *buf);

void init (buffer_t *buf){
   pthread_mutex_init (&buf->M,NULL);
   pthread_cond_init (&buf->PIENO, NULL);
   pthread_cond_init (&buf->VUOTO, NULL);
   buf->cont=0;
   buf->leggi=0;
   buf->scrivi=0;
}
```

```
void produci (buffer_t *buf, int mes) {
   pthread_mutex_lock (&buf->M);
   if (buf->cont==DIM)   /* il buffer e' pieno? */
       pthread_cond_wait (&buf->PIENO, &buf->M);
   /* scrivi mes e aggiorna lo stato del messaggio */
   buf->messaggio[buf->scrivi]=mes;
   buf->cont++;
   buf->scrivi++;
   /* la gestione del buffer è circolare */
   if (buf->scrivi==DIM) buf->scrivi=0;
   /* risveglia un eventuale thread consum. sospeso */
  pthread_cond_signal (&buf->VUOTO);
  pthread_mutex_unlock (&buf->M);
}
```

```
int consuma (buffer_t *buf){
   int mes;
   pthread_mutex_lock(&buf->M);
   if (buf->cont==0) /* il buffer e' vuoto? */
       pthread_cond_wait (&buf->VUOTO, &buf->M);
   /* Leggi il messaggio e aggiorna lo stato del buffer*/
   mes = buf->messaggio[buf->leggi];
   buf->cont--;
   buf->leggi++;
   /* la gestione è circolare */
   if (buf->leggi>=DIM) buf->leggi=0;
   /* Risveglia un eventuale thread produttore */
   pthread_cond_signal(&buf->PIENO) ;
   pthread_mutex_unlock(&buf->M);
  return mes;
}
```

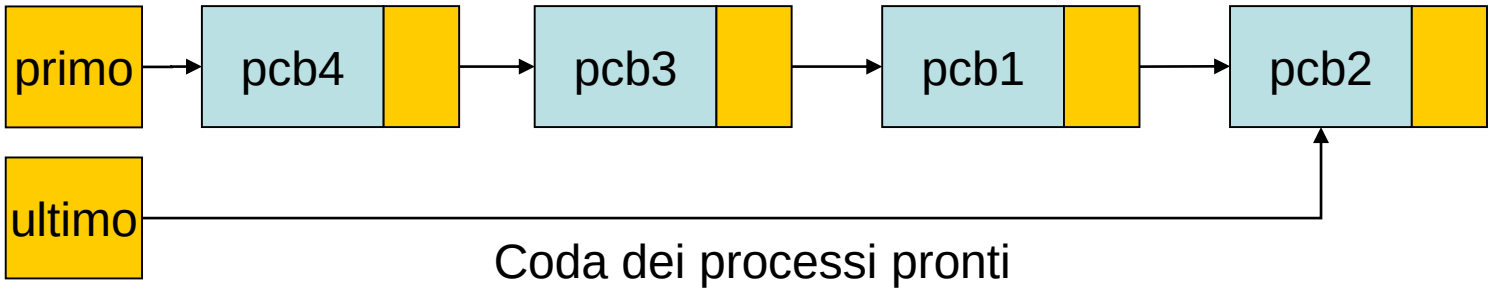
```
void *produttore (void *arg){
   int n;
   for (n=0; n<MAX; n++){
        printf ("produttore %d -> %d\n", (int)arg,n);
        produci (&buf, n);
        sleep(1);
   }
   produci (&buf, FINE);
}
void *consumatore (void *arg){
   int d;
   while (1){
        d=consuma (&buf) ;
        if (d == FINE) break;
        printf ("      %d <- consumatore %d\n",d,(int)arg);
        sleep(2);
   }
}
```

```
int main () {
   int i;
   int nprod=1,ncons=1;
   pthread_t prod[nprod], cons[ncons];
   init (&buf);
   /*Creazione thread */
   for (i=0;i<nprod;i++)
       pthread_create(&prod[i], NULL, produttore,
   (void*)i);
   for (i=0;i<ncons;i++)
       pthread_create(&cons[i], NULL, consumatore,
   (void*)i);
   /* Attesa teminazione thread creati */
   for (i=0;i<nprod;i++)
       pthread_join (prod[i], NULL);
   for (i=0;i<ncons;i++)
       pthread_join (cons[i], NULL);
   return 0;
}
```

Scheduling

Scheduling a breve termine

- Lo **scheduler a breve termine** (short term scheduler) è il componente del SO che si occupa di selezionare, dalla coda di pronto, il processo a cui assegnare la CPU.
- Lo scheduler è eseguito molto frequentemente e deve essere quindi realizzato in modo molto efficiente in termini di velocità d'esecuzione.
- Spesso si indica con il termine **scheduler** la parte che implementa le politiche (strategie) mentre il componente che implementa i meccanismi (cambio di contesto) prende il nome di **dispatcher**.



- In alcuni sistemi operativi, oltre lo scheduling a breve termine, sono previsti altri livelli di scheduling
  - **Scheduling a lungo termine (long term scheduling)**
  - **Scheduling a medio termine (medium term scheduler)**

Scheduling a lungo termine

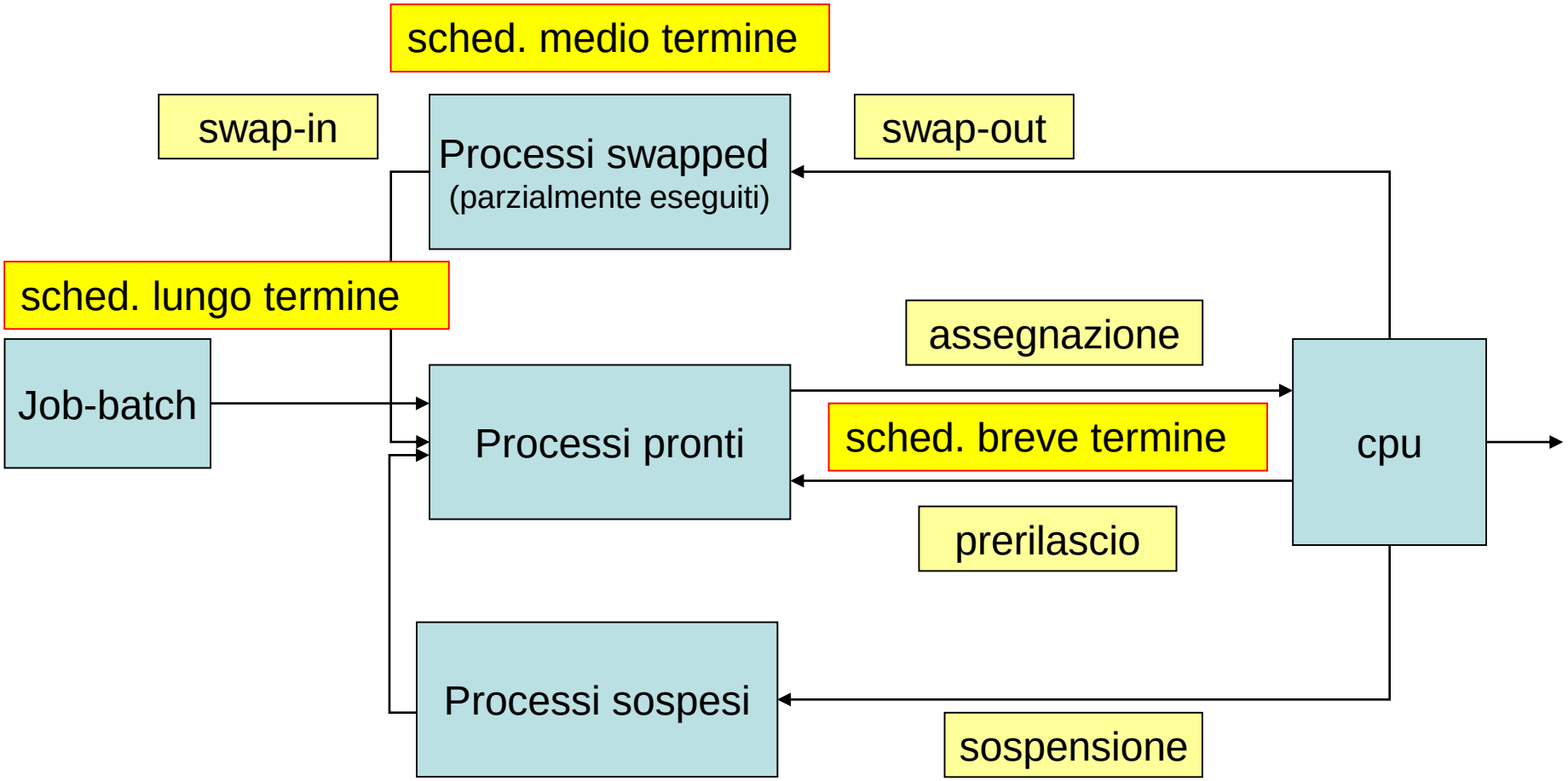
- Nei sistemi batch è la funzione del SO che provvede a scegliere i programmi memorizzati in memoria secondaria da trasferire in memoria principale, da inserire nella coda dei processi pronti e quindi essere eseguiti.
- La selezione è eseguita in modo da bilanciare la presenza nella coda di pronto di processi di tipo **CPU-bound** e processi di tipo **I/O-bound,** per evitare che una prevalenza di uno dei due tipi di processo porti ad un uso non ottimo della CPU e delle risorse.

- Un altro importante compito dello scheduler a lungo termine è di controllare il **grado di multiprogrammazione**, cioè il numero di processi che sono presenti in memoria principale nello stesso tempo.

Scheduling a medio termine

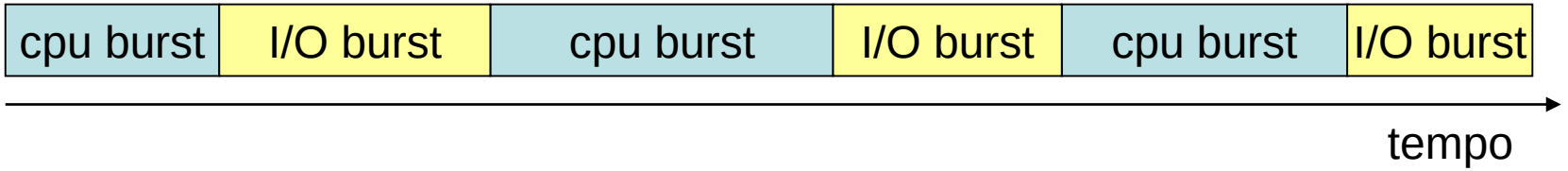
- Lo **scheduling a medio termine** (medium term scheduling) si occupa di trasferire temporaneamente processi dalla memoria ram alla memoria secondaria (dischi), operazione di **swap-out** e viceversa, operazione di **swap-in**.
- Lo scheduler a lungo termine e lo scheduler a medio termine sono eseguiti con frequenze molto inferiori rispetto a quella dello scheduler a breve termine, in quanto sono molto complesse.

livelli di scheduling



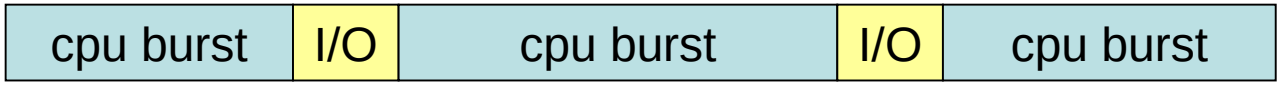
Comportamento dei processi

- Durante la sua attività, un processo generalmente alterna le seguenti fasi**:**
  - **CPU burst**: fase in cui viene impiegata soltanto la CPU senza I/O;
  - **I/O burst:** fase in cui il processo effettua input/output da/verso una risorsa(dispositivo) del sistema.
- Il termine burst (raffica) indica una sequenza di istruzioni.

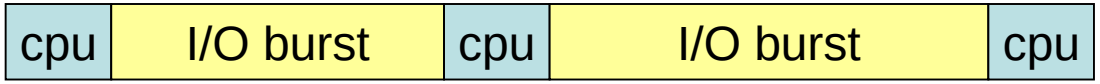


- Quando un processo esegue I/O burst, non utilizza la CPU. In un sistema multiprogrammato, lo scheduler assegna la CPU a un nuovo processo.

- I processi, in base al tipo di istruzioni che eseguono si classificano in:
  - Processi ***CPU-bound (compute-bound)***: se eseguono prevalentemente istruzioni di computazione; CPU burst di *lunga* durata, intervallati da pochi I/O burst di *breve* durata.



  - Processi ***I/O-bound***: se eseguono prevalentemente istruzioni di I/O; CPU burst di *breve* durata, intervallati da I/O burst di *lunga* durata.



- Gli algoritmi di scheduling si possono classificare in due categorie:
  - **con prelazione** (***pre-emptive***): al processo in esecuzione può essere revocata la CPU. Il SO può, in base a determinati criteri, sottrarre ad esso la CPU per assegnarla ad un nuovo processo.
  - **senza prelazione** (***non pre-emptive***): la CPU rimane allocata al processo in esecuzione fino a quando esso si sospende volontariamente (ad esempio, per I/O), o termina.
- Ad esempio, i sistemi a divisione di tempo (*time sharing*) hanno uno scheduling ***pre-emptive.***
- Gli algoritmi **non pre-emptive** sono più semplici, e richiedono minor overhead di sistema in quanto effettuano meno cambi di contesto. D'altra parte sono meno flessibili in quanto offrono una minore gamma di strategie di scheduling. Sono adatti per sistemi batch ma non per sistemi time-sharing.
- Le prime versioni dei sistemi windows (fino alla versione 3.1) utilizzavano algoritmi non pre-emptive. Dalla versione windows 95 fu usato uno scheduler con revoca.
- Linux e Unix adottano algoritmi pre-empitive.

Parametri di scheduling

- Per analizzare e confrontare i diversi algoritmi di scheduling, si considerano i seguenti parametri, i primi tre relativi ai processi e gli altri relativi al sistema:
  - **Tempo di attesa,** è la quantità di tempo che un processo trascorre nella coda di pronto, in attesa della CPU.
  - **Tempo di completamento (Turnaround time),** è l'intervallo di tempo che passa tra l'avvio del processo e il suo completamento.
  - **Tempo di risposta,** è l'intervallo di tempo tra avvio del processo e l'inizio della prima risposta.
  - **Utilizzo della CPU,** esprime la percentuale media di utilizzo della CPU nell'unità di tempo.
  - **Produttività (Throughput rate)** (del sistema), esprime il numero di processi completati nell'unità di tempo.

- Generalmente i parametri che devono essere massimizzati sono:
  - **Utilizzo della CPU (**al massimo: 100%)
  - **Produttività (Throughput)**

  invece, devono essere **minimizzati**:
  - **Tempo medio di completamento (Turnaround)** (sistemi *batch*)
  - **Tempo medio di attesa**
  - **Tempo medio di risposta** (sistemi *interattivi*)

- Non è possibile ottenere tutti gli obiettivi contemporaneamente.

- A seconda del tipo di SO, gli algoritmi di scheduling possono avere **diversi obiettivi**; tipicamente:
  - nei sistemi **batch**:
    - **massimizzare il throughput** e **minimizzare il tempo medio di completamento**
  - nei sistemi **interattivi**:
    - **minimizzare il tempo medio di risposta** dei processi
    - **minimizzare il tempo medio di attesa** dei processi
- Nei sistemi real-time, l'algoritmo con revoca non sempre è necessario dato che i programmi real-time sono progettati per essere eseguiti per brevi periodi di tempo e poi si bloccano.

Principali algoritmi di scheduling

- Alcuni algoritmi sono usati sia nei sistemi batch che nei sistemi interattivi.

FCFS

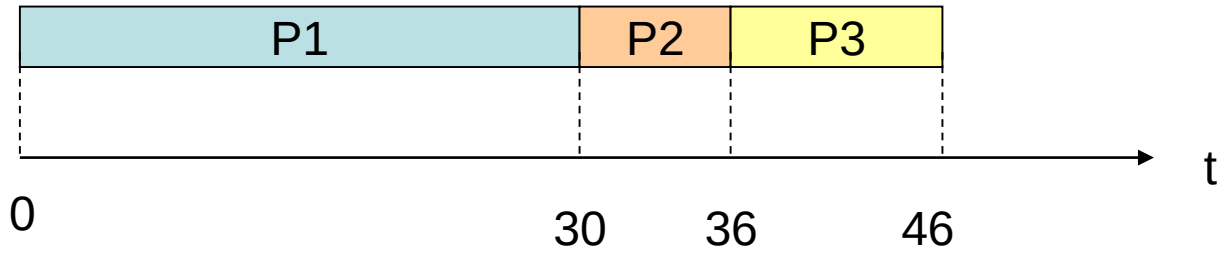
- FCFS (First Came First Served) è l'algoritmo più semplice. La CPU è assegnata ai processi seguendo l'ordine con cui essa è stata richiesta, ovvero la cpu è assegnata al processo che è in attesa da più tempo nella coda di pronto.
- Quando un processo acquisisce la CPU, resta in esecuzione fino a quando si blocca volontariamente o termina. Quando un processo in esecuzione si blocca si seleziona il primo processo presente nella coda di pronto. Quando un processo da bloccato ritorna pronto, è accodato nella coda di pronto.
- E' un algoritmo senza diritto di prelazione.

- E’ inefficiente nel caso in cui ci sono molti processi che si sospendono frequentemente o nel caso di sistemi interattivi. Il tempo di attesa risulterebbe troppo lungo

Esempio

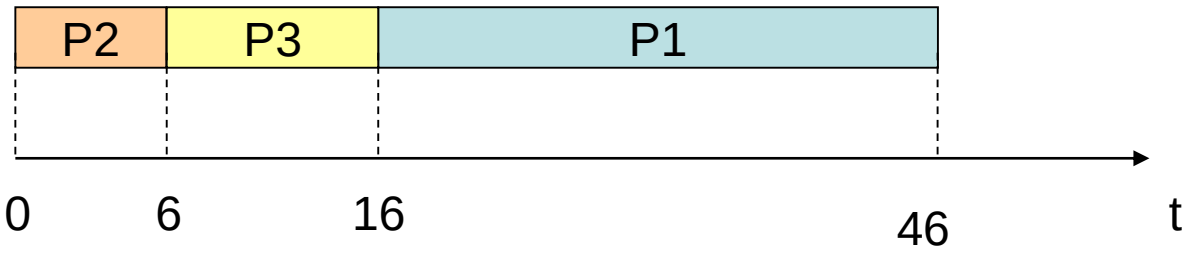
Calcoliamo il tempo medio di attesa di tre processi P1, P2 e P3 avviati allo stesso istante e aventi rispettivamente i tempi (in millisecondi) di completamento pari a: T1=30, T2=6 e T3=10.

Il diagramma temporale (di Gantt) è il seguente.



Tempo $_{\text{attesa medio}}$ = (0+30+36)/3 =22

- Se cambiassimo l'ordine di scheduling nel seguente: {P2, P3, P1} il tempo medio di attesa passerebbe da 22 a 7,33.



Tempo $_{\text{attesa medio}}$ = (0+6+16)/3 = 7,33

- Ma il FCFS non consente di cambiare l'ordine dei processi. Quindi nel caso in cui ci siano processi con brevi CPU-burst (processi I/O-bound) in attesa dietro a processi con lunghi CPU-burst (processi CPU-bound), il tempo di attesa sarà alto.
- Inoltre, in considerazione delle prestazioni del FCFS in una situazione dinamica, supponiamo di avere un processo CPU-bound e molti processi I/O-bound e si verifichi il seguente scenario. Il processo CPU-bound ottiene la CPU.

- Durante questo tempo, tutti gli altri processi terminano i loro I/O e si spostano nella coda di pronto, in attesa di ottenere la CPU. Mentre i processi sono in attesa nella coda di pronto, i dispositivi di I/O sono inattivi. Successivamente, il processo CPU-bound termina il suo burst ed è traferito nella coda di un dispositivo di I/O. Tutti i processi I/O-bound, che hanno brevi burst di CPU, eseguono le loro sequenze d'istruzioni in modo rapido e tornano nelle code di I/O. A questo punto, la CPU rimane inattiva. Il processo CPU-bound sarà poi posto di nuovo nella coda pronto e tornerà in esecuzione. Anche in questo caso, tutti i processi di I/O saranno rimessi nella coda di pronto fino al termine del burst del processo CPU-bound. Vi è un ***effetto convoglio,*** tutti gli altri processi aspettano che un processo con un lungo CPU-burst lasci la CPU. Questo effetto porta a un utilizzo, sia della CPU che dei dispositivi, inferiore a quanto possa essere se i processi più *brevi* fossero eseguiti prima dei processi più *lunghi*.

- La figura mostra una situazione in cui si verifica l'effetto convoglio. In questo esempio, P1 è un processo CPU-bound; P2, P3 e P4 sono processi I/O-bound. P1 effettua operazioni di I/O nell'intervallo [t1, t3]. Si ha che nell'intervallo [t2, t3] la CPU è inattiva.
- Nella figura sono mostrati anche gli intervalli di tempo in cui lo scheduler va in esecuzione.