Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2021-2022

Pietro Frasca

Lezione 9

Martedì 9-11-2021

I thread nello standard POSIX: la libreria pthreads

- La maggior parte dei sistemi operativi supporta i *thread a livello kernel* e pertanto il thread è l'unità di scheduling*.*
- Come già descritto, il processi hanno uno spazio di indirizzamento privato e quindi non possono condividere dati tra loro. I processi possono scambiarsi dati mediante messaggi o allocando segmenti di memoria condivisa che dovranno essere gestiti in mutua esclusione (mediante opportune system call).
- Il *thread,* invece, possono condividere tra loro lo spazio di indirizzi cui essi appartengono.
- Per realizzare applicazioni con i thread in POSIX si può utilizzare la libreria ***pthreads****.*
- La libreria pthread definisce il tipo **pthread_t** per crare i thread all'interno di processi concorrenti (la definizione è contenuta nel file header **pthread.h**).

- Lo standard POSIX prevede che i thread siano creati all'interno di un processo. In particolare, al codice della function main (nel caso del linguaggio C) corrisponde il thread iniziale.
- Il thread iniziale può generare, attraverso chiamate di sistema, nuovi thread che possono condividere uno spazio di indirizzi (ad esempio variabili globali e function).
- Generalmente, per compilare con ***gcc*** un'applicazione che usa lo standard pthread, è necessario specificare l'opzione ***–lpthread***. Ad esempio, ***gcc pro.c –o pro –lpthread*** compila il file sorgente ***pro.c*** creando il file eseguibile di nome ***pro****.*

Creazione di thread

La creazione di un *thread* si ottiene mediante la chiamata di sistema **pthread_create**:

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine) (void *),
    void *arg);
```

- *thread* : è il puntatore alla variabile di tipo **pthread_t** del nuovo thread;
- *attr* : può essere usato per specificare eventuali attributi da assegnare al thread come ad esempio la priorità del thread, oppure NULL (valori di default).
- *start_routine* : è il puntatore alla funzione che contiene il codice iniziale del thread creato;

- *arg* è il puntatore all'eventuale vettore contenente i valori dei parametri da passare alla funzione *start_routine*.
- La chiamata **pthread_create** restituisce
  - **0 in caso di successo**
  - **oppure un codice di errore**.
- Ogni nuovo *thread* va in esecuzione in modo concorrente con il thread *genitore* e **condivide** con esso lo stesso spazio di indirizzamento del processo nel quale è definito.

Terminazione di un thread

- Un thread può terminare chiamando *pthread_exit()*:

```
#include <pthread.h>
void pthread_exit (void *stato) ;
```

La funzione ***pthread_exit()*** fa terminare il thread chiamante e restituisce un valore, tramite il parametro ***stato,*** che è disponibile ad un altro thread nello stesso processo che chiama la funzione ***pthread_join()***.

- Quando un thread termina, le risorse condivise di processo (es. variabili, descrittori di file) non vengono rilasciate.
- Dopo che l'ultimo thread in un processo termina, il processo termina come chiamando la exit*()* con uno stato di uscita zero; quindi, le risorse condivise nel processo vengono rilasciate.

Unione di thread

- Un thread può attendere la terminazione di un altro thread chiamando la funzione ***pthread_join ().***

```
int pthread_join(pthread_t thread, void *stato);
```

- Il thread che chiama la ***pthread_join()*** attende che il thread specificato dal parametro *thread sia* terminato. Se questo thread è già terminato, ***pthread_join()*** ritorna immediatamente. Se il parametro *stato* non è NULL, ***pthread_join()*** copia nella variabile puntata da *stato* il valore di uscita del thread terminato, cioè il valore che questo ha fornito alla funzione ***pthread_exit()***.
- Se più thread tentano simultaneamente di *unirsi* con lo stesso thread, cioè chiamano la pthread_join () specificando nel parametro lo stesso thread, i risultati sono indefiniti.

Esempio

```
/* Semplice struttura di un'applicazione multithread. Il
   thread main crea un insieme di thread (nell'esempio 2)
   e attende la loro terminazione. Ciascun thread inizia
   la sua esecuzione dalla funzione indicata nel terzo
   parametro della system call pthread_create().
*/

#include <pthread.h>
int a=10;              /* variabili globali condivise
char buffer[1024];        tra i thread */
void *codice_Th1 (void *arg){
  <CODICE DI TH1>
  pthread_exit(0);
}
void *codice_Th2 (void *arg){
  <CODICE DI TH2>
  pthread_exit(0);
}
```

```
int main(){
  pthread_t th1, th2;
  int ret;
  // creazione e attivazione del primo thread
  if (pthread_create(&th1, NULL, codice_Th1, "Lino")!=0){
    fprintf(stderr,"Errore di creazione thread 1 \n");
    exit(1);
  }
  // creazione e attivazione del secondo thread
  if (pthread_create(&th2,NULL,codice_Th2, "Eva")!=0){
    fprintf(stderr,"Errore di creazione thread 2 \n");
    exit(1);
  }
  // attesa della terminazione del primo thread
  ret=pthread_join(th1,NULL);
  if (ret !=0)
    fprintf(stderr,"join fallito %d \n",ret);
  else
    printf("terminato il thread 1 \n");
```

```
// attesa della terminazione del secondo thread
   ret=pthread_join(th2,NULL);
  if (ret !=0)
    fprintf(stderr,"join fallito %d \n",ret);
  else
    printf("terminato il thread 2 \n");
}
```

Esempio

Il thread main crea due thread th1 e th2 e attende la loro terminazione. I due thread eseguono concorrentemente lo stessa funzione codice_T.

```
1.   #include <stdio.h>
2.   #include <stdlib.h>
3.   #include <pthread.h>
4.   //variabili condivise:
5.   char MSG [] ="Ciao!";
6.   void *codice_T (void *arg){
7.     int i;
8.     for (i=0; i<5;i++) {
9.       printf ("Thread %s: %s\n“, (char*)arg, MSG);
10.      sleep(1) ; /* sospensione per 1 secondo. */
11.    }
12.    pthread_exit (0) ;
13.  }
```

```
1.   int main() {
2.     pthread_t th1, th2;
3.     int ret;
4.     /* creazione primo thread: */
5.     if (pthread_create(&th1,NULL,codice_T, "1") < 0) {
6.       fprintf (stderr, "Errore di creazione thread 1\n");
7.       exit(1);
8.     }
9.     /* creazione secondo thread: */
10.    if (pthread_create(&th2,NULL,codice_T, "2") < 0) {
11.      fprintf (stderr, "Errore di creazione thread 2\n");
12.      exit(1);
13.    }
14.    ret = pthread_join(th1,NULL);
15.    if (ret != 0)
16.      fprintf (stderr, "join fallito %d \n“, ret);
17.    else printf ("terminato il thread 1 \n) ;
18.    ret = pthread_join(th2,NULL);
19.    if (ret != 0)
20.      fprintf (stderr, "join fallito %d\n“, ret) ;
21.    else printf("terminato il thread 2\n);
22.    return 0;
23.  }
```

Sincronizzazione tra processi/thread

- Come descritto in precedenza, i processi o i thread possono interagire tra loro in due modi: ***cooperazione*** **e** ***competizione****.*

Cooperazione

- Generalmente la cooperazione tra processi avviene mediante operazioni di sincronizzazione e comunicazione.
- Il modello *produttore-consumatore* è molto usato per la comunicazione tra processi. In questo paradigma due processi, l'uno detto ***produttore*** produce messaggi e li scrive in un buffer comune e l'altro detto ***consumatore*** legge i messaggi dal buffer e li elabora (consuma).
- Generalmente i processi per cooperare devono sincronizzare le loro attività nel tempo.

Competizione

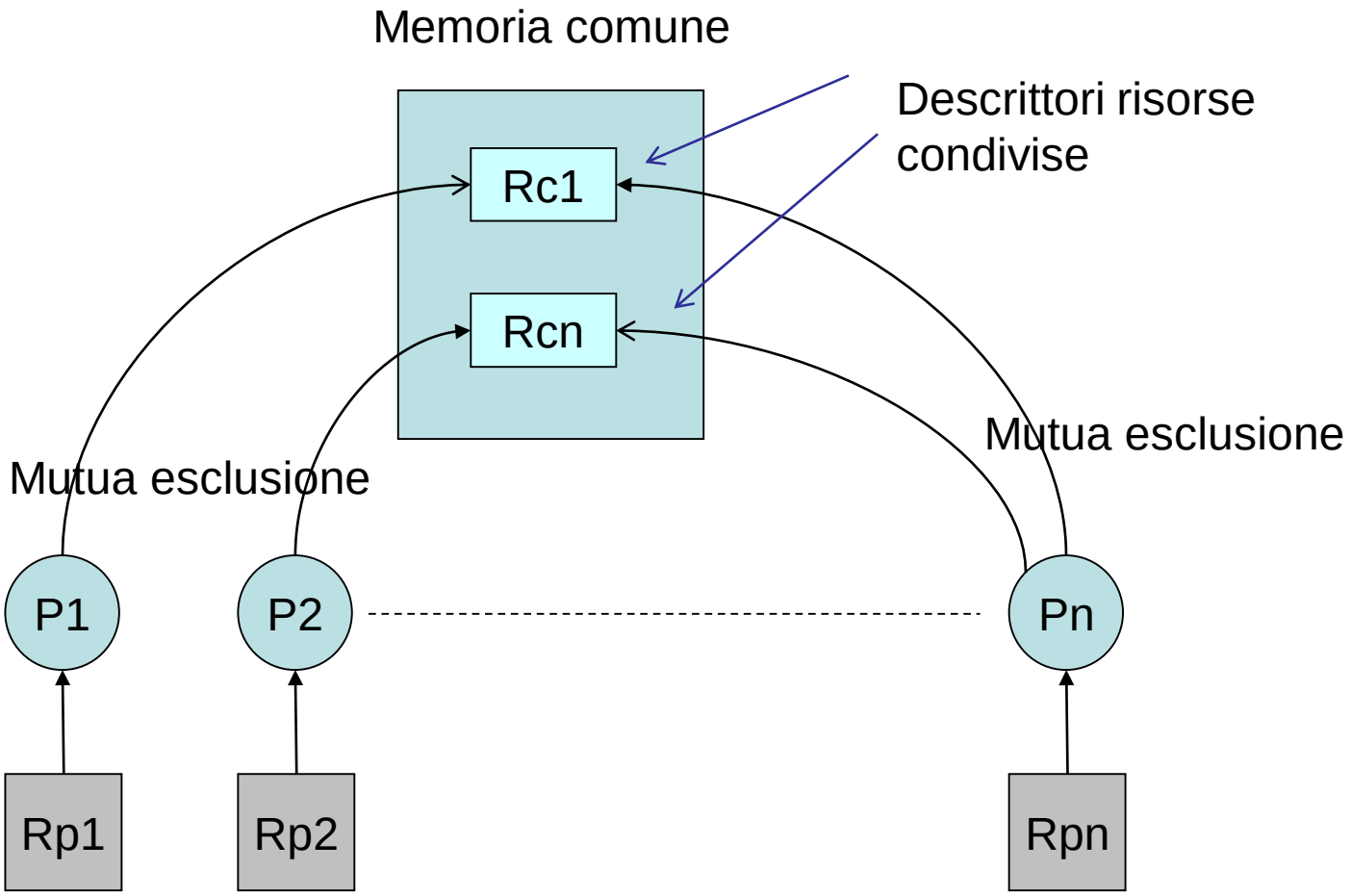
- La competizione si ha quando i processi richiedono risorse comuni che non possono essere usate contemporaneamente, come ad esempio una struttura dati, un file o un dispositivo.
- Sia per la cooperazione che per la competizione, è necessario che le operazioni eseguite dai processi sulle **risorse comuni** siano effettuate in ***mutua esclusione nel tempo***.
- L'interazione tra processi si ottiene mediante diversi **strumenti di sincronizzazione** la cui scelta dipende dal tipo di **modello di interazione** tra i processi:

  - **Modello a memoria condivisa (ambiente globale)**
  - **Modello a scambio di messaggi (ambiente locale)**

Modello a memoria condivisa (ambiente globale)

- L’interazione, sia competizione che cooperazione, tra i processi avviene tramite memoria condivisa.
- Generalmente, questo modello è usato in architetture di calcolatori sia con un solo processore che multiprocessore. In quest’ultimo caso i processori sono collegati e condividono un’unica memoria nella quale saranno allocate le risorse comuni.
- Ogni risorsa è rappresentata con una struttura dati che è allocata in un area di memoria condivisa. Ad esempio, un dispositivo di I/O, è rappresentato da un struttura dati detta ***descrittore del dispositivo*** che rappresenta le sue proprietà e il suo stato.
- Per un processo, una risorsa può essere **privata o condivisa (o globale)**. Nel primo caso il solo processo proprietario può operare su di essa, mentre nel secondo caso è accessibile a più processi.
- Uno strumento di sincronizzazione in questo modello è il **semaforo** e le chiamate di sistema di sincronizzazione **wait** e **signal**.

Esempio di descrittore di dispositivo

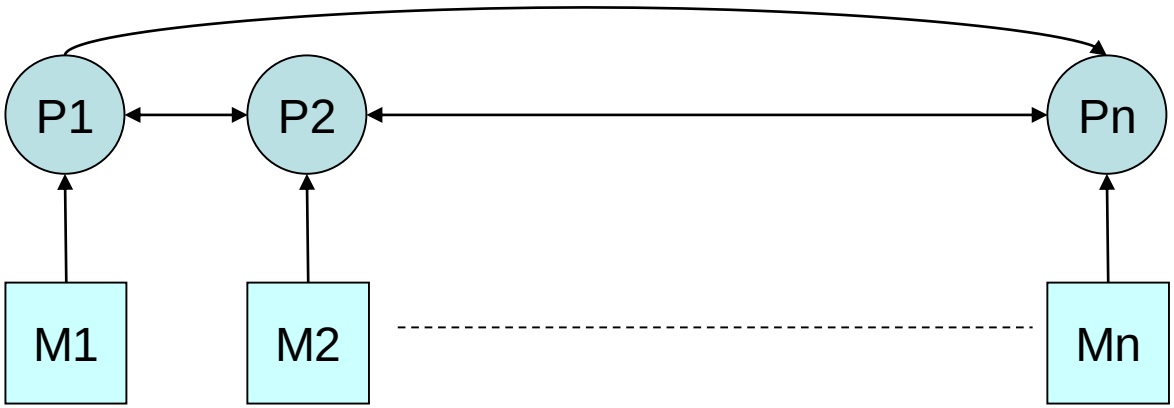


Interazione tra processi in sistemi a memoria comune

Modello ad ambiente locale

- Ogni processo può vedere ed accedere solo alle proprie risorse locali che non sono accessibili agli altri processi. I processi interagiscono esclusivamente tramite *scambio di messaggi*.
- Questo modello è generalmente usato nelle reti di calcolatori, ma può essere anche usato in sistemi mono o multi processore a memoria condivisa. Lo scambio di messaggi avviene sia tramite rete di comunicazione che tramite segmenti di memoria comune.
- Un esempio di strumento di sincronizzazione è dato dalle chiamate di sistema *send()* e *receive()*.
- Le chiamate send e receive, così come il semaforo con le chiamate wait e signal sono strumenti a basso livello, forniti dal kernel.

Scambio di messaggi



Interazione tra processi in sistemi a memoria locale

Problema della mutua esclusione

- Sia per la cooperazione che per la competizione è necessario che i processi eseguano le operazioni che riguardano le risorse comuni in ***mutua esclusione***.
- Con mutua esclusione si intende che le operazioni con le quali i processi accedono alle risorse comuni non si sovrappongano nel tempo.
- Queste operazioni prendono il nome di ***sezioni critiche***.

Soluzioni al problema della mutua esclusione

- La soluzione del problema della mutua esclusione consiste nel realizzare un protocollo che i processi devono seguire per interagire correttamente con la risorsa condivisa.
- Un processo, prima di entrare nella sezione critica, dovrà eseguire una serie di operazioni, detta ***sezione d'ingresso (prologo)***, per assicurarsi l'uso esclusivo della risorsa, nel caso sia libera, oppure ne impediscano l'accesso nel caso sia occupata.
- Al termine dell'esecuzione della sezione critica il processo deve rilasciare la risorsa per consentirne l'allocazione ad altri processi che la richiedono. Per questo dovrà eseguire un'altra serie di operazioni detta ***sezione di uscita (epilogo)***.

- Un semplice esempio delle operazioni di prologo e di epilogo si ottiene utilizzando una variabile intera condivisa ***occupato*** il cui valore è **1** se la risorsa è occupata o **0** se essa è libera.

Prologo:

```
while (occupato == 1);
occupato = 1;
<sezione critica>
```

Epilogo:

```
occupato = 0;
```

- Affinché la soluzione sia valida è necessario che il SO permetta ai processi di eseguire le istruzioni di **lettura e scrittura** della variabile di controllo condivisa (nell'esempio *occupato*) in **modo atomico**.

- Molti processori possiedono istruzioni che consentono di leggere e modificare il contenuto di una locazione in un unico ciclo di memoria. Un esempio è dato dall'istruzione **TSL** (**Test and Set Lock**).
- L'istruzione **TSL R, X** copia il contenuto della locazione di memoria **X** nel registro **R** del processore e viene scritto in X un **valore diverso da 0**.
- Nel caso di sistemi multiprocessore, il processore che esegue la TSL **blocca il bus di memoria** per impedire che altri processori accedano alla memoria fino a quando non ha completato l'operazione di TSL.
- La mutua esclusione si ottiene realizzando due funzioni **lock(x)** e **unlock(x):**

- **Lock(x):**

```
LOCK:
   TSL R, X    copia il valore di X in R e pone X=1
               (R=X;X=1)
   CMP R,0     verifica se R==0
   JNE LOCK    se R!=0 riesegue il ciclo
   RET         ritorno
```

- **Unlock(x)**

```
   mov x,0      scrive in X il valore 0
   RET          ritorna al chiamante
```

Esempio di due processi

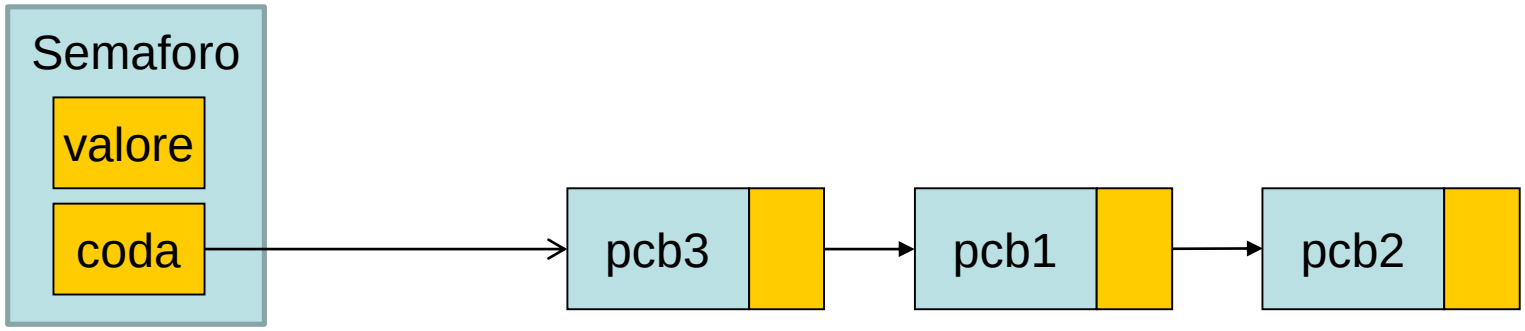
```
P1
Prologo:   lock(x)
           <sezione critica P1>
Epilogo:   unlock(x)

P2
Prologo:   lock(x)
           <sezione critica P2>
Epilogo:   unlock(x)
```

Questa soluzione è caratterizzata da condizioni di **attesa attiva** dei processi. La soluzione è valida quindi per sistemi multiprocessore ed è limitata al caso di sezioni critiche **brevi**.

Semafori

- un semaforo **s** è una struttura dati gestita principalmente dalle funzioni **wait(s)** e **signal(s) e** dalla funzione di inizializzazione **init(s, valore).**
- La struttura dati è costituita da una variabile intera non negativa ***valore*** e da una coda di descrittori di processi sospesi, ***coda.***

```
typedef struct {
    int valore;
    struct processo *coda;
} semaforo;
```



- *La **wait()*** è chiamata da un processo per verificare lo stato di un semaforo secondo il seguente pseudocodice:

```
void wait(semaforo s){
  if (s.valore==0) {
    <il processo viene sospeso>
    <il descrittore del processo viene inserito in
     s.coda>
   }
   else
     s.valore=s.valore-1;
}
```

- Se **s.valore** = 0, la *wait()* porta il processo nello **stato di bloccato** e inserisce il suo descrittore nella coda **s.coda** associata al semaforo.
- Se **s.valore** è > 0, esso viene decrementato di 1 e il processo continua la sua esecuzione;
- La primitiva ***signal()*** risveglia eventuali processi che si trovano sospesi sul semaforo. Se la coda è vuota la signal incrementa di 1 il valore del semaforo:

```
void signal(semaforo s) {
  if ( <se la coda s.coda non è vuota>)
     <estrai dalla prima posizione di s.coda il
      descrittore del processo portandolo nello
      stato di pronto>
   }
    else
      s.valore=s.valore+1;
}
```

- La ***signal()*** **non è bloccante** per il processo che la chiama, mentre la ***wait()*** è **bloccante** se **s.valore=0**.
- Le chiamate *wait()* e *signal()* devono essere realizzate in modo che siano eseguite in modo indivisibile. L'atomicità delle funzioni *wait()* e *signal()* si realizza a livello di kernel **disabilitando le interruzioni** del processore durante la loro esecuzione.
- Il semaforo è stato ideato da Dijkstra, e usato per la prima volta nel sistema operativo Theos.
- Il nome originale della *wait()* era ***P*** e quello della *signal()* era ***V***. Tali nomi erano stati attribuiti dallo stesso Dijkstra, e corrispondono alle iniziali delle parole olandesi *proberen* (verificare) e *verhogen* (incrementare).

Soluzione al problema della mutua esclusione con semafori.

- Si associa alla risorsa condivisa un semaforo, **inizializzandolo al valore 1** (**libero**) e usando per ogni processo che richiede la risorsa il seguente protocollo:

```
Init(mutex,1);
Prologo:  wait(mutex);
            <sezione critica>;
Epilogo:  signal(mutex);
```

- Esempio di due processi P1, P2 che accedono alla stessa risorsa comune R. Tale schema è valido per qualsiasi numero di processi.

```
P1
Init(mutex,1);
Prologo: wait(mutex);
         <sezione critica P1>;
Epilogo: signal(mutex);

P2
Prologo: wait(mutex);
         <sezione critica P2>;
Epilogo: signal(mutex);
```

- La soluzione mostrata evita condizioni di attesa attiva in quanto un **processo viene sospeso** se trova il semaforo occupato.
- Generalmente, la coda associata al semaforo è gestita con politica FCFS per evitare che qualche processo che si trovi sospeso possa entrare in una situazione di attesa indefinita (starvation).
- Il semaforo che può assumere solo i due valori 0 e 1 prende il nome di **semaforo binario**, e spesso viene chiamato **mutex** (mutua esclusione).
- La **correttezza** della soluzione dipende dal **valore iniziale del semaforo** che deve essere posto a **1** e al corretto posizionamento delle funzioni di sistema *wait()* e *signal()* prima e dopo la sezione critica.

- Nei **sistemi multiprocessore**, per garantire che *wait()* e *signal()* siano eseguite in mutua esclusione sul semaforo, è necessario che i processi utilizzino le funzioni *lock()* e *unlock()*, secondo il protocollo seguente:

```
lock(x);
    wait(mutex);
unlock(x);
    <sezione critica>;
lock(x);
    signal(mutex);
unlock(x);
```

- La *lock()* garantisce che le chiamate *wait()* e *signal()* siano eseguite da un processo alla volta.

- La *wait()* e la *signal()***,** relative al semaforo ***mutex*****,** assicurano la mutua esclusione delle sezioni critiche su una **risorsa *R***, mentre la variabile *x*, con le *lock(x)* e *unlock(x)* assicura la mutua esclusione delle primitive *wait()* e *signal()* sul semaforo *mutex*.