SAPIENZA
UNIVERSITÀ DI ROMA

Programmazione Sistemi Multicore

Ripasso del linguaggio di programmazione C

Christian Cardia & Gabriele Saturni

cardia@di.uniroma1.it – saturni@di.uniroma1.it
Dip. Informatica, st. 317 – via Salaria 113, Roma

POSIX PTHREAD

- Prima di iniziare, è necessario eseguire alcuni passaggi

1. Aggiungere la libreria pthrread → #include <pthread.h> al codice sorgente.

2. Se usate gcc, potete semplicemente specificare -pthread che imposterà tutte le definizioni e le librerie time-link appropriate. Su altri compilatori, potrebbe essere necessario definire _REENTRANT e collegarsi a -lpthread.

- Opzionale: alcuni compilatori potrebbero richiedere la definizione di _POSIX_PTHREAD_SEMANTICS per alcune chiamate di funzione come sigwait ().

POSIX PTHREAD - creazione

- Un thread è rappresentato dal tipo **pthread_t**. Per creare un thread, è disponibile la seguente funzione:

```
1  int pthread_create(pthread_t *thread, pthread_attr_t *attr,
2                     void *(*start_routine)(void *), void *arg);
```

- **pthread_t *thread:** è un puntatore ad un identificatore di thread in cui verrà scritto l'identificatore del thread creato.
- **pthread_attr_t *attr:** attributi da applicare al thread
- **void *(*start_routine)(void *):** è il nome (indirizzo) della procedura da fare eseguire dal thread. Deve avere come unico argomento un puntatore.
- **void *arg:** argomenti da passare alla funzione start_routine

POSIX PTHREAD

- Vediamo altre funzioni importanti:

```
1  void pthread_exit(void *value_ptr);
2  int pthread_join(pthread_t thread, void **value_ptr);
3
```

- **pthread_exit(void *value_ptr):** termina l'esecuzione del thread da cui viene chiamata, immagazzina il valore puntato da value_ptr, restituendolo ad un altro thread che attende la sua fine. Il sistema libera le risorse allocate al thread.

- **pthread_join(pthread_t thread, void **value_ptr):** sospende il thread chiamante in attesa di una terminazione corretta del thread specificato come primo argomento (pthread_t thread) con un dato facoltativo * value_ptr passato dalla chiamata del thread di terminazione a pthread_exit ()

POSIX PTHREAD

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 10

/* create thread argument struct for thr_func() */
typedef struct _thread_data_t {
  int tid;
  double stuff;
} thread_data_t;

/* thread function */
void *thr_func(void *arg) {
  thread_data_t *data = (thread_data_t *)arg;

  printf("hello from thr_func, thread id: %d\n", data->tid);

  pthread_exit(NULL);
}
```

POSIX PTHREAD

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 10

/* create thread argument struct for thr_func() */
typedef struct _thread_data_t {
  int tid;
  double stuff;
} thread_data_t;

/* thread function */
void *thr_func(void *arg) {
  thread_data_t *data = (thread_data_t *)arg;

  printf("hello from thr_func, thread id: %d\n", data->tid);

  pthread_exit(NULL);
}
```

Caricamento delle librerie

POSIX PTHREAD

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 10

/* create thread argument struct for thr_func() */
typedef struct _thread_data_t {
  int tid;
  double stuff;
} thread_data_t;

/* thread function */
void *thr_func(void *arg) {
  thread_data_t *data = (thread_data_t *)arg;

  printf("hello from thr_func, thread id: %d\n", data->tid);

  pthread_exit(NULL);
}
```

Definiamo un numero di thread che verranno creati. Costante con il costrutto #define

POSIX PTHREAD

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 10

/* create thread argument struct for thr_func() */
typedef struct _thread_data_t {
  int tid;
  double stuff;
} thread_data_t;

/* thread function */
void *thr_func(void *arg) {
  thread_data_t *data = (thread_data_t *)arg;

  printf("hello from thr_func, thread id: %d\n", data->tid);

  pthread_exit(NULL);
}
```

Definiamo la struct contenente i dati del thread

POSIX PTHREAD

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 10

/* create thread argument struct for thr_func() */
typedef struct _thread_data_t {
  int tid;
  double stuff;
} thread_data_t;
```

Funzione che viene eseguita dal thread

```c
/* thread function */
void *thr_func(void *arg) {
  thread_data_t *data = (thread_data_t *)arg;

  printf("hello from thr_func, thread id: %d\n", data->tid);

  pthread_exit(NULL);
}
```

POSIX PTHREAD

```
int main(int argc, char **argv) {
  pthread_t thr[NUM_THREADS];
  int i, rc;
  /* create a thread_data_t argument array */
  thread_data_t thr_data[NUM_THREADS];

  /* create threads */
  for (i = 0; i < NUM_THREADS; ++i) {
    thr_data[i].tid = i;
    if ((rc = pthread_create(&thr[i], NULL, thr_func, &thr_data[i]))) {
      fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
      return EXIT_FAILURE;
    }
  }
  /* block until all threads complete */
  for (i = 0; i < NUM_THREADS; ++i) {
    pthread_join(thr[i], NULL);
  }

  return EXIT_SUCCESS;
}
```

POSIX PTHREAD

```
int main(int argc, char **argv) {
  pthread_t thr[NUM_THREADS];
  int i, rc;
  /* create a thread_data_t argument array */
  thread_data_t thr_data[NUM_THREADS];

  /* create threads */
  for (i = 0; i < NUM_THREADS; ++i) {
    thr_data[i].tid = i;
    if ((rc = pthread_create(&thr[i], NULL, thr_func, &thr_data[i]))) {
      fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
      return EXIT_FAILURE;
    }
  }
  /* block until all threads complete */
  for (i = 0; i < NUM_THREADS; ++i) {
    pthread_join(thr[i], NULL);
  }

  return EXIT_SUCCESS;
}
```

Nome del thread

POSIX PTHREAD

```
int main(int argc, char **argv) {
  pthread_t thr[NUM_THREADS];
  int i, rc;
  /* create a thread_data_t argument array */
  thread_data_t thr_data[NUM_THREADS];

  /* create threads */
  for (i = 0; i < NUM_THREADS; ++i) {
    thr_data[i].tid = i;
    if ((rc = pthread_create(&thr[i], NULL, thr_func, &thr_data[i]))) {
      fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
      return EXIT_FAILURE;
    }
  }
  /* block until all threads complete */
  for (i = 0; i < NUM_THREADS; ++i) {
    pthread_join(thr[i], NULL);
  }

  return EXIT_SUCCESS;
}
```

Puntatore alla funzione che il thread deve eseguire

Valore attributo posto a null per ottenere comportamento di default

POSIX PTHREAD

```
int main(int argc, char **argv) {
  pthread_t thr[NUM_THREADS];
  int i, rc;
  /* create a thread_data_t argument array */
  thread_data_t thr_data[NUM_THREADS];

  /* create threads */
  for (i = 0; i < NUM_THREADS; ++i) {
    thr_data[i].tid = i;
    if ((rc = pthread_create(&thr[i], NULL, thr_func, &thr_data[i]))) {
      fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
      return EXIT_FAILURE;
    }
  }
  /* block until all threads complete */
  for (i = 0; i < NUM_THREADS; ++i) {
    pthread_join(thr[i], NULL);
  }

  return EXIT_SUCCESS;
}
```

Argomenti passati alla funzione

POSIX PTHREAD - MUTEX

- MUTEX → abbreviazione per «mutual exclusion»
- Le variabili mutex sono uno dei principali mezzi per implementare la sincronizzazione tra i thread
- Agiscono come dei lock per proteggere l'accesso alle risorse condivise
- Il concetto basilare del mutex adottato in pthread è che solo il thread che possiede il lock su una variabile può usarla in quel momento

POSIX PTHREAD - MUTEX

- Le variabili mutex devono essere dichiarate con il tipo **pthread_mutex_t** e devono essere inizializzate prima di essere usate.

pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;

- Tale istruzione inizializza la variabile staticamente.Se invece si preferisce farlo dinamicamente si può usare:

pthread_mutex_init()

- Questo metodo permette di usare il mutex settandolo con un particolare attributo **attr**
- Il mutex è inizialmente unlocked

POSIX PTHREAD - MUTEX

- L'attributo **attr** è usato per settare le proprietà del mutex.
- Il tipo dell'attributo deve essere del tipo pthread_mutexattr_t altrimenti, per un comportamento di default si può usare NULL.
- La pthread definisce tre tipologie di attributi:
  - **Protocol** specifica il protocollo usato per evitare l'inversione di priorità (problema che si può verificare durante l'accesso alla sezione critica)
  - **Priocelling** specifca il valore di priorità del mutex
  - **Process-shared** specifica il processo condiviso del mutex

POSIX PTHREAD - MUTEX

- I metodi **pthread_mutexattr_init()** e **pthread_mutexattr_destroy()** sono usate per creare e distruggere un mutex.
- In particolare il destroy deve essere usato per liberare la memoria se un mutex non è più necessario

POSIX PTHREAD - MUTEX

- I seguenti metodi servono per bloccare e sbloccare un mutex:
- pthread_mutex_lock(mutex) → usata da un thread per acquisire un lock su una specifica variabile mutex. Se un altro thread ha già il lock questa primitiva bloccal il thread fino a quando il mutex non viene sbloccato
- pthread_mutex_trylock(mutex) → tenta di acquisire un lock. Tuttavia, se il mutex è già bloccato, la funzione restitusce immediatamente un codice di errore «busy» . Questa routine è utile per prevenire il deadlock
- pthread_mutex_unlock(mutex) → rilascia il lock. Restituisce un errore se:
  1. il mutex è gia sbloccato
  2. Il mutex è bloccato da un altro thread

POSIX PTHREAD - MUTEX

- Non c'è nulla di magico nei mutex. Agiscono come dei signori ben educati.
- Spetta al programmatore assicurarsi che tutti i thread necessari eseguano correttamente il blocco mutex e sblocchino le chiamate. Il seguente scenario mostra un errore logico:

Thread 1	Thread 2	Thread 3
Lock	Lock	
A = 2	A = A+1	A = A*B
Unlock	Unlock	

- **Domanda:** Quando più di un thread è in attesa di un mutex bloccato, a quale thread verrà concesso il blocco prima che venga rilasciato?

POSIX PTHREAD - MUTEX

Thread 1	Thread 2	Thread 3
Lock	Lock	
A = 2	A = A+1	A = A*B
Unlock	Unlock	

- **Domanda:** Quando più di un thread è in attesa di un mutex bloccato, a quale thread verrà concesso il blocco prima che venga rilasciato?
- **Risposta:** A meno che non venga utilizzata la pianificazione prioritaria dei thread (non coperta), l'assegnazione verrà lasciata allo scheduler del sistema nativo e potrebbe sembrare più o meno casuale.

POSIX PTHREAD -MUTEX

- Nell'esempio che andremo a mostrare illustreremo un programma, basato su thread e mutex, per implementare il prodotto scalare.
- I dati principali sono resi disponibili a tutti i thread attraverso una struttura accessibile a livello globale.
- Ogni thread funziona su una parte diversa dei dati.
- Il thread principale attende che tutti i thread completino i loro calcoli, quindi stampa la somma risultante.

PTHREAD MUTEX - esempio


```
/*
La seguente struttura contiene le informazioni necessarie
per consentire alla funzione "dotprod" di accedere ai suoi dati di input e
posizionare il suo output nella struttura.
*/

typedef struct
 {
   double      *a;
   double      *b;
   double      sum;
   int     veclen;
 } DOTDATA;

/* Definire variabili accessibili a livello globale e un mutex */

#define NUMTHRDS 4
#define VECLEN 100
   DOTDATA dotstr;
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;
```

```
/*
La funzione dotprod viene attivata quando viene creato il thread.
Tutti gli input per questa routine sono ottenuti da una struttura
 di tipo DOTDATA e tutti gli output di questa funzione sono scritti in
 questa struttura. Il vantaggio di questo approccio è evidente per il
 programma multi-thread: quando viene creato un thread ne passiamo uno singolo
 argomento per la funzione attivata - in genere questo argomento
 è un numero di thread. Tutte le altre informazioni richieste dal
 la funzione è accessibile dalla struttura accessibile a livello globale.
 */
  void *dotprod(void *arg)
  {
     /* Definizione variabili locali */
     int i, start, end, len ;
     long offset;
     double mysum, *x, *y;
     offset = (long)arg;
     len = dotstr.veclen;
     start = offset*len;
     end   = start + len;
     x = dotstr.a;
     y = dotstr.b;
     /*
     Eseguire il dot prod e assegnare il risultato
     alla variabile appropriata nella struttura.
     */
     mysum = 0;
     for (i=start; i<end ; i++)
      {
        mysum += (x[i] * y[i]);
      }
     /*
    Blocca un mutex prima di aggiornare il valore nella condivisione
     struttura e sbloccalo dopo l'aggiornamento.
     */
     pthread_mutex_lock (&mutexsum);
     dotstr.sum += mysum;
     pthread_mutex_unlock (&mutexsum);
     pthread_exit((void*) 0);
  }
```

```
/*
Il programma principale crea thread che fanno tutto il lavoro e quindi
stampare il risultato al completamento. Prima di creare i thread,
i dati di input vengono creati. Poiché tutti i thread aggiornano una struttura condivisa,
abbiamo bisogno di un mutex per l'esclusione reciproca. Il thread principale deve attendere
per completare tutti i thread, attende ognuno dei thread. Specifichiamo
un valore dell'attributo thread che consente al thread principale di unirsi a
discussioni che crea. Nota anche che liberiamo le maniglie quando lo sono
non è più necessario.
*/
        int main (int argc, char *argv[])
        {
           long i;
           double *a, *b;
           void *status;
           pthread_attr_t attr;

           /* alloca le variabili e le inizializza */
           a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
           b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));

           for (i=0; i<VECLEN*NUMTHRDS; i++)
             {
             a[i]=1.0;
             b[i]=a[i];
             }

           dotstr.veclen = VECLEN;
           dotstr.a = a;
           dotstr.b = b;
           dotstr.sum=0;

           pthread_mutex_init(&mutexsum, NULL);
```

```
    /* crea i thread per fare il dotproduct  */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for(i=0; i<NUMTHRDS; i++)
    {
    /*
   Ogni thread lavora su una porzione distinta di datia. l? offsetè specificato da
    'i'. La dimensione dei dati per ogni thread è indicata da VECLEN.
    */
    pthread_create(&callThd[i], &attr, dotprod, (void *)i);
    }

    pthread_attr_destroy(&attr);

    /* Aspetta gli altri thread */
    for(i=0; i<NUMTHRDS; i++)
        {
        pthread_join(callThd[i], &status);
        }

    /* Dopo il join, stampa i risultati e libera la memoria */
    printf ("Sum =  %f \n", dotstr.sum);
    free (a);
    free (b);
    pthread_mutex_destroy(&mutexsum);
    pthread_exit(NULL);
}
```

Esercizio 1

- Problema del lettore-scrittore: un thread si occupa di scrivere la stringa ciaociao, mentre altri tre si occupano della sua lettura. Non è possible avere letture e scritture contemporaneamente. Risolvere il problema usando i mutex.

Soluzione Esercizio 1

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER ;
char buffer [10];
int pronto = 0;

void * reader(void *arg){
    while( pronto == 0 ){ /*non fa nulla finché la variabile pronto non è diversa da 0*/
    }
    pthread_mutex_lock (& mutex);
    printf("Ho letto %s \n", buffer);
    pthread_mutex_unlock (& mutex);
    return NULL;
}

void * writer(void *arg){
    pthread_mutex_lock (& mutex);
    sprintf(buffer, "ciaociao\n");/*come printf ma scrive non su video ma su una stringa*/
    pronto = 1;
    pthread_mutex_unlock (& mutex);
    return NULL;
}
```

Soluzione Esercizio 1

```
int main(){
    pthread_t th_r1 , th_r2 , th_r3;
    pthread_t th_w1;
    pthread_create (&th_r1 , NULL , reader , NULL);
    pthread_create (&th_r2 , NULL , reader , NULL);
    pthread_create (&th_r3 , NULL , reader , NULL);
    pthread_create (&th_w1 , NULL , writer , NULL);
    pthread_join(th_r1 , NULL);
    pthread_join(th_r2 , NULL);
    pthread_join(th_r3 , NULL);
    pthread_join(th_w1 , NULL);
    return 0;
}
```

Esercizio 2

- Problema del lettore-scrittore: un thread si occupa di riempire un array di 20 interi, con dei valori a piacere, mentre altri tre si occupano della sua lettura. Non è possible avere letture e scritture contemporaneamente. Risolvere il problema usando i mutex.

- Nota: la stampa degli elementi dell'array deve seguire il formato:[elem1, elem2, elem3, elemn]

Soluzione – Esercizio 2

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER ;
int array_interi [20];
int pronto = 0;
void * reader(void *arg){
    while( pronto == 0 ){ /*non fa nulla finché la variabile pronto non è diversa da 0*/
    }
    pthread_mutex_lock (& mutex);
    printf("Ho letto: [");
    for(int i=0; i<20; i++){
        if(i < 19) {
            printf("%d,", array_interi[i]);
        } else {
            printf("%d", array_interi[i]);
        }
    }
    printf("]\n");
    pthread_mutex_unlock (& mutex);
    return NULL;
}
void * writer(void *arg){
    pthread_mutex_lock (& mutex);
    for(int i=0; i<20; i++) {
        array_interi[i]+=i;
    }
    pronto = 1;
    pthread_mutex_unlock (& mutex);
    return NULL;
}
```

Esercizio 3

- Si scriva un programma che legge da input una stringa e la memorizza dinamicamente.
- Un thread si occupa di modificare la stringa sostituendo tutti i caratteri ‘a’ con il carattere ‘b’.
- Altri tre thread si occupano della sua lettura.
- Non è possibile avere letture e scritture contemporaneamente. Risolvere il problema usando i mutex.

Esercizio 3- Soluzione

SAPIENZA UNIVERSITÀ DI ROMA

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <string.h>

//prototipo funzione
char *stringaInput();

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER ;

int pronto = 0;

void * reader(void *arg){
    char *stringa = (char *) arg;

    while( pronto == 0 ){
    /*non fa nulla finché la variabile pronto
     * non è diversa da 0
     */
    }
    pthread_mutex_lock (& mutex);
    printf("\nSono il lettore. Stringa: %s",stringa);
    pthread_mutex_unlock (& mutex);
    return NULL;
}
void * writer(void *arg){
    char *stringa = (char *) arg;
    pthread_mutex_lock (& mutex);

    for(int i=0; i<strlen(stringa); i++) {
        if((stringa[i]=='a')){
            stringa[i] = 'b';
        }
    }
    pronto = 1;
    pthread_mutex_unlock (& mutex);
    return NULL;
}
```

Esercizio 3- Soluzione

SAPIENZA UNIVERSITÀ DI ROMA

```
40 int main(){
41
42     char *c;
43     c = NULL;
44     c = stringaInput();
45
46     if (c==NULL){
47         printf("Memoria non allocata! \n");
48         exit (1);
49     }
50
51     printf("Stringa inserita (Thread non creati ancora): %s",c);
52
53     pthread_t th_r1 , th_r2 , th_r3;
54     pthread_t th_w1;
55     pthread_create (&th_r1 , NULL , reader , c);
56     pthread_create (&th_r2 , NULL , reader , c);
57     pthread_create (&th_r3 , NULL , reader , c);
58     pthread_create (&th_w1 , NULL , writer , c);
59     pthread_join(th_r1 , NULL);
60     pthread_join(th_r2 , NULL);
61     pthread_join(th_r3 , NULL);
62     pthread_join(th_w1 , NULL);
63
64     printf("\n");
65
66     return 0;
67 }
```

Esercizio 3- Soluzione

SAPIENZA UNIVERSITÀ DI ROMA

```
70  char *stringaInput(){
71
72      char *stringa = NULL;
73      char c;
74      int numChar = 0;
75
76      stringa = malloc(sizeof(char)*(numChar+1));
77      if (stringa == NULL){
78          printf("Non è possibile allocare spazio!\n");
79          exit(1);
80      }
81      stringa[numChar] = '\0';
82      printf("Inserire la stringa: \n");
83      while( (c=getchar()) != '\n' ){
84          numChar++;
85          stringa = realloc(stringa,(numChar+1)*(sizeof(char)));
86          if (stringa == NULL){
87              printf("Non è possibile allocare spazio!\n");
88              exit(1);
89          }
90          stringa[numChar-1] = c;
91          stringa[numChar] = '\0';
92      }
93
94      printf("\n");
95
96      return stringa;
97
98  }
```

Esercizio 4

- Si scriva un programma che legge da input dei numeri interi (fino al numero -1) e li salva dinamicamente.
- L'array di interi costruito dinamicamente avrà quindi come ultimo intero il numero -1.
- Un thread si occupa di modificare ogni singolo numero intero semplicemente moltiplicandolo per due.
- Altri tre thread si occupano della sua lettura.
- Non è possibile avere letture e scritture contemporaneamente. Risolvere il problema usando i mutex.

Esercizio 4 - Soluzione


```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <string.h>

//prototipo funzione
int *inserisciArray();

pthread mutex t mutex = PTHREAD MUTEX INITIALIZER ;

int pronto = 0;

void * reader(void *arg){
    int *array = (int *) arg;

    while( pronto == 0 ){
    /*non fa nulla finché la variabile pronto
     * non è diversa da 0
     */
    }
    pthread_mutex_lock (& mutex);
    printf("\nSono il lettore. Interi-->\n");

    int i=0;
    for(i=0;array[i]!=-1;i++){
        printf("%d - ",array[i]);
    }
    printf("\n");

    pthread_mutex_unlock (& mutex);
    return NULL;
}
```

Esercizio 4 - Soluzione

SAPIENZA UNIVERSITÀ DI ROMA

```
33  void * writer(void *arg){
34      int *array = (int *) arg;
35      pthread_mutex_lock (& mutex);
36
37      printf("\nSono lo scrittore. Interi inseriti-->");
38      int i=0;
39      for(i=0;array[i]!=-1;i++){
40          printf("%d - ",array[i]);
41      }
42      printf("\n");
43
44      for(i=0;array[i]!=-1;i++){
45          array[i] = array[i] * 2;
46      }
47
48      pronto = 1;
49      pthread_mutex_unlock (& mutex);
50      return NULL;
51  }
```

Esercizio 4 - Soluzione


```
53 int main(){
54
55     int *array;
56     array = NULL;
57     array = inserisciArray();
58
59     if (array==NULL){
60         printf("Memoria non allocata! \n");
61         exit (1);
62     }
63
64     /*int i=0;
65     for(i=0;array[i]!=-1;i++){
66         printf(" %d - ",array[i]);
67     }*/
68
69     pthread_t th_r1 , th_r2 , th_r3;
70     pthread_t th_w1;
71     pthread_create (&th_r1 , NULL , reader , array);
72     pthread_create (&th_r2 , NULL , reader , array);
73     pthread_create (&th_r3 , NULL , reader , array);
74     pthread_create (&th_w1 , NULL , writer , array);
75     pthread_join(th_r1 , NULL);
76     pthread_join(th_r2 , NULL);
77     pthread_join(th_r3 , NULL);
78     pthread_join(th_w1 , NULL);
79
80     printf("\n");
81
82     return 0;
83 }
```

Esercizio 4 - Soluzione

SAPIENZA UNIVERSITÀ DI ROMA

```
86  int *inserisciArray(){
87
88      int *array = NULL;
89      int numElementi=0;
90      while(1){
91          printf("Inserire un intero (-1 per continuare): \n");
92          int a=0;
93          scanf("%d",&a);
94          array = realloc(array, (numElementi+1)*(sizeof(int)) );
95          if(array==NULL){
96              printf("Non è possibile allocare spazio!\n");
97              exit(1);
98          }
99          numElementi++;
100         array[numElementi-1] = a;
101         if(a==-1){
102             break;
103         }
104
105     }
106
107     return array;
108
109 }
```

Esercizio 5

- Si scriva un programma che definisce due funzioni:
- *char * leggiStringa()* –legge una stringa in input memorizzandola dinamicamente e ritorna un puntatore a questa stringa
- *void sostituisciNumeri(char *p)* – prende come parametro un puntatore ad una stringa e sostituisce tutti i numeri con il carattere * (asterisco)
- Infine si dichiari la funzione *main* che per prima cosa chiama la funzione *leggiStringa()*, stampa la stringa inserita in input, chiama la funzione *sostituisciNumeri* ed infine stampa la stringa modificata

Per fare l'esercizio si utilizzi l'allocazione dinamica.

N.B. ricordarsi che le stringhe devono terminare sempre con il carattere terminatore stringa, ovvero '\0'

Esercizio 5 – Soluzione (1/2)

```c
#include <stdio.h>
#include <stdlib.h>

char *leggiStringa();
void sostituisciNumeri(char *p);

int main() {

  char *stringa = leggiStringa();

  if(stringa == NULL){
      printf("Impossibile allocare la memoria! \n");
      return 1;
  }

  printf("Stringa input: %s \n",stringa);

  sostituisciNumeri(stringa);

  printf("Stringa senza numeri: %s \n",stringa);

  return 0;
}
```

Esercizio 5 – Soluzione (2/2)


```c
char *leggiStringa(){

  char c,*stringa;
  stringa = malloc(sizeof(char));

  if (stringa == NULL){
      return stringa;
  }

  scanf("%c",&c);
  int i;
  for(i=0;c!='\n';i++) {
        stringa[i]=c;
        if(!(stringa=realloc(stringa,(i+2)*sizeof(char)))) {
            printf("Errore nell\'allocazione della memoria");
            return stringa;
        }
        scanf("%c",&c);
  }

  stringa[i]='\0';

  return stringa;

}

void sostituisciNumeri(char *p){
    int i = 0;
    while(p[i] != '\0'){
        if( (p[i]<='9') && (p[i]>='0') ){
            p[i] = '*';
        }
        i++;
    }

    return;
}
```

Esercizio 6

- Si scriva un programma che definisce due funzioni:
- *char * leggiStringa()* –legge una stringa in input memorizzandola dinamicamente e ritorna un puntatore a questa stringa
- *void concatena(char *p1, char *p2)* – prende in input due parametri che sono due puntatori a due rispettive stringhe. La funzione deve concatenare in *p1*, le due stringhe puntate rispettivamente da *p1* e *p2*
- Infine si dichiari la funzione *main* che per prima cosa permette di inserire due stringhe all'utente (richiamando la funzione *leggiStringa*), le concatena (richiamando la funzione *concatena*) ed infine stampa la stringa concatenata.

Esercizio 6 – Soluzione (1/3)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *stringaInput();
void concatena(char *s1, char *s2);

int main(void)
{

    char *p1 = stringaInput();
    char *p2 = stringaInput();

    printf("Stringa1 inserita: %s \n",p1);
    printf("Stringa2 inserita: %s \n",p2);

    concatena(p1,p2);

    printf("Concatenzaione: %s \n",p1);
    return 0;

}
```

Esercizio 6 – Soluzione (2/3)

```
25  char *stringaInput(){
26
27      char *stringa = NULL;
28      char c;
29      int numChar = 0;
30
31      stringa = malloc(sizeof(char)*(numChar+1));
32      if (stringa == NULL){
33          printf("Non è possibile allocare spazio!\n");
34          exit(1);
35      }
36      stringa[numChar] = '\0';
37      printf("Inserire la stringa: \n");
38      while( (c=getchar()) != '\n' ){
39          numChar++;
40          stringa = realloc(stringa,(numChar+1)*(sizeof(char)));
41          if (stringa == NULL){
42              printf("Non è possibile allocare spazio!\n");
43              exit(1);
44          }
45          stringa[numChar-1] = c;
46          stringa[numChar] = '\0';
47      }
48
49      printf("\n");
50
51      return stringa;
52
53  }
```

Esercizio 6 – Soluzione (3/3)

```
56
57  void concatena(char *s1, char *s2){
58      int dim1 = strlen(s1);
59      int i = 0;
60
61      while(s2[i] != '\0'){
62          s1 = realloc(s1,(dim1+2)*sizeof(char));
63          if (s1 == NULL){
64              printf("Non è possibile allocare ulteriore spazio!\n");
65              exit(1);
66          }
67          s1[dim1] = s2[i];
68          s1[dim1+1] = '\0';
69          dim1++;
70          i++;
71      }
72
73
74  }
```