Dato il seguente frammento di codice:

```
...
int main(int argc, int argv[])
{
 char line[128];
 int seconds;
 char message[64];
 while(1){
    printf("Allarme >");
    if(fgets(line, sizeof(line), stdin)==NULL) exit(0);
    if(strlen(line)<=1)
    continue;
    if(sscanf(line, "%d %64[^\n]", &seconds, message)<2) {
    ...
    }
  }
}
```

completatelo per realizzare il generatore di allarmi tramite thread. Specificate le dichiarazioni delle
variabili e delle funzioni aggiuntive al posto dei . . . , eventualmente la vostra soluzione può definire
diversamente le variabili seconds e message già presenti. Ponete attenzione affinchè la vostra
soluzione gestica correttamente anche molteplici allarmi definiti simultaneamente.
SOL:

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

typedef struct alarm_tag{
 int seconds;
 char message[64];
} alarm_t;
void *alarm_thread(void *arg){

 alarm_t *alarm= (alarm_t *) arg;

 sleep(alarm->seconds);
 printf("(%d) %s\n", alarm->seconds, alarm->message);
 free(alarm);
 return(NULL);
}

int main(int argc, char *argv[]){
 int status;
 char line[128];
 alarm_t *alarm;
```

```
 pthread_t thread;
 while(1){
    printf("Alarm >");
    if(fgets(line, sizeof(line), stdin)==NULL) exit(0);
    if(strlen(line)<=1) continue;

    alarm = (alarm_t *) malloc(sizeof(alarm_t));
    if(alarm==NULL){
        perror("Allocate alarm");
        exit(-1);
    }

    if(sscanf(line, "%d %64[^\n]", &alarm->seconds, alarm->message)<2)
    {
        fprintf(stderr, "Bad command\n");
    }
    else
    {
        status= pthread_create(&thread, NULL, alarm_thread, alarm);
        if(status !=0)
        {
            perror("Create alarm thread");
            exit(-1);
        }
    }
 }
}
```

Questa è una possibile soluzione, non l'unica. La malloc della struttura serve per assicurarsi che le
informazioni sugli allarmi non vengano sovrascritte ad ogni iterazione in quanto alla funzione
alarm_thread è passato il puntatore alla struttura. Andava bene anche non usare la malloc nel
main, però poi si doveva copiare in una variabile locale a alarm_thread le informazioni sull'allarme
passate come argomento alla funzione.

Usando i semafori POSIX scrivete lo pseudocodice necessario per implementare un meccanismo a
barriera su n processi. Lo pseudocodice deve includere la funzione wall(n) necessaria per
realizzare la sincronizzazione e un main nel quale si generano gli n processi che usano la funzione

wall(n).
SOL:

```
// Variabili condivise da tutti i processi (essendo pseudocodice non pretendevo l'allocazione in memoria condivisa)
Sem_t W;
mutex m;
int arrive = 0 // Contatore condiviso dei processi arrivati alla barriera, inizialmente 0
// funzione eseguita da ciascun processo
Synch_wall(N){ // N: numero componenti della barriera
    down(m); // aggiornamento var. condivisa, va protetta
    arrive +=1; // ulteriore processo arrivato
    if(arrive<N) // se non sono l'ultimo
        up(m); // DEVO liberare mutex, altrimenti altri bloccati; prima di...
        down(W); // ... bloccarmi sulla barriera
    else // sono l'ultimo ...
        for(int i=1; i<N; i++) // ... sveglio tutti gli N-1 precedenti
            up(W);
}

int main(int argc, int argv){
    int i, N, pid;
    <controllo presenza di un singolo parametro, altrimenti errore>
    N = atoi(argv[1]); // N numero dei processi totali in barriera
    SemInit(W, 0) // Inizializzo semaforo a 0
    for(i=0; i<N; i++)
    if((pid=fork())==-1){
        printf("Errore nella fork\n");
        exit(-1);
    }
    if((pid==0){ // se verificato eseguo istruzioni processo figlio
        <Operazione qualsiasi da svolgere prima della barriera>
        Synch_wall(N); // invoco funzione barriera
        <Operazione qualsiasi da svolgere dopo la barriera>
        exit(0);
    }
    for(i=0; i<N; i++) // il main attende i processi figli generati
        wait();
}
```

Esempio di memoria condivisa

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <wait.h>

int k;
int m=10;

int main(int argc, char *argv[])
{
  pid_t n;
  int j;

  if (argc!=2) {
    fprintf(stderr,"Chiamare con un argomento numerico\n");
    exit(1);
  }

  k = atoi(argv[1]);

  id = shmget(IPC_PRIVATE, sizeof(int), 0600);
  m = shmatt(id, 0, 0)
  (*m)=10;

  if((n=fork())== (pid_t)-1){
    perror("fork fallita");
    exit(1);
  }
  else if (n==(pid_t)0)
        {/* processo figlio */
          printf("Il pid del figlio e' %d\n", getpid());
          //sleep(10);
          for (j=0;j<k;j++){


        (*m)++; //regione critica
          }
          exit(0);
        }
        else
        {/* processo padre */
          printf("Il pid del padre e' %d\n", getpid());
          for (j=0;j<k;j++){

           (*m)--;       //regione critica
          }
          wait(NULL);
          printf(" m = %d \n",m);
        }
}
```

Considerate la seguente funzione per la soluzione del problema dei filosofi:

```
# define N 5 /* numero di filosofi */
1. void philosopher(int i) /* i: indice del filosofo */
2. {
    3. while(TRUE){
        4. take_fork(i)
        5. take_fork((i+1) % N);
        6. eat();
        7. put_fork((i+1) % N)
        8. put_fork(i);
    9. }
11.}
```

Dettagliate le funzioni take_fork() e put_fork() indicando le operazioni di sincronizzazione al
loro interno. Indicate quale/i semafori vengono usati e come sono inizializzati. Questa soluzione è
soggetta a deadlock, per risolvere il problema modificate lo pseudocodice affinché siano ammessi al
più N-1 filosofi nella fase di acquisizione delle forchette. Aggiungete i semafori e/o variabili necessarie
e specificate come saranno inizializzati/e.
SOL

```
sem S[N]; // Dichiaro N semafori,uno per ciascuna forchetta
sem_init(S[i],1); /* inizializzo a 1 tutti gli N semafori di indice i
(per brevità non ho usato un ciclo) */
void take_fork(int i) /* i: indice della forchetta */
{
    down(S[i]);/*tento acquisizione forchetta, mi sospendo se è già in uso */
}
void put_fork(int i) /* i: indice della forchetta */
{
    up(S[i]);/* rilascio la forchetta */
}
```

Le chiamate a take_fork(i) quindi tentano di acquisire la forchetta di destra e di sinistra separatamente tramite delle down sui corrispondenti semafori.
Questa soluzione può generare un deadlock se ciascun filosofo riesce ad acquisire una sola forchetta (ad esempio quella alla sua destra) in quanto nessun altro potrà acquisire quella alla propria destra e mangiare.

Ci sono diverse soluzioni a questo problema, nel testo è richiesto espressamente di risolverlo permettendo ad al più N-1 filosofi di tentare di mangiare.
Per farlo è necessario usare un semaforo contatore inizializzato a N-1 e modificare la funzione philosopher in questo modo:

```
sem sem_cont; //* Dichiaro un semaforo contatore *
sem_init(sem_cont,N-1); /* inizializzo a N-1 il semaforo contatore */
1. void philosopher(int i) /* i: indice del filosofo */
2. {
3. while(TRUE){
    4. down(sem_cont); /*al massimo N-1 filosofi possono
    tentare di mangiare, l'N-esimo si sospende */
    5. down(S[i]);
    6. down(S[(i+1) % N)]);
    7. eat();
    8. up(S[(i+1) % N)];
    9. up(S[i]);
    10. up(sem_cont); /* incremento il numero di filosofi che possono
mangiare, eventualmente risvegliando quello precedentemente sospeso*/
11. }
12.}
```

La sincronizzazione a barriera serve in presenza di applicazioni che procedono per fasi: nessun processo/thread può proseguire nell'esecuzione della fase successiva finché tutti gli altri non abbiano completato la fase corrente. Questo comportamento è realizzato inserendo una barriera al termine di una fase. Quando i processi/thread raggiungono la barriera si sospendono in attesa che tutti gli altri raggiungano la barriera. Nel momento in cui l'ultimo raggiunge la barriera tutti potranno proseguire l'esecuzione della fase successiva. Usando i semafori POSIX scrivete lo pseudo-codice necessario per implementare un meccanismo a barriera su n thread, dove n è noto a priori. Lo pseudo-codice deve includere la funzione wall(n) necessaria per realizzare la sincronizzazione e un main nel quale si generano gli n thread che usano la funzione wall(n). Descrivere le differenze di implementazione della vostra soluzione nel caso usaste i processi al posto dei thread.

```
Sem S;
mutex m;
SemInit(S, 0) // Inizializzo semaforo a 0
SemInit(m, 0) // Inizializzo semaforo a 1
shared int arrive = 0 // Contatore condiviso dei processi arrivati alla barriera,
inizialmente 0
// funzione eseguita da ciascun processo/thread
Synch_wall(N){ // N: numero componenti della barriera
    down(m); // aggiornamento var. condivise va protetto
    arrive ++; // ulteriore processo arrivato
    if(arrive<N){ // se non sono l'ultimo
        up(m); // DEVO liberare mutex, altrimenti altri bloccati; prima di….
        down(S);
    } // …bloccarmi sulla barriera
    else { // sono l'ultimo …
        for(int i=1; i<N; i++) // … sveglio tutti gli N-1 precedenti
            up(S);
        arrive =0;
        up(m)
    }
}
```

Nell'implementazione tramite thread le variabili che devono essere condivise (numero thread finora arrivati alla barriera e semaforo) devono essere dichiarate globali. Nel caso dei processi tali variabili si devono esplicitamente allocare in memoria condivisa usando le funzioni di libreria shm_open()e mmap().

Pseudocodice barriera con processi e variabili condivise

```
shered sem_t Sem; // semaforo condiviso tra i processi
shared sem_t mutex; // mutex per proteggere var condivisa C
shared int C=0; // conta i processi arrivati alla barriera


wall(int n){
// n numero totale processi

  down(mutex);
  C++;

  if (C<n) {
    up(mutex);
    down(Sem);
  } else
  {
    for i = 1 to N-1
      up(Sem);
    up(mutex)
  }



}


main(){

  N  = 3;
  sem_init(Sem, 0);
  sem_init(mutex, 1);
  for i=1 to N{
    pid = fork()
      if figlio{

        <processamento locale>
        wall(N);

        <success process locale>

      }
  }
}
```

Shell con esecuzione in background

```
void runcommand(char **cline, int amp)  /* esegue un comando */
{
  pid_t pid;
  int exitstat,ret;

  pid = fork();
  if (pid == (pid_t) -1) {
     perror("smallsh: fork fallita");
     return;
  }

  if (pid == (pid_t) 0) {   /* processo figlio */

    /* esegue il comando il cui nome e' il primo elemento di cline,
       passando cline come vettore di argomenti */

    execvp(*cline,cline);
    perror(*cline);
    exit(1);
  }

  /* non serve "else"... ma bisogna aver capito perche' :-)  */

  /* qui aspetta nel caso non ci sia un &
    altrimenti non aspetta */

  if(amp == 0)   /* non eseguo in background e attendo altrimenti non attendo*/
  {
    ret = wait(&exitstat);
    if (ret == -1) perror("wait");
  }
}

int main()
{
  while(userin(prompt) != EOF)
    procline();
  return 0;
}
```

Soluzione con tutti i filosofi a tavola senza escluderne uno ma evitando il deadlock

```
/* 5 filosofi con possibile deadlock */
#define NFILOSOFI 5

sem_t *sem;                    /* puntatore array di semafori -> uno per filosofo*/
pthread_t threads[NFILOSOFI]; /* array dei threads */

char *stato; /* puntatore ad array di 5 caratteri in
            memoria condivisa per lo stato dei 5 filosofi,
        T = thinking, H = hungry, E = eating */

/* stampa lo stato dei filosofi */
void print()
{
    int i;
    for (i = 0; i < NFILOSOFI; i++)
        putchar(stato[i]);
    printf("\n");
}

void *proc(void *i_generic)
{
    int i = *((int*)i_generic); /* i = numero del filosofo e forchetta associata
(converto da puntatore generico a int) */
    int j, n;
    int m;

    m = 50000000 + 30000000 * i; /* tempo per cui usera' le forchette;
      i filosofi con i piu' alto sono piu' lenti a mangiare, questo differenzia
      un pochino il loro comportamento e su qualche sistema favorisce
      il verificarsi del deadlock */

    for (n = 0; n < 10; n++)
    {
        /* Il filosofo aspetta */
        printf("FILOSOFO %d E' AFFAMATO:       ", i);
        stato[i] = 'H'; /* Hungry */
        print();
        sem_wait(&sem[i]);

        /* se il filosofo attuale è affamato, e quello alla sua sx e dx non è
affamato può mangiare*/
        if (stato[i] == 'H' && stato[i + 1] != 'E' && stato[(i + 1) % NFILOSOFI] !=
'E')
        {
            /* Il filosofo prende la forchetta sx */
            printf("FILOSOFO %d HA PRESO LA FORCHETTA %d \n", i, i);
            /* Il filosofo prende la forchetta dx */
```

```
            printf("FILOSOFO %d HA PRESO LA FORCHETTA %d \n", i, (i + 1) %
NFILOSOFI);
            /* Il filosofo mangia */
            stato[i] = 'E';

            /* mette in attesa il prossimo filosofo, mentre mangia */
            sem_wait(&(sem[(i + 1) % NFILOSOFI]));
            printf("FILOSOFO %d MANGIA:       ", i);
            print();
            /* ogni processo usa le risorse per un tempo diverso */
            for (j = 0; j < m; j++);
        }

        /* il filosofo smette di mangiare rilasciando forchetta sx e dx */
        stato[i] = 'T';
        printf("FILOSOFO %d SMETTE:       ", i);
        print();
        sem_post(&(sem[(i + 1) % NFILOSOFI]));
        sem_post(&(sem[i]));
    }

    pthread_exit(NULL);
}

int main()
{
    int i;
    sem = malloc(NFILOSOFI * sizeof(sem_t));
    stato = malloc(NFILOSOFI * sizeof(char));

    /* inizializzo i semafori per sincronizzarsi sulle forchette */
    for (i = 0; i < NFILOSOFI; i++)
    {
        sem_init(&sem[i], 1, 1);
        stato[i] = 'T'; /* 1 stato : thinking */
    }

    /* I filosofi "si siedono a tavola" -> creo i thread*/
    for (i = 0; i < NFILOSOFI; i++)
        pthread_create(&threads[i], NULL, proc, (void*)&i);

    /* i filosofi mangiano 1 alla volta, aspettandosi a vicenda */
    for (i = 0; i < NFILOSOFI; i++)
    {
        pthread_join(threads[0], NULL);
        printf("Terminato thread %ld\n", threads[0]);
    }

    return 0;
}
```

Soluzione problema dei filosofi con semafori privati

```
#define N              5                /* number of philosophers */
#define LEFT           (i+N-1)%N        /* number of i's left neighbor */
#define RIGHT          (i+1)%N          /* number of i's right neighbor */
#define THINKING       0                /* philosopher is thinking */
#define HUNGRY         1                /* philosopher is trying to get forks */
#define EATING         2                /* philosopher is eating */
typedef int semaphore;                  /* semaphores are a special kind of int */
int state[N];                           /* array to keep track of everyone's state */
semaphore mutex = 1;                    /* mutual exclusion for critical regions */
semaphore s[N];                         /* one semaphore per philosopher */

void philosopher(int i)                 /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {                      /* repeat forever */
        think( );                       /* philosopher is thinking */
        take_forks(i);                  /* acquire two forks or block */
        eat( );                         /* yum-yum, spaghetti */
        put_forks(i);                   /* put both forks back on table */
    }
}
```

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                       /* enter critical region */
    state[i] = HUNGRY;                  /* record fact that philosopher i is hungry */
    test(i);                            /* try to acquire 2 forks */
    up(&mutex);                         /* exit critical region */
    down(&s[i]);                        /* block if forks were not acquired */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                       /* enter critical region */
    state[i] = THINKING;                /* philosopher has finished eating */
    test(LEFT);                         /* see if left neighbor can now eat */
    test(RIGHT);                        /* see if right neighbor can now eat */
    up(&mutex);                         /* exit critical region */
}

void test(i)                            /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

"ovviamente" down(&s[i]) è dopo up(&mutex)

per non fermarsi quando fa down, in take_forks;
o per svegliare il vicino, in put_forks;