UNIVERSITA' DEGLI STUDI DI MODENA E REGGIO EMILIA

CORSO DI LAUREA MAGISTRALE DI INGEGNERIA INFORMATICA
Anno Accademico 2020/2021
Tesina - PROGETTAZIONE SISTEMI OPERATIVI

Risoluzione di N 5 programmi utilizzando il linguaggio C con l'ausilio della libreria pthread.h e del costrutto Monitor per la sincronizzazione tra processi

Elaborato da:
Casalini Andrea
N matr: 162435

INDICE

CAPITOLO 1 – INTRODUZIONE

Questa tesina ha lo scopo di spiegare l'implementazione in linguaggio C di 5 esercizi sulla concorrenza.
I programmi sono stati forniti in pseudo-Pascal, e l'obiettivo è la corretta gestione delle risorse condivise attraverso il controllo della concorrenza e della cooperazione tra thread.
Quindi bisogna mantenere in uno stato consistente tutte le strutture dati del processo attraverso il costrutto monitor, cioè uno strumento di alto livello che nello standard POSIX è composto da un semaforo binario di mutua esclusione detto "mutex" e da "condition variables". Queste ultime gestiscono la sincronizzazione tra processi attraverso un controllo su una condizione che se è soddisfatta permette l'accesso alla risorsa richiesta al processo corrente.
La sincronizzazione avviene attraverso due primitive "wait" e "signal", dove solitamente la primitiva "wait" di una condition variable viene posta all'interno di un ciclo while che controlla una certa condizione. Finché questo controllo avrà esito "vero" il processo si bloccherà su questa "wait" quindi dovrà ancora attendere prima di poter aver accesso alla risorsa richiesta.
Per lo sviluppo del codice è stato scelto di utilizzare la libreria <pthread.h> in ambiente UNIX.

CAPITOLO 2

In ogni esercizio le funzioni di start routine presentano un ciclo infinito; quindi, i thread non ritorneranno mai un valore reale al main. All'interno di queste funzioni sono presenti delle funzioni di sleep() per simulare le varie attese/esecuzioni delle varie azioni.
Per avviare il programma bisognerà passare da riga di comando anche il numero di thread che si ha intenzione di creare per la simulazione corrente.
Per terminare le varie simulazioni bisogna eseguire un arresto forzato attraverso il comando ctrl-C.
Inoltre, in tutte le soluzioni sono state utilizzate le seguenti librerie.

- `#include <pthread.h>`
- `#include <semaphore.h>`
- `#include <math.h>`
- `#include <stdio.h>`
- `#include <stdlib.h>`
- `#include <unistd.h>`
- `#include <stdbool.h>`

2.1 Rotonda Stradale

2.1.1 TESTO

Si consideri una rotatoria nel mezzo di una confluenza tra un numero N (per esempio 6) di strade.
Secondo il vigente codice stradale
- il verso di rotazione è antiorario;
- un'auto ha la precedenza all'ingresso della rotonda e, una volta nella rotonda, deve dare la precedenza alle auto che vogliono accedere dai rami intermedi.

Poiché la rotatoria ha un ovvio limite fisico di capacità (il numero massimo di auto contenute è CMAX) è possibile una situazione di deadlock: la rotonda è piena di auto che non possono uscire perché in attesa di fare entrare auto dai lati intermedi, le auto dai lati intermedi non possono d'altronde entrare per problemi di capacità.
Si implementi una politica di gestione della rotonda, facendo uso del costrutto Monitor, che eviti situazioni di deadlock. Descrivere la struttura dei processi macchine. Discutere eventuali soluzioni dotate di maggiore parallelismo e i loro problemi.

2.1.2 IMPLEMENTAZIONE E STRATEGIE ADOTTATE

Riportiamo le variabili globali utilizzate per lo svolgimento corretto dell'esercizio 2.1 e la funzione di inizializzazione di tali variabili myInit().

```
#define N 4     /*numero rami della rotonda*/
#define NMAX 20 /*numero massimo di auto*/

pthread_cond_t enter[N];        /*condition variable per processi che entrano in rotonda*/
pthread_cond_t daiprec[N];      /*condition variable per processi gia in rotonda che danno la precedenza*/
pthread_mutex_t mutex;          /*semaforo binario per la mutua esclusione*/
int content[N];                 /*sospesi in ogni ramo*/
int cont;                       /*numero car in rotonda*/
int sospesidaiprec[N];          /*sospesi per dare la precedenza a chi entra in rotonda*/
int sospesienter[N];            /*sospesi per entrare in rotonda*/

void myInit(){
    pthread_mutex_init(&mutex, NULL);
    for(int i=0;i<N;i++){
        pthread_cond_init(&enter[i], NULL);
        pthread_cond_init(&daiprec[i], NULL);
        content[i]=0;
        sospesidaiprec[i]=0;
        sospesienter[i]=0;
    }
    cont=0;
}
```

Nel main vengono creati i “NUM_THREADS” (valore passato come parametro al programma) attraverso la funzione pthread_create().
Attraverso una #define andiamo a impostare il numero N di uscite/ingressi che avrà la nostra rotonda nella simulazione corrente.
E con la #define NMAX andiamo a settare il numero massimo di veicoli.
Ogni thread ad ogni iterazione andrà a definire in modo random a quale ingresso entrerà nella rotonda e a quale uscirà, attraverso la funzione rand().
Sono state utilizzate due tipi di coda per ogni ingresso/uscita dalla rotonda, una per gestire l’ingresso in rotonda e l’altra per gestire le precedenze da fornire agli altri thread che vogliono entrare in rotonda.
È stato utilizzato un mutex per gestire la mutua esclusione alle variabili condivise come “cont” che rappresenta il numero di auto all’interno della rotonda o i contatori dei processi sospesi.

```
void *auto(void*id){
    int *pi = (int  *)id;
    int *ptr;
    ptr = (int *)malloc(sizeof(int));
    if (ptr == NULL){
        printf("Problemi con l'allocazione di ptr\n");
        exit(-1);
    }
    int i;
    int o;
    int k=0;
    while(1){
        /*decido random quale è l'ingresso e quale è l'uscita*/
        i= (rand() % (N)) ;
        o= (rand() % (N)) ;
        /*arrivo all ingresso della rotatoria*/
        printf("Utente-[Thread%d e identificatore %lu] ENTRO ALL'INGRESSO[%d] E VOGLIO USCIRE ALLA [%d] (iter. %d)\n", *pi, pthread_self(),i,o,k);
        ingresso(i);
        /*entro in rotatoria*/
        printf("Utente-[Thread%d e identificatore %lu] RUOTO (iter. %d)\n", *pi, pthread_self(), k);
        ruota(i,o);
        sleep(5);
        /*esco dalla rotatoria*/
        printf("Utente-[Thread%d e identificatore %lu] ESCO  alla [%d]  (iter. %d)\n", *pi, pthread_self(),o,k);
        esci(o);
        k++;
        sleep(2);/*tempo prima che l'utente rientri in rotatoria*/
    }
}
```

La funzione “Auto” chiamata da ogni thread entra in un ciclo infinito che ricordiamo può essere interrotto solo attraverso l’esecuzione da terminale del comando ctrl-C. All’interno di ogni ciclo, ogni thread eseguirà tre operazioni attraverso tre differenti funzioni (“ingresso()”,”ruota()”,”esci()”).

Nella funzione “ingresso()” il processo corrente controlla di poter accedere alla rotatoria verificando che nel caso in cui entrasse anche l’auto corrente non si superi il numero massimo di veicoli. Nel caso in cui venga violato tale vincolo la funzione si arresterà su una “wait” in attesa di essere svegliata quando un veicolo già all’interno uscirà. Nel caso opposto invece viene aggiunto il veicolo che entrerà in rotatoria.

Nella funzione “ruota()” il processo corrente è già entrato in rotatoria e deve verificare ogni volta che passa davanti ad una uscita se ci sono dei veicoli in coda (in quella uscita) a cui dare la precedenza in ingresso.

Nella funzione “esci()” il processo corrente esce dalla rotatoria quindi viene aggiornato il numero di veicoli correntemente all’interno. Infine vengono segnalati i processi che possono essere riattivati che erano sospesi nella funzione “ingresso()”.

2.1.3 EVITARE STARVATION

Per evitare la starvation, la riattivazione dei processi esterni bloccati non è sempre nello stesso ordine così da non favorire i processi che entrano da rami della rotatoria con indice più basso.
La politica adottata prevede la riattivazione a partire sempre dal successivo rispetto all’uscita del processo corrente.

```
void esci(int o){
    pthread_mutex_lock(&mutex);
    cont--;
    if (cont<NMAX && cequalcunoincoda()){
        for(int j=0;j<N;j++){
            int tmp=0;
            //al posto di j mettere (j+o+1)/(N-1) per evitare starvation
            while(cont<NMAX &&tmp<sospesienter[(j+o+1)/(N-1)]){
                tmp++;
                pthread_cond_signal(&enter[(j+o+1)/(N-1)]);
            }
        }
    }
    pthread_mutex_unlock(&mutex);
}
```

2.2 Stanza con ingresso unico da due corridoi a senso unico

2.2.1 TESTO

Un locale possiede due accessi o corridoi, per consentire l'ingresso e l’uscita degli utenti. La protezione civile ha previsto un numero massimo di persone presenti nel locale (CAP).
Ogni accesso può, in un certo istante, permettere il passaggio o in ingresso o in uscita ad al massimo un certo numero di utenti (MAX).
In un dato istante, i gruppi che stanno passando attraverso un corridoio sono tutti nello stesso verso. Inoltre, se il corridoio è pieno, si ritarda ogni accesso a richieste ulteriori.
Gli utenti possono anche presentarsi in gruppi (di consistenza numerica inferiore a MAX). Si noti che il gruppo entra tutto o aspetta: non è possibile quindi un passaggio frazionato di un gruppo.
Si descriva la politica di sincronizzazione per i corridoi ed il locale, usando almeno uno dei seguenti costrutti (facoltativamente più di uno):
* monitor, regione critica condizionale.

La politica di sincronizzazione e la conseguente soluzione devono essere adeguatamente commentata. Inoltre, si descriva la politica scelta discutendo adeguatamente riguardo:
- alle possibilità di starvation
- alla possibilità di lasciare sottoutilizzate le risorse.

2.2.2 IMPLEMENTAZIONE E STRATEGIE ADOTTATE

Riportiamo le variabili globali utilizzate per lo svolgimento corretto dell'esercizio 2.2 e la funzione di inizializzazione di tali variabili myInit().

```
#define CAPAC 10    /*capacità stanza*/
#define MAX 3       /*capacità corridoio*/

int corridoio;      /*numero di corridoio 1 o 2*/
int gruppo;         /*numero di persone che compongono il gruppo da 1 a MAX*/
int contatt;        /*elementi in attesa della sala*/
int cap;          /*capacità corrente della sala*/

typedef enum{
    in, out
}dir;

dir direz[2];                /*2 rappresenta il numero di corridoi*/
int nutenti[2];              /*2 rappresenta il numero di corridoi*/
pthread_mutex_t mutex;       /*semaforo binario di mutua esclusione*/
pthread_cond_t attsala;      /*coda in attesa di capienza sufficiente nella stanza*/
pthread_cond_t codain[2];    /*2 rappresenta il numero di corridoi*/
pthread_cond_t codaout[2];   /*2 rappresenta il numero di corridoi*/
int attesa_coda_in[2];
int attesa_coda_out[2];

void myInit(){
    for(int i=0;i<2;i++){
        nutenti[i]=0;
        direz[i]=in;
        attesa_coda_in[i]=0;
        attesa_coda_out[i]=0;
        pthread_cond_init(&codain[i], NULL);
        pthread_cond_init(&codaout[i], NULL);
    }
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&attsala, NULL);
    contatt=0;
    cap=0;
}
```

Nel main vengono creati i "NUM_THREADS" (valore passato come parametro al programma) attraverso la funzione pthread_create().
Attraverso una #define andiamo a settare la capacità CAPC della stanza e la capacità MAX dei corridoi. Il numero di corridoi è fissato a 2 da specifiche del problema.
Ogni thread ad ogni iterazione andrà a definire in modo random con quale corridoio entreremo ed usciremo dalla stanza e il numero di componenti di ogni gruppo che entra/esce dalla stanza (deve essere minore di MAX) attraverso la funzione rand().
Sono state utilizzate 2 variabili per gestire le code generate in ingresso e in uscita da entrambi i corridoi.
È stato utilizzato un mutex per gestire la mutua esclusione alle variabili condivise come "cap" che rappresenta la capacità corrente della sala o i contatori dei processi sospesi delle due code.

```
void *utente(void*id){
    int *pi = (int  *)id;
    int *ptr;
    ptr = (int *)malloc(sizeof(int));
    if (ptr == NULL)
    {
        printf("Problemi con l'allocazione di ptr\n");
        exit(-1);
    }
    int i=0;
    while(1){
        /*attribuisco un numero casuale di bagagli ad ogni utente compreso tra 1 e N*/
        int n= (rand() % (MAX)) + 1 ;   /*numero utenti nel gruppo*/
        /*attribuisco un numero casuale per il corridoio scelto o 0 o 1*/
        int c=(rand() % (2));           /*numero corridoio scelto*/
        /* entrano*/
        printf("Utente-[Thread%d e identificatore %lu] ENTRIAMO IN N.[%d] NEL CORRIDOIO [%d] (iter. %d)\n", *pi, pthread_self(),n,c,i);
        INcorrAccesso(c,n);
        sleep(1);/*transita*/
        INcorrRilascio(c,n);
        /*aspetto*/
        printf("Utente-[Thread%d e identificatore %lu] ASPETTO                (iter. %d)\n", *pi, pthread_self(), i);
        sleep(3);
        /*escono*/
        printf("Utente-[Thread%d e identificatore %lu] ESCONO                 (iter. %d)\n", *pi, pthread_self(), i);
        OUTcorrAccesso(c,n);
        sleep(1);/*transita*/
        OUTcorrRilascio(c,n);
        i++;
        sleep(2);/*tempo prima che l'utente rientri in fila per entrare nella sala*/
    }

}
```

La funzione “utente” chiamata da ogni thread entra in un ciclo infinito che ricordiamo può essere interrotto solo attraverso l’esecuzione da terminale del comando ctrl-C. All’interno di ogni ciclo, ogni thread eseguirà quattro operazioni attraverso quattro differenti funzioni (“INcorrAccesso()”, ”INcorrRilascio()”, ”OUTcorrAccesso()”, ”OUTcorrRilascio()”). Inoltre sono state inserite delle sleep() per simulare il tempo di transito nei corridoi e il tempo di permanenza nella stanza dei vari gruppi.

La funzione INcorrAccesso() controlla prima che il gruppo corrente ci stia nella sala e nel caso in cui tale condizione non fosse rispettata si blocca su una “wait”. Poi controlla di poter entrare nel corridoio, in quanto magari quest’ ultimo è già saturo oppure è adibito al senso opposto.
Quando si superano tutti i controlli si aggiornano le variabili che tengono traccia di quante persone ci sono nel corridoio e nella sala. Allo stesso tempo viene anche aggiornato il verso che ha assunto il corridoio preso.

Nella funzione INcorrRilascio () il gruppo abbandona il corridoio per entrare nella sala; quindi, viene nuovamente aggiornato il numero di utenti che sono nel corridoio. Nel caso in cui il corridoio si fosse svuota, segnalo che tale corridoio lo si può usare per far uscire utenti dalla sala e al contrario se ci sono ancora utenti in transito verso la sala in quel corridoio si segnala che possono entrare altri utenti nel corridoio per entrare in sala.

La funzione OUTcorrAccesso() aggiorna il numero di utenti nella stanza e riattiva tutti i processi che si erano bloccati per mancanza di capacità della stanza. Poi controllo che gli utenti possano entrare nel corridoio, nel caso in cui non potessero mi blocco (ipoteticamente in quel momento il corridoio è adibito nel senso opposto cioè IN e stanno transitando degli utenti).
Se tutti i controlli vanno a buon fine aggiorno il numero di utenti nel corridoio e aggiorno la direzione del corridoio preso per uscire.

La funzione OUTcorrRilascio() aggiorna la capienza del corridoio usato per uscire e segnala che il corridoio può essere usato da altri gruppi per uscire e per altri gruppi per entrare.

2.2.3 SOLUZIONE ALTERNATIVA

Nella prima soluzione la sala è impegnata anche da utenti che non sono autorizzati ad entrare a causa della direzione opposta del corridoio quindi è stata proposta una soluzione alternativa dove i due controlli della "INcorrAccesso()" vengono svolti insieme per ovviare a questo problema.

```
void INcorrAccesso(int c,int n){
    pthread_mutex_lock(&mutex);

    /*controllo se possono entrare se no WAIT corridoio*/
    while(((direz[c]!=in) && (nutenti[c]!=0))||((direz[c]==in)&&((nutenti[c]+n)>MAX))||((direz[c]==in)&&(attesa_coda_out[c]!=0)&&(cap+n>CAPAC))){
        attesa_coda_in[c]++;
        pthread_cond_wait(&codain[c],&mutex);
        attesa_coda_in[c]--;
    }
    cap+=n;
    nutenti[c]+=n;
    direz[c]=in;
    printf("il processo con pid [%lu] entra nel corridoio [%d] e con [%d] componenti del gruppo. NEL CORRIDOIO SONO IN [%d]\n",pthread_self(),c,n,nutenti[c]);
    pthread_mutex_unlock(&mutex);
}
```

Utilizzando questa soluzione andiamo a ridurre anche il numero di variabili utilizzate in quanto non ci serve più avere una coda di attesa solo per l'ingresso in sala e allo stesso tempo il contatore dei sospesi per l'ingresso in sala.

2.2.4 EVITARE STARVATION

2.3 Pompe di benzina

2.3.1 TESTO

Un distributore di benzina ha a disposizione P pompe e una cisterna da L litri. Le automobili arrivano al distributore e richiedono un certo numero di litri di benzina (diverso per ogni automobile e minore di L). La cisterna è rifornita da una autobotte che la riempie fino alla capacità massima e solo se nessuna automobile sta facendo benzina, avendo comunque la priorità sulle automobili. La automobili possono fare benzina solo se c'è una pompa libera, se la quantità di benzina richiesta è disponibile nella cisterna e se l'autobotte non sta riempiendo la cisterna. Dopo aver fatto benzina, liberano la pompa utilizzata.
Si implementi una soluzione usando il costrutto monitor per modellare il distributore di benzina e i processi per modellare le automobili e l'autobotte e si descriva la sincronizzazione tra i processi. Nel rispettare i vincoli richiesti, si cerchi di massimizzare l'utilizzo delle risorse. Si discuta se la soluzione proposta può presentare starvation e in caso positivo per quali processi, e si propongano modifiche e/o aggiunte per evitare starvation.

2.3.2 IMPLEMENTAZIONE E STRATEGIE ADOTTATE

Riportiamo le variabili globali utilizzate per lo svolgimento corretto dell'esercizio 2.3 e la funzione di inizializzazione di tali variabili myInit().

```
#define P 4         /*numero pompe*/
#define L 1000      /*numero litri disponibili dal benzinaio*/
#define MAXRICHIESTA 100 /*richeista massima dei veicoli*/
int benzdisp;       /*benzina disponibile*/
int pompedisp;      /*pompe disponibili*/
int sospesi;        /*numero automobili sospese*/
pthread_cond_t codaAM;  /*coda automobili*/
pthread_cond_t codaAB;  /*coda autobotte*/
pthread_mutex_t mutex;  /*semaforo binario per la mutua esclusione*/
int sospesaAB;      /*sospensione autobotte*/

void myInit(){
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&codaAM, NULL);
    pthread_cond_init(&codaAB, NULL);
    sospesaAB=0;
    sospesi=0;
    benzdisp=L;
    pompedisp=P;
}
```

Nel main vengono creati i “NUM_THREADS” di tipo automobile (valore passato come parametro al programma) e un solo thread di tipo autobotte attraverso la funzione pthread_create().
Attraverso una #define andiamo a settare il numero di pompe che possiede il benzinaio e il numero di litri che può contenere la cisterna di benzina che possiede il benzinaio.
Ogni thread automobile ad ogni iterazione andrà a definire in modo random quanti litri andare a richiedere al benzinaio attraverso la funzione rand().
Sono state utilizzate due code per gestire i processi autobotte e i processi automobile.
È stato utilizzato un mutex per gestire la mutua esclusione alle variabili condivise come “benzdisp” e “pompedisp” che rappresentano quanta benzina è rimasta nella cisterna del benzinaio e quante pompe sono libere.

```
void *autobotte(void*id){
    int *pi = (int  *)id;
    int *ptr;
    ptr = (int *)malloc(sizeof(int));
    if (ptr == NULL){
        printf("Problemi con l'allocazione di ptr\n");
        exit(-1);
    }
    int i=0;
    while(1){
        printf("AUTOBOTTE-[Thread%d e identificatore %lu] (iter. %d)\n", *pi, pthread_self(),i);
        Rifornisci();
        i++;
        sleep(10);
    }
}
```

```c
void *automobile(void*id){
    int *pi = (int  *)id;
    int *ptr;
    ptr = (int *)malloc(sizeof(int));
    if (ptr == NULL){
        printf("Problemi con l'allocazione di ptr\n");
        exit(-1);
    }
    /*attribuisco un numero casuale di bagagli ad ogni utente compreso tra 1 e N*/
    int i=0;
    while(1){
        int l= (rand() % (MAXRICHIESTA)) + 1 ;   /*numero litri richiesti dal veicolo*/
        /* entrano*/
        printf("Automobile-[Thread%d e identificatore %lu] RICHIEDO [%d] LITRI (iter. %d)\n", *pi, pthread_self(),l,i);
        Richiedi(l);
        /*aspetto*/
        printf("Automobile-[Thread%d e identificatore %lu] ASPETTO              (iter. %d)\n", *pi, pthread_self(), i);
        sleep(3);
        /*escono*/
        printf("Automobile-[Thread%d e identificatore %lu] RILASCIO             (iter. %d)\n", *pi, pthread_self(), i);
        Rilascia(l);
        /*tempo prima che l'utente rientri in fila per entrare dalbenzinaio*/
        i++;
        sleep(2);
    }
}
```

La funzione “automobile”, chiamata da tutti i “NUM_THREAD” thread di tipo automobile, entra in un ciclo infinito che ricordiamo può essere interrotto solo attraverso l’esecuzione da terminale del comando ctrl-C. All’interno di ogni ciclo, ogni thread eseguirà due operazioni attraverso due differenti funzioni (“richiedi()”, ”rilascia()”). Inoltre sono state inserite delle sleep() per simulare il tempo necessario per fare il rifornimento e il tempo prima che un veicolo abbia una nuova necessità di fare il pieno.
La funzione “autobotte”, chiamata dal thread che simula l’autobotte, entra in un ciclo infinito che periodicamente fa passare l’autobotte a rifornire la cisterna del benzinaio attraverso la funzione rifornisci().

La funzione Rifornisci() chiamata dal thread Autobotte si occupa di rifornire la cisterna del benzinaio quando nessuna automobile sta facendo il pieno. Nel caso in cui ci fossero automobili che stanno facendo il pieno il processo si blocca su una wait in attesa che il benzinaio sia vuoto. Quando l’autobotte rifornisce si aggiornano le variabili sul numero di litri di benzina disponibili. Infine vengono segnalati tutti i processi sospesi.

La funzione Richiedi() chiamata da tutti i thread Automobile verifica di poter entrare a fare il pieno. Per farlo devono essere soddisfatte tre condizioni:

- Ci devono essere pompe disponibili
- Ci deve essere abbastanza benzina nella cisterna del benzinaio per fare il pieno
- Non ci deve essere l’autobotte sospesa in attesa che si svuoti il benzinaio

Se non si superano questi controlli ci si blocca su una wait, invece nel caso opposto possiamo effettivamente fare il pieno quindi aggiornare il numero di pompe disponibili (una diviene dedicata al processo corrente) e il numero di litri disponibili nella cisterna del benzinaio (tolti i litri richiesti dal processo corrente).

La funzione Rilascia() simula la fine del rifornimento da parte dell’automobile che quindi rende di nuovo disponibile una pompa per altri utenti. Prima di tutto verifica se tutte le pompe sono libere e se lo sono risveglia l’autobotte per rifornire la cisterna. Infine risveglia tutti i processi automobile sospesi.

2.3.3 SOLUZIONE ALTERNATIVA

È stata implementata anche una versione alternativa dove sono state utilizzate due procedure per l’autobotte (invece di svolgere l’intro processo attraverso la funzione Rifornisci() abbiamo

utilizzato due funzioni Entra() ed Esci() separate all'interno del ciclo infinito dell'autobotte da una sleep() per simulare il tempo necessario per riempire la cisterna).

```
void *autobotte(void*id){
    int *pi = (int  *)id;
    int *ptr;
    ptr = (int *)malloc(sizeof(int));
    if (ptr == NULL){
        printf("Problemi con l'allocazione di ptr\n");
        exit(-1);
    }
    int i=0;
    while(1){
        printf("AUTOBOTTE-[Thread%d e identificatore %lu] (iter. %d)\n", *pi, pthread_self(),i);
        Entra();
        sleep(1); /*riempi la cisterna*/
        Esci();
        i++;
        sleep(10);
    }
```

È stato necessario introdurre una nuova variabile "riempimento" che andremo ad inizializzare all'interno della myInit() con valore inziale a false.

```
bool riempimento;   /*variabile per riempimento da autobotte*/

void myInit(){
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&codaAM, NULL);
    pthread_cond_init(&codaAB, NULL);
    sospesaAB=0;
    sospesi=0;
    benzdisp=L;
    pompedisp=P;
    riempimento=false;
}
```

Questo valore viene cambiato in True mentre l'autobotte rifornisce la cisterna così da non far entrare veicoli anche se ci sono pompe disponibili.
Infatti nel ciclo while della funzione richiedi() abbiamo introdotto un nuovo controllo che va a verificare che l'autobotte non stia rifornendo la cisterna.

```
void Richiedi(int l){
    pthread_mutex_lock(&mutex);
    while(pompedisp==0 || benzdisp<l || sospesaAB !=0 || riempimento){
```

2.3.4 EVITARE STARVATION

Nel caso reale la capacità della cisterna è così grande che è impossibile che la richiesta di un solo veicolo la svuoti quasi completamente, ma nel nostro caso di studio può capitare che veicoli ne richiedano grandissime quantità.
È stato individuato il rischio di starvation per i processi automobile che richiedono molti litri di benzina, che quando si risvegliano anche dopo il riempimento della cisterna non fanno in tempo a fare rifornimento perché non trovano capacità sufficiente disponibile nella cisterna.
È stata proposta una seconda soluzione alternativa per risolvere questo rischio starvation attraverso l'implementazione di due code differenti per i processi automobile.

La nuova coda definita come “codaAMU” (automobili urgenti) è formata dalle automobili che richiedono più dell’80% della capienza massima della cisterna.
Al momento del rifornimento dell’autobotte se ci saranno processi sospesi di tipo urgente saranno risvegliati prima di quelli con priorità “normale”.

```
void Rifornisci(){
    pthread_mutex_lock(&mutex);
    if(pompedisp<P){
        /*ci sono automobili che stanno facendo benzina*/
        sospesaAB=1;
        pthread_cond_wait(&codaAB,&mutex); /*prima sospendo*/
        sospesaAB=0;
        /*quandovengo rilasciato*/
    }
    benzdisp=L;
    printf("AUTOBOTTE HA RIFORNITO\n");
    /*risveglio automobili in coda*/
    if (sospesiU!=0){
        pthread_cond_signal(&codaAMU);
    }
    else{
        int s=sospesi;
        for(int i=0;i<s;i++)
            pthread_cond_signal(&codaAM);
    }

    pthread_mutex_unlock(&mutex);
}
```

2.4 Il deposito bagagli

2.4.1 TESTO

Si supponga di avere un deposito bagagli composto da V vani ognuno dei quali in grado di contenere N valigie. Gli utenti arrivano con un numero variabile (ma minore di N) di valigie, le depositano e, dopo un certo tempo, le ritirano. Tutte le valigie di uno stesso utente devono essere depositate all’interno di un unico vano, ma uno stesso vano può contenere le valigie di più utenti. Gli utenti che non riescono a depositare le valigie per problemi di capacità si pongono in attesa che si liberi dello spazio.
Si discutano le possibili politiche di gestione del deposito bagagli e se ne implementi una usando il costrutto monitor. Si discuta se nella politica implementata sono possibili casi di starvation e, in caso affermativo, si propongano soluzioni per eliminarli.

2.4.2 IMPLEMENTAZIONE E STRATEGIE ADOTTATE

Riportiamo le variabili globali utilizzate per lo svolgimento corretto dell’esercizio 2.4 e la funzione di inizializzazione di tali variabili myInit().

```
#define V 2  /*numero di vani*/
#define N 4  /*numero bagagli per vano*/

/*variabili globali*/
/* semaforo di mutua esclusione per l'accesso a tutte le variabili condivise
(simula il semaforo di mutua esclusiome associato ad una istanza di tipo monitor) */
pthread_mutex_t mutex;
pthread_cond_t coda;
int contatore_sospesi;      /*contatore di utenti in coda in attesa di risveglio*/
int cap_vano[V];         /*array struttura deposito*/

void myInit(){
    for(int i=0;i<V;i++)
        cap_vano[i]=0;
    contatore_sospesi=0;
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&coda, NULL);
}
```

Nel main vengono creati i “NUM_THREADS” (valore passato come parametro al programma) attraverso la funzione pthread_create().
Attraverso una #define andiamo a settare il numero di vani “V” esistenti all’interno del deposito e il numero “N” di bagagli che possono essere inseriti in ogni vano.
Ogni thread andrà a definire in modo random quanti bagagli l’utente deve depositare nel deposito (attraverso la funzione rand()).
E’ stata utilizzata una coda unica per gestire i processi sospesi.
È stato utilizzato un mutex per gestire la mutua esclusione alle variabili condivise come “cap_vano[V]” che rappresenta la struttura dati dell’intero deposito.

```
void *user(void*id){
    int *pi = (int  *)id;
    int *ptr;
    ptr = (int *)malloc(sizeof(int));
    if (ptr == NULL)
    {
        printf("Problemi con l'allocazione di ptr\n");
        exit(-1);
    }
    /*attribuisco un numero casuale di bagagli ad ogni utente compreso tra 1 e N, si può anche spostare all'interno del
    ciclo while cosi ogni volta  l'utente ha bagagli diversi*/
    int n_bagagli= (rand() % (N)) + 1 ;
    int i=0;
    while(1){
        /*cap_vano bagaglio*/
        int quale_vano=V;
        printf("Utente-[Thread%d e identificatore %lu] ENTRO CON N.[%d] BAGAGLI (iter. %d)\n", *pi, pthread_self(),n_bagagli,i);
        lascia(&quale_vano,n_bagagli);
        /*aspetto*/
        printf("Utente-[Thread%d e identificatore %lu] ASPETTO (iter. %d)\n", *pi, pthread_self(), i);
        sleep(5);
        /*ritiro bagaglio*/
        printf("Utente-[Thread%d e identificatore %lu] ESCO    (iter. %d)\n", *pi, pthread_self(), i);
        prendi(quale_vano,n_bagagli);
        i++;
        sleep(2);/*tempo prima che l'utente rientri nel deposito per depositare altri bagagli*/
    }
```

La funzione “user” chiamata da ogni thread entra in un ciclo infinito che ricordiamo può essere interrotto solo attraverso l’esecuzione da terminale del comando ctrl-C. All’interno di ogni ciclo, ogni thread eseguirà due operazioni attraverso due differenti funzioni (“lascia()”, ”prendi()”). Inoltre sono state inserite delle sleep() per simulare il tempo in cui i bagagli sono lasciati nel deposito e per simulare il tempo necessario prima che l’utente abbia una nuova necessità di lasciare i bagagli nel deposito.

La funzione lascia() controlla se c'è un vano disponibile che riesca a contenere tutti i bagagli dell'utente. Se non esiste di blocca su una wait.
Se esiste i bagagli vengono inseriti nel vano.

La funzione prendi() preleva i bagagli dai vani aggiornando la variabile sullo stato dei vani e successivamente rilascia le "signal" per risvegliare i processi sospesi.

2.4.3 GESTIONE EFFICACE DELLE RISORSE

Per ottenere un miglior utilizzo delle risorse è stata implementata una nuova versione.
Quando andiamo ad assegnare i bagagli dell'utente nel vano, quest'ultimo lo dovremmo andare a cercare in modo ciclico e non sempre partendo dal vano numero 1. Per fare ciò bisogna introdurre una nuova variabile "alterna_vano".

```
void lascia(int *quale_vano,int n_bagagli){
    pthread_mutex_lock(&mutex);
    *quale_vano=0;
    while(*quale_vano==0){
        for(int i=alterna_vano;i<(alterna_vano+V);i++){
            if(cap_vano[(i-1)%V]+n_bagagli<=N){
                *quale_vano=i;
            }
        }
        if(*quale_vano==0){
            /*non ho trovato un vano libero per i bagagli quindi devo attendere*/
            contatore_sospesi++;
            //printf("-->sono [%lu] e sono bloccato\n",pthread_self());
            pthread_cond_wait(&coda, &mutex);
            contatore_sospesi--;
        }
    }
    printf("INSERISCO BAGAGLIO NEL VANO %d e sono il thread con id %lu\n",*quale_vano,pthread_self());
    cap_vano[*quale_vano-1]+=n_bagagli;
    alterna_vano=(alterna_vano+1)%V;
    pthread_mutex_unlock(&mutex);
```

2.5 Ingresso e uscita da uno stadio

2.5.1 TESTO

Uno stadio ospita un torneo internazionale di calcio al quale partecipano squadre italiane e non (quindi straniere). L'orario delle partite del torneo prevede il continuo susseguirsi di incontri durante l'intero arco di ogni giornata. Si prevede quindi una continua affluenza di tifosi in ingresso ed in uscita dallo stadio.
Lo stadio è accessibile attraverso due corridoi (Cancello Nord e Cancello Sud) utilizzabili sia per l'ingresso che per l'uscita.
I tifosi accedono allo stadio in gruppi (ognuno dei quali deve essere considerato indivisibile nell'accesso e nell'uscita): a gruppi di consistenza numerica maggiore deve essere associata una maggiore priorità nell'accesso e nell'uscita dallo stadio.
Per motivi di sicurezza si è deciso di gestire ogni corridoio in modo tale che gruppi di tifosi di squadre italiane non possano incrociarsi (nello stesso corridoio) con gruppi di tifosi di squadre straniere: quindi, ogni corridoio può funzionare a doppio senso di percorrenza soltanto nel caso in cui gli utenti che lo attraversano siano tutti tifosi dello stesso tipo (cioè tutti italiani, oppure tutti stranieri). A questo proposito, si assuma che tutti i gruppi siano omogenei (o italiani o stranieri).
Per ragioni di ospitalità, inoltre, si è stabilito di favorire ragionevolmente (nell'accesso e nell'uscita dallo stadio) i gruppi di tifosi stranieri.

Lo stadio ha una capienza massima MAX_C (che esprime il numero massimo di tifosi consentito all'interno dello stadio) oltre la quale non è permesso l'accesso di ulteriori persone.
Infine, nel tentativo di evitare situazioni di saturazione, tifosi che desiderano uscire dallo stadio devono avere la precedenza sui tifosi che intendono entrare.
Si definisca una politica di gestione dello stadio che risponda alle specifiche date. Descrivere la politica e la si implementi usando il costrutto monitor.

2.5.2 IMPLEMENTAZIONE E STRATEGIE ADOTTATE

Riportiamo le variabili globali utilizzate per lo svolgimento corretto dell'esercizio 2.1 e la funzione di inizializzazione di tali variabili myInit().

```
#define MAX_C 100   //capienza stadio
#define MAX_G 45    //numero massimo di componenti di un gruppo
typedef enum{
    italiani, stranieri
}nazionalita;
typedef enum{
    nord, sud
}cancello;
int tifosi_in_stadio;   //numero di tifosi nello stadio
pthread_mutex_t mutex;  //semaforo binario per mutua esclusione
pthread_cond_t coda_in[2][2][MAX_G];    //struttura dati di condition variable per l'ingresso [corridoio][nazionalita][numeroPersone]
pthread_cond_t coda_out[2][2][MAX_G];   //struttura dati di condition variable per l'uscita [corridoio][nazionalita][numeroPersone]
int n_corridoio[2][2][2];            /*numero di persone in ogni corridoio*/
int sospesi_coda_in[2][2][MAX_G];       //struttura dati per contatore sospesi in ingresso [corridoio][nazionalita][numeroPersone]
int sospesi_coda_out[2][2][MAX_G];      //struttura dati per contatore sospesi in uscita [corridoio][nazionalita][numeroPersone]

void myInit(){
    pthread_mutex_init(&mutex, NULL);
    for(int i=0;i<2;i++){
        for(int k=0;k<2;k++){
            for(int s=0;s<2;s++){
                n_corridoio[i][k][s]=0;            //n_corridoio[corridoio][nazionalità][direzione]
            }
        }
    }
    for(int i=0;i<2;i++){
        for(int k=0;k<2;k++){
            for(int s=0;s<MAX_G;s++){
                sospesi_coda_in[i][k][s]=0;
                sospesi_coda_out[i][k][s]=0;
                pthread_cond_init(&coda_in[i][k][s], NULL);     //coda_in[corridoio][nazionalità][num]
                pthread_cond_init(&coda_out[i][k][s], NULL);
            }
        }
    }
    tifosi_in_stadio=0;
}
```

Nel main vengono creati i "NUM_THREADS" (valore passato come parametro al programma) attraverso la funzione pthread_create().
Attraverso una #define andiamo a impostare il numero MAX_C cioè la capienza massima dello stadio e MAX_G cioè la capienza massima dei corridoi (nord e sud).
Ogni thread ad ogni iterazione andrà a definire in modo random a quale ingresso entrerà nella stadio (se quello sud o quello nord) e a quale uscirà, il numero dei componenti del gruppo la nazionalità del gruppo attraverso la funzione rand().
Sono state utilizzate due strutture dati (una per l'ingresso e una per l'uscita) per gestire l'insieme di code individuate a seconda del cancello scelto, della tipologia di tifoso (italiano o straniero) e infine dal numero di tifosi che compongono quel gruppo che sta entrando/uscendo. Il numero di componenti del gruppo è fondamentale passarlo in quanto il problema esplicita l'esigenza di dare la priorità di ingresso/uscita agli stranieri, ma anche ai gruppi più consistenti.

È stato utilizzato un mutex per gestire la mutua esclusione alle variabili condivise come “tifosi_in_stadio” che rappresenta il numero totale di tifosi presenti all’interno dello stadio o le strutture dati dei contatori dei processi sospesi.

```
void *Tifosi(void*id){
    int *pi = (int  *)id;
    int *ptr;
    ptr = (int *)malloc(sizeof(int));
    if (ptr == NULL)···
    /*attribuisco un numero casuale di bagagli ad ogni utente compreso tra 1 e N*/
    int c_in;
    int c_out;
    int num;
    int tipo;

    int k=0;
    while(1){
        /*decisione cancello di ingresso*/
        c_in= (rand() % (2)) ;
        /*decisione cancello di uscita*/
        c_out= (rand() % (2)) ;
        /*componenti gruppo random*/
        num=(rand()%(MAX_G-1)+1);
        /*decisione random del tipo di gruppo se italiano o straniero*/
        tipo=(rand() % (2)) ;
        /*arrivo all ingresso dello stadio*/
        printf("Utente-[Thread%d e identificatore %lu] ENTRO ALL'INGRESSO[%d] E SONO DI TIPO [%d] IN [%d]   (iter. %d)\n", *pi, pthread_self(),c_in,tipo,num,k);
        acq_in(c_in,num,tipo);
        sleep(2);
        ril_in(c_in,num,tipo);
        /*entro in stadio*/
        printf("Utente-[Thread%d e identificatore %lu] RIMANE NELLO STADIO                              (iter. %d)\n", *pi, pthread_self(), k);
        sleep(5);
        /*esco dalla rotatoria*/
        printf("Utente-[Thread%d e identificatore %lu] ESCO  DALLO STADIO DALL USCITA [%d]               (iter. %d)\n", *pi, pthread_self(),c_out,k);
        acq_out(c_out,num,tipo);
        sleep(2);
        ril_out(c_out,num,tipo);
        k++;
        sleep(2);/*tempo prima che l'utente rientri in rotatoria*/
    }
}
```

La funzione “Tifosi” chiamata da ogni thread entra in un ciclo infinito che ricordiamo può essere interrotto solo attraverso l’esecuzione da terminale del comando ctrl-C. All’interno di ogni ciclo, ogni thread eseguirà quattro operazioni attraverso quattro differenti funzioni (“acq_in()”,”ril_in()”,”acq_out()”,”ril_out()”).

Nella funzione “acq_in()” il processo corrente verifica che il gruppo arrivato in ingresso in uno dei due cancelli possa entrare in relazione ai vincoli imposti dal problema:

- Lo stadio non deve sforare la sua capienza massima
- Nel corridoio non ci devono essere altri gruppi dell’altra tifoseria
- Non ci devono essere in attesa di uscire gruppi dell’altra tifoseria

Se i vincoli non sono soddisfatti la funzione si arresterà su una “wait” in attesa di essere riattivata da una “signal”.
Nel caso opposto il gruppo può entrare e vengono aggiornate le variabili che indicano quanti utenti ci sono nello stadio e quanti utenti stanno percorrendo i corridoi delle varie nazionalità.
Infine, viene rilasciata una “signal” verso un processo omologo per permettere il risveglio di più processi bloccati nelle stesse condizioni.

Nella funzione “ril_in()” il processo corrente aggiorna le variabili che rappresentano lo stato del corridoio, e segnala ai possibili gruppi di tifoseria opposta che possono usare tale corridoio per uscire.

Nella funzione “acq_out()” il processo corrente verifica che non ci siano gruppi di tifoseria opposta nel corridoio, se si ci si blocca su una “wait”. Nel caso opposto invece il gruppo uscirà dallo stadio e verranno aggiornate le variabili degli utenti all’interno dello stadio e degli utenti nel corridoio (il gruppo che deve uscire deve prima entrare nel corridoio).
Essendosi liberata della capacità nello stadio si procederà infine a segnalare in ingresso ai vari gruppi che possono entrare.
Come per la “acq_in()” anche qui andiamo a chiamare una “signal” un processo omologo per permettere il risveglio di più processi bloccati nelle stesse condizioni.

Nella funzione “ril_out()” il processo corrente esce dallo stadio definitivamente quindi viene aggiornata la variabile dello stato del corridoio usato per uscire e viene segnalato ai vari gruppi dell’altra tifoseria che possono entrare.

2.5.3 EVITARE STARVATION

Questa versione può presentare casi di starvation per i processi formati da gruppi piccoli, in quanto per specifiche del problema è richiesto che abbiano maggiore priorità i gruppi più grandi. Nel caso in cui si volesse risolvere il problema senza violare le specifiche del problema si può pensare che ad intervalli regolari di tempo andiamo a risvegliare i processi in ordine opposto quindi partiremmo dai gruppi più piccoli. Oppure si può pensare di aumentare il livello di priorità della coda dopo un certo numero di iterazioni dove il processo rimane sempre bloccato.