Università degli Studi di Modena e Reggio Emilia
Dipartimento di Ingegneria Enzo Ferrari

Corso di Laurea in Ingegneria Informatica

TESINA
PROGETTAZIONE DI SITEMI OPERATIVI

Relatrice
Prof.ssa Letizia Leonardi

Candidato
Marco Moroni

Anno Accademico 2021/2022

Indice

1 Introduzione

In questo documento si vanno a descrivere i due problemi assegnati per mail:

A — argomento 5;

B — esercizio 7.5.3, relativo all'argomento 8.

In questa tesi si utilizza, per semplicità, la denominazione di "Esercizio A" e "Esercizio B" rispettivamente per i due esercizi sopracitati.

1.1 Esercizio A

L'esercizio A richiede di parallelizzare un programma di somma. Il programma legge una lista di "task" da un file. Ogni task consiste in un carattere che definisce il codice di un'azione da intraprendere, e un numero. Il carattere codice può assumere i valori:

- "p" — tempo di processamento;
- "w" — tempo di attesa.

Il file di input simula diversi carichi di lavoro in un sistema a multiprocessore. Per questo esercizio il numero `n` che segue il codice, ha uno scopo diverso in base al valore del codice:

- "p" — il numero `n` indica il valore da processare in maniera atomica in 4 funzioni di aggregazioni diverse:
  - somma;
  - minimo;
  - massimo;
  - numero di valori dispari.
- "w" — il numero `n` indica i secondi di pausa del processore.

```
p 1
w 2
p 2
p 3
```

Per l'esempio sopracitato l'output che ne deriva è:

- somma: 6;
- minimo: 1;
- massimo: 3;
- numero di valori dispari: 2

1.2 Esercizio B

L'esercizio B è stato scritto da Grant Hutchins, ispirato da un'amica intenta a prendere la cittadinanza presso l'edificio storico "Faneuil Hall" di Boston. Per questo esercizio si prendono come riferimenti tre categorie diverse persone (thread):

- N immigrati;
- M spettatori;
- 1 giudice.

Gli immigrati aspettano in fila, effettuano il check-in, e dopodichè si siedono ad aspettare. Ad un certo punto entra il/la giudice nell'edificio. Mentre il/la giudice è all'interno, nessun altro può entrare, gli spettatori però possono lasciare l'edificio. Quanto tutti gli immigrati hanno effettuato il check-in, il/la giudice conferma la loro cittadinanza,e a questo punto ogni immigrato che è stato confermato dal/la giudice può ritirare il proprio certificato di cittadinanza americana. Il/la giudice lascia l'edificio dopo un periodo P casuale di tempo. Tutti coloro che sono stati confermati possono ritirare il proprio certificato e lasciare l'aula, mentre gli immigrati che non hanno fatto in tempo a farsi confermare dal giudice, dovranno ripresentarsi nuovamente in un altro momento, quindi lasciare l'edificio, rientrare, effettuare il check-in, sedersi ed aspettare nuovamente l'arrivo del/la giudice.

2 Soluzioni

2.1 Esercizio A

Per affrontare questo esercizio prima di tutto si è dovuto generare il file con all'interno l'elenco dei task.

Si è pensato per la risoluzione dell'esercizio di creare una struttura dati globale che contenesse al suo interno tutte le variabili condivise tra i vari thread:

- `sum` — somma;
- `min` — minimo;
- `max` — massimo;
- `n_odd` — numero di valori dispari;
- `*fp` — il puntatore al file `file.txt`;
- `is_end` — una variabile che indica la fine dell'esecuzione.

Per ognuna di queste variabili (ad eccezione di `is_end` che non ha problemi di mutua esclusione) è stato creato un semaforo ad-hoc (sempre all'interno della struttura condivisa) che garantisce l'esecuzione atomica dell'operazione.

2.1.1 Inizializzazione

Si è cominciato quindi con l'inizializzazione delle variabili condivise.

```
void __init__(char *filename)
{
    pthread_mutexattr_t m_attr;
    pthread_mutexattr_init(&m_attr);
    pthread_mutex_init(&m.read_mutex, &m_attr);
    pthread_mutex_init(&m.sum_mutex, &m_attr);
    pthread_mutex_init(&m.odd_mutex, &m_attr);
    pthread_mutex_init(&m.min_mutex, &m_attr);
    pthread_mutex_init(&m.max_mutex, &m_attr);

    m.sum = m.n_odd = m.is_end = 0;
    m.min = INT_MAX;
    m.max = INT_MIN;

    m.fp = fopen(filename, "r");

    if (!m.fp)
    {
        printf("ERROR: could not open %s\n", filename);
        exit(EXIT_FAILURE);
    }
}
```

2.1.2 Funzioni di aggregazione

Ogni funzione implementa al suo interno le sezioni critiche, in modo da non delegare al thread la mutua esclusione, alleggerire il codice dal punto di vista visivo ed essere scalabile, nel senso che se in futuro si volesse aggiungere un altro thread con uno scopo diverso ma che utilizza al suo interno una (o più) delle seguenti funzioni, non si debba preoccupare della mutua esclusione, prevenendo così anche la ripetzione di codice.

```
void sum(int n)
{
    pthread_mutex_lock(&m.sum_mutex);
    m.sum += n;
    pthread_mutex_unlock(&m.sum_mutex);
}

void min(int n)
{
    pthread_mutex_lock(&m.min_mutex);
    if (n < m.min)
    {
        m.min = n;
    }
    pthread_mutex_unlock(&m.min_mutex);
}

void odd(int n)
{
    if ((n & 1) == 1)
    {
        pthread_mutex_lock(&m.odd_mutex);
        m.n_odd++;
        pthread_mutex_unlock(&m.odd_mutex);
    }
}

void max(int n)
{
    pthread_mutex_lock(&m.max_mutex);
    if (n > m.max)
    {
        m.max = n;
    }
    pthread_mutex_unlock(&m.max_mutex);
}
```

2.1.3 Lettura del file

Per la gestione del file si possono prendere due strade separate:

1. leggere riga per riga durante l'esecuzione del programma;
2. leggere il file nella sua totalità prima di avviare i thread e immagazzinare i caratteri in una matrice condivisa tra i thread.

Si è deciso di abbracciare l'approccio numero 1, sia per motivi di memoria, perchè il `file.txt` contenente i task potrebbe essere composto da numerose righe che potrebbero avere un impatto sulla memoria dell'ordine delle centinaia di megabyte da memorizzare in una struttura; sia per motivi di ottimizzazione, dato che una lettura totale iniziale avrebbe causato un rallentamento non prevedibile (dato dalle dimensioni del file) prima dell'esecuzione dei thread.

Facendo così, il rallentamento della lettura da file viene quindi equamente distribuito sui vari thread che faranno la richiesta di lettura del file.

```
int read_char(int arr[])
{
    // Optimization.
    if (m.is_end)
        return -1;

    char action;
    int num;

    pthread_mutex_lock(&m.read_mutex);
    int res = fscanf(m.fp, "%c   %d\n", &action,
                                              s&num);
    pthread_mutex_unlock(&m.read_mutex);

    if (res != 2)
    {
        m.is_end = 1;
        return -1;
    }

    arr[1] = num;

    if (action == 'p')
    {
        arr[0] = PROCESS;
    }
    else if (action == 'w')
    {
        arr[0] = WAIT;
    }
    else
    {
        printf("ERROR: Unrecognized action: '%c'\n",
                                              action);
        exit(EXIT_FAILURE);
    }
    return 0;
}
```

2.1.4 Thread lavoratore

Il thread invoca la funzione `read_char`, che con un puntatore passato come parametro e riempie quell'array con i due carateri letti dal file (codice e valore). Dopodichè il thread esamina la tipologia del lavoro ed esegue le quattro funzioni di aggregazione oppure "dorme", senza aprire nessuna sezione critica, perchè ogni funzione ha la propria.

```
void *thread(void *arg)
{
    int arr[2];

    while (read_char(arr) != -1)
    {
        int n = arr[1];
        if (arr[0] == PROCESS)
        {
            sum(n);
            odd(n);
            min(n);
            max(n);
        }
        else
        {
            // Seconds / 10.
            // For test-only purpose.
            msleep(n * 100);
        }
    }

    pthread_exit(NULL);
}
```

2.1.5 Thread principale

Il compito del thread Main è dare luce agli altri thread, inizializzando un array con il numero di thread da creare `N_THREAD`, crearli e aspettare che ognuno di essi finisca il proprio lavoro. Dopodichè stampare a video il tempo di esecuzione calcolato e i diversi risultati ottenuti. Il thread Main, è il thread master, che una volta creati i thread lavoratori, eseguirà anch'esso lo stesso codice dei lavoratori.

```
int main(int argc, char *argv[])
{
    ...

    // Create workers
    for (int i = 0; i < N_THREAD; i++)
        pthread_create(&p[i], &a, thread, NULL);

    // Execute master thread
    thread(NULL);

    // Wait for all
    for (int i = 0; i < N_THREAD; i++)
        pthread_join(p[i], NULL);

    ...
}
```

2.2 Esercizio B

A differenza dell'esercizio A che si doveva solo lavorare su diverse variabili condivise, in questo esercizio è anche importante sincronizzare tra di loro i vari thread. Per fare ciò ci si avvale dell'utilizzo dei semafori privati. Risultano molto effcienti per risolvere questo tipo di problemi dato che un'azione di un thread permette ad un altro thread di continuare la sua esecuzione.

Si comincia con l'inizializzazione delle variabili condivise e dei semafori.

```
void __init__()
{
    pthread_mutexattr_t m_attr;
    pthread_condattr_t c_attr;

    pthread_mutexattr_init(&m_attr);
    pthread_condattr_init(&c_attr);

    pthread_mutex_init(&m.mutex, &m_attr);

    pthread_cond_init(&m.priv_enter_immigrant,
                                        &c_attr);
    pthread_cond_init(&m.priv_leave_immigrant,
                                        &c_attr);
    pthread_cond_init(&m.priv_swear_immigrant,
                                        &c_attr);
    pthread_cond_init(
        &m.priv_get_certificate_immigrant,&c_attr);
    pthread_cond_init(&m.priv_judge_enter, &c_attr);
    pthread_cond_init(&m.priv_judge_check, &c_attr);
    pthread_cond_init(&m.priv_spectator, &c_attr);

    m.immigrants_checked_in = m.immigrants_in =
    m.blocked_immigrants = m.blocked_judge =
    m.spectators_in = m.is_judge_in =
    m.to_leave_immigrants = m.total_certs = 0;
}
```

2.2.1 Immigrati

Gli immigrati hanno 4 vincoli particolari:

- non possono entrare fintanto che il giudice è nell'edificio;
- non possono avere il certificato fintanto che il/la giudice non glielo conferma;
- prima che il/la giudice possa rientrare, tutti quelli che non hanno ricevuto la conferma devono uscire dall'edificio e ripresentarsi come fossero nuovi, e ripetere gli step precedenti alla conferma;
- non possono uscire fintanto che il/la giudice è nell'edificio.

2.2.1.1 Ingresso

Ogni volta che un immigrato entra nell'edificio si aumenta il contatore degli immigrati nell'edificio (`immigrants_in`).

```
// Block immigrant semaphore
void enter_immigrant(int ID)
{
    pthread_mutex_lock(&m.mutex);

    // Immigrant cannot enter until judge is in
    while (m.is_judge_in)
    {
        m.blocked_immigrants++;
        pthread_cond_wait(&m.priv_enter_immigrant,
                                        &m.mutex);
        m.blocked_immigrants--;
    }

    printf("%d) Enter Immigrant...\n", ID);

    m.immigrants_in++;

    pthread_mutex_unlock(&m.mutex);
}
```

2.2.1.2 Check-in

La normativa recita: *'Una volta che tutti gli immigrati hanno effettuato il*

check-in, il/la giudice può confermare la richiesta di cittadinanza per naturalizzazione'; il/la giudice deve quindi aspettare che tutti gli immigrati entrati nell'edificio prima di lui effettuino il check-in, prima che possa cominciare il suo lavoro di conferma della cittadinanza.

Quando gli immigrati effettuano il check-in viene incrementata la variabile `immigrants_checked_in`. Quando questa variabile raggiunge lo stesso valore di `immigrants_in` vuol dire che tutti gli immigrati hanno effettuato il check-in e il/la giudice può quindi cominicare a lavorare.

```
// Immigrant check-in

void check_in(int ID)
{
    pthread_mutex_lock(&m.mutex);
    printf("%d) Init check in...\n", ID);

    msleep(DELAY_OPERATIONS_MSEC);

    printf("%d) Finit check in...\n", ID);

    m.immigrants_checked_in++;

    // If the judge is in the building,
    // and all immigrants have done the check-in
    // the judge can start his job,
    // otherwise no.
    if (m.immigrants_checked_in == m.immigrants_in
                                && m.is_judge_in)
        pthread_cond_signal(&m.priv_judge_check);

    pthread_mutex_unlock(&m.mutex);
}
```

2.2.1.3 Riposo e giuramento

Una volta che un immigrato ha effettuato il check-in, si siede e aspetta il suo turno prima di effettuare il giuramento davanti al/la giudice.

```
// Immigrant sit down
void sit_down(int ID)
{
    printf("%d) Init sit down...\n", ID);

    msleep(NULL);

    printf("%d) Finit sit down...\n", ID);
}

// Immigrant swear
void swear(int ID)
{
    pthread_mutex_lock(&m.mutex);

    while (!m.is_judge_in)
    {
        printf("waiting for judge enter...\n");
        pthread_cond_wait(&m.priv_swear_immigrant,
                                        &m.mutex);
    }

    printf("%d) Init swear...\n", ID);
    msleep(NULL);
    printf("%d) Finit swear...\n", ID);

    pthread_mutex_unlock(&m.mutex);
}
```

2.2.1.4 Ritiro certificato

Una volta che l'immigrato ha finito il giuramento ed il/la giudice ha confermato la sua cittadinanza, può ritirare il certificato e lasciare l'edificio. Può capitare però che il/la giudice lasci l'aula improvvisamente. Tutti coloro che al momento dell'uscita del/la giudice non hanno ricevuto il certificato devono lasciare l'edificio e ripresentarsi, prima che il/la giudice possa entrare nuovamente. Entra quindi in gioco la variabile `to_leave_migrants` che tiene il conteggio delle persone che devono lasciare l'edificio prima che rientri il/la giudice.

Il certificato risulta:

- valido: se il certificato lo consegna il/la giudice, quindi il/la giudice è ancora dentro l'edificio;
- invalido: se il certificato viene consegnato quando il/la giudice non è presente in aula.

Si è pensato a questo metodo per non introdurre ulteriori variabili e semafori bloccanti.

```
int get_certificate(int ID)
{
    pthread_mutex_lock(&m.mutex);

    pthread_cond_wait(
        &m.priv_get_certificate_immigrant, &m.mutex);

    if (m.is_judge_in)
    {
        printf("%d) Got CERTIFICATE!\n", ID);
        m.total_certs++;
        pthread_mutex_unlock(&m.mutex);
        return 1;
    }
    else
    {
        pthread_mutex_unlock(&m.mutex);
        return 0;
    }
}
```

2.2.1.5 Uscita

Una volta ritirato il certificato valido, il non-più-immigrato può uscire dall'edificio e camminare per la città di Boston da cittadino americano.

```
// Release immigrant semaphore
void leave_immigrant(int ID)
{
    pthread_mutex_lock(&m.mutex);

    m.immigrants_in--;
    m.immigrants_checked_in--;

    // 'to_leave_migrants' are them that have been
    //  completed all steps but didn't get the
    // confirmation by the judge, so they have to
    // leave and enter again the building.
    //
    // It's a significant variable, because judge
    // cannot re-enter until all immigrants leave
    // the building.
    if (m.to_leave_immigrants)
    {
        // The last, free the blocked judge
        // (if is blocked...)
        if (m.to_leave_immigrants == 1)
            pthread_cond_signal(&m.priv_judge_enter);

        m.to_leave_immigrants--;
    }

    printf("%d) Leave Immigrant... immigrants_in:
            %d to_leave: %d\n", ID, m.immigrants_in,
            m.to_leave_immigrants);

    // If others immigrants are blocked, free them
    if (m.blocked_immigrants)
        pthread_cond_signal(&m.priv_enter_immigrant);

    pthread_mutex_unlock(&m.mutex);
}
```

2.2.1.6 Thread immigrato

Il thread dell'immigrato a questo punto risulta molto semplice, basta sola-

mente mettere in ordine i passi da compiere. Il tutto dentro ad un `while` che continua fintanto che non ha ricevuto il certificato.

```
void *immigrant(void *arg)
{
    int ID = (int *)arg;
    int cert_ok = 0;

    while (!cert_ok)
    {
        msleep(NULL);
        enter_immigrant(ID);

        check_in(ID);
        sit_down(ID);
        swear(ID);
        cert_ok = get_certificate(ID);

        leave_immigrant(ID);
    }
}
```

2.2.2 Giudice

Il/la giudice, come già anticipato negli scorsi paragrafi deve:

- entrare nell'edificio;
- aspettare che tutti gli immigrati abbaino effettuato il check-in;
- confermare la cittadinanza agli immigrati.

Non c'è un numero minimo o massimo di persone a cui può rilasciare la cittadinanza, ma il/la giudice quando vuole può lasciare l'edificio.

2.2.2.1 Ingresso

Quando il/la giudice entra si setta la variabile `is_judge_in` a 1. Aspetta ad entrare fintanto che ci sono degli immigrati che non hanno fatto in tempo a farsi confermare la cittadinanza dal turno precedente.

```
// Block judge semaphore
void enter_judge()
{
    pthread_mutex_lock(&m.mutex);

    while (m.to_leave_immigrants)
    {
        pthread_cond_wait(&m.priv_judge_enter,
                                &m.mutex);
    }

    printf("ENTER JUDGE!\n");
    m.is_judge_in = 1;

    pthread_cond_broadcast(&m.priv_swear_immigrant);

    pthread_mutex_unlock(&m.mutex);
}
```

2.2.2.2 Conferma

Il/la giudice comincia a confermare un immigrato alla volta, solamente quando tutti loro hanno effettuato il check-in.

```
void confirm()
{
    pthread_mutex_lock(&m.mutex);
    printf("Confirm...\n");

    while(m.immigrants_in != m.immigrants_checked_in)
    {
        pthread_cond_wait(&m.priv_judge_check,
                                    &m.mutex);
    }

    pthread_cond_signal(
            &m.priv_get_certificate_immigrant);

    pthread_mutex_unlock(&m.mutex);
}
```

2.2.2.3 Uscita

Quando il/la giudice decide di uscire, si alza e lascia l'edificio. Nell'uscire da la possibilità a tutti coloro che stavano aspettando fuori, di poter ora entrare dentro. Inoltre obbliga in modo indiretto gli immigrati che non hanno ricevuto il certificato a uscire e rientrare, perchè fintanto che rimangono dentro il/la giudice non può rientrare. La variabile `to_leave_migrants` prende il valore degli immigrati all'interno dell'edificio nell'istante in cui esce.

```
// Release judge semaphore
void leave_judge()
{
    pthread_mutex_lock(&m.mutex);

    m.is_judge_in = 0;

    printf("LEAVE JUDGE!\n");

    pthread_cond_broadcast(
                &m.priv_get_certificate_immigrant);
    pthread_cond_broadcast(&m.priv_enter_immigrant);
    pthread_cond_broadcast(&m.priv_spectator);

    m.to_leave_immigrants = m.immigrants_in;

    pthread_mutex_unlock(&m.mutex);
}
```

2.2.2.4 Thread giudice

Il ragionamento per il thread giudice è simile a quello dell'immigrato: si ripetono in serie le azioni da eseguire fintanto che gli immigrati da certificare sono finiti.

```
void *judge()
{
    int i;

    while (m.total_certs != N_IMMIGRANTS)
    {
        msleep(5000);
        enter_judge();
        msleep(5000);

        for (i = 0; i < m.immigrants_in; i++)
        {
            confirm();
        }

        msleep(5000);

        leave_judge();
    }
}
```

2.2.3 Spettatori

Gli spettatori non hanno particolari vincoli e sono pressochè liberi di fare quello che vogliono. L'unica limitazione è che quando il giudice è dentro nell'edificio, nessuno può entrare.

2.2.3.1 Ingresso

Come anticipato, gli spettatori non possono entrare se il giudice è nell'edificio.

```
// Block spectator semaphore
void enter_spectator()
{
    pthread_mutex_lock(&m.mutex);

    // Spectator cannot enter until judge is in
    while (m.is_judge_in)
    {
        pthread_cond_wait(&m.priv_spectator,
                                        &m.mutex);
    }

    m.spectators_in++;

    pthread_mutex_unlock(&m.mutex);
}
```

2.2.3.2 Partecipazione

Gli spettatori partecipano per un periodo di tempo casuale e poi lasciano l'edificio.

```
void spectate(int ID)
{
    printf("%d) Spectating...\n", ID);
    msleep(NULL);
}
```

2.2.3.3 Uscita

Gli spettatori escono semplicemente decrementando la loro corrispettiva variabile.

```
// Release spectator semaphore
void leave_spectator()
{
    pthread_mutex_lock(&m.mutex);

    m.spectators_in--;

    pthread_mutex_unlock(&m.mutex);
}
```

2.2.3.4 Thread spettatore

Come i due thread precedentemente menzionati esegue in serie i suoi comandi: lo spettatore entra, partecipa alla seduta, e poi lascia l'aula.

```
void *spectator(void *arg)
{
    int ID = (int *)arg;

    enter_spectator();
    spectate(ID);
    leave_spectator();
}
```

3 Test

Per testare il funzionamento dei programmi è necessario posizionarsi nella cartella del relativo esercizio ed eseguire il comando make.

3.1 Esercizio A

L'esercizio A richiede 2 parametri di input:

- numero di thread
- file dei task

```
./es 20 file.txt
```

3.2 Esercizio B

L'esercizio B richiede anch'esso 2 parametri di input:

- numero di thread immigrati
- numero di thread spettatori

```
./es 40 30
```

4 Conclusioni

Nel codice, in punti precisi sono stati aggiunti degli `msleep` che rappresentatno delle pause (casuali e non) per rendere l'esecuzione leggibile e più reale. I due esercizi sono stati testati diverse volte su un Macbook Pro M1 pro con architettura arm64, e hanno sempre prodotto i risulati voluti.

Bibliografia

[1] *didattica.agentgroup.unimo.it*

[2] *stackoverflow.com*

[3] *cplusplus.com*