# Sistemi Operativi

Esercizi Svolti di Sistemi Operativi

*Anno Accademico: 2024/25*

*Giacomo Sturm*

*Dipartimento di Ingegneria Civile, Informatica e delle Tecnologie Aeronautiche*
*Università degli Studi "Roma Tre"*

Sorgente del file LaTeXdisponibile al seguente link:
https://github.com/00Darxk/Sistemi-Operativi

# Indice

**1 Traduzione di Indirizzi** **1**

**2 Programmazione Multiprocesso** **3**
2.1 Esercizio 1 . . . 3
2.2 Esercizio 2 . . . 3
2.3 Esercizio 3 . . . 4
2.4 Esercizio 5 . . . 4
2.5 Esercizio 6 . . . 5
2.6 Esercizio 7 . . . 5

**3 Gestione della Memoria** **7**
3.1 Allocazione di Memoria Dinamica . . . 7
3.2 Gestione di Vettori Dinamici . . . 8
3.3 Manipolazione della memoria . . . 10
3.4 Implementazione di `my_alloc()` e `my_free()` . . . 11

**4 Programmazione Concorrente** **17**
4.1 Introduzione: Moltiplicazione tra Matrici . . . 17
4.2 Produttori e Consumatori . . . 20
4.3 Produttori e Consumatori su File . . . 24
4.4 Simulazione di uno Scheduler . . . 28
4.5 Problema dei Filosofi a Cena . . . 31
4.6 Problema del Barbiere Sonnolento . . . 33

# 1 Traduzione di Indirizzi

Questi esercizi sono stati generati con strumenti presenti nella repository `ostep-homework`, utilizzata anche per realizzare gli esercizi presenti nei quiz del corso.

Dato un sistema con uno spazio fisico di 32K ed uno spazio degli indirizzi di 16K e pagine di dimensione 4K. Con PTE eventi il bit più significativo, il più a sinistra, è il bit di validità, se è 1 la traduzione è valida, altrimenti se è 0 non è valida:

- `0`: `0x8000000c`;
- `1`: `0x00000000`;
- `2`: `0x00000000`;
- `3`: `0x80000006`.

Determinare l'indirizzo fisico dei seguenti indirizzi virtuali oppure specificare se sono invalidi:

- `0x00003229` (`12841`);
- `0x00001369` (`4969`);
- `0x00001e80` (`7808`);
- `0x00002556` (`9558`);
- `0x00003a1e` (`14878`).

Per semplificare la procedura per determinare il valore dei bit più significativi, si vuole sapere in quale pagina è presente l'indirizzo, quindi è sufficiente determinare il quoziente tra l'indirizzo virtuale e la dimensione della pagina:

$$\text{VPN} : \left\lfloor \frac{12841}{4098} \right\rfloor = 3$$

L'offset si calcola come la posizione nella pagina di questo indirizzo calcolata come:

$$\text{Offset} : 12831 - 3 * 4096 = 553$$

Il PFN corrispondente è `80000006`, è valido avendo il bit più significativo pari ad uno, ed ha un indirizzo fisico 6. Per cui l'indirizzo fisico corrispondente è:

$$6 * 4096 + 553 = 25129 \tag{1.0.1}$$

Il secondo indirizzo 4969 si riferisce al secondo PFN, dato che $\lfloor 4969/4096 \rfloor = 1$. Ma questa pagina ha il bit più significativo pari a zero, quindi la traduzione non è valida. Allo stesso modo per l'indirizzo 7808. L'indirizzo 9558 si riferisce alla seconda terza pagina, anch'essa si riferisce ad un indirizzo fisico con il bit più significativo pari a zero, quindi neanche questa traduzione è valida.

L'ultimo indirizzo appartiene alla quarta VPN:

$$\begin{aligned} \text{VPN} : \left\lfloor \frac{14878}{4096} \right\rfloor &= 3 \\ 14878 - 4096 * 3 &= 2590 \\ 6 * 4096 + 2590 &= 27166 \end{aligned} \tag{1.0.2}$$

# 2 Programmazione Multiprocesso[1]

Esercizi alle pagine 14 e 15 del capitolo 5, esclusa la seconda parte del 3, il numero 4 ed il numero 8. I codici sono disponibili presso il seguente link Esercitazione del 29 Ottobre 2024.

## 2.1 Esercizio 1

Write a program that calls `fork()`. Before calling `fork()`, have the main process access a variable, and set its value to something. What value is the variabile in the child process? What happens to the variabile when both the child and parent change the value of $x$?

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


int main() {
        int x = 0;
        x = 100;
        int rc = fork();
        x++;
        printf("Viene incrementato di uno: %d\n", x);
}
```

La variabile nel processo genitore e figlio ha lo stesso valore e vengono aggiornate indipendentemente tra di loro, allo stesso modo, poiché i due processi hanno spazi degli indirizzi diversi e nel codice non c'è distinzione tra processo figlio o genitore.

## 2.2 Esercizio 2

Write a program that opens a file (with the open() system call) and then calls `fork()` to create a new process. Can both the child and parent access the file descriptor returned by `open()`? What happens when they are writing to the file concurrently, i.e., at the same time?

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>


int main(){
        int fd = open("domanda2.txt", O_CREAT | O_TRUNC | O_WRONLY, S_IRWXU);
        int rc = fork();
        if(rc == 0){ // child
```

[1] *Dall'esercitazione in laboratorio del 29 Ottobre 2024*

```
                char* txt = "Ciao, sono il figlio\n";
                write(fd, txt, strlen(txt));

        } else { // parent
                char* txt = "Ciao, sono il padre\n";
                write(fd, txt, strlen(txt));
        }
        close(fd);
}
```

Entrambi i processi possono accedere al file `domanda2.txt` mediante lo stesso file descriptor, ed in base alle flag inserite nella funzione `open()` possono sovrascrivere il contenuto con `O_TRUNC`, oppure inserirlo in coda con `O_APPEND`.

Per distinguere tra il processo padre ed il processo figlio si utilizza il valore di ritorno di `fork()` salvato nella variabile `rc`, 0 se nel processo figlio, ed il PID del processo figlio nel genitore.

## 2.3 Esercizio 3

Write another program using `fork()`. The child process should print "hello"; the parent process should print "goodbye".

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


int main(){
        int rc = fork();
        if(rc == 0){ // child
                printf("Goodbye\n");
        } else { // parent
                pritnf("Hello\n");
        }
}
```

## 2.4 Esercizio 5

Now write a program that uses `wait()` to wait for the child process to finish in the parent. What does `wait()` return? What happens if you use `wait()` in the child?

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
```

```c
int main(){
        int rc = fork();
        if(rc == 0){ // child
                printf("Hello\n");
        } else {
                wait(NULL);
                printf("Goodbye\n");
        }
}
```

La chiamata di sistema `wait()` restituisce un intero corrispondente al PID del figlio che ha aspettato. Prende come parametro un puntatore ad uno stato del processo da aspettare, di default, quindi con `NULL` aspetta la sua terminazione. Se viene chiamato su un processo che non presenta figli, come il processo figlio in questo caso, termina la chiamata termina istantaneamente.

## 2.5 Esercizio 6

Write a slight modification of the previous program, this time using `waitpid()` instead of `wait()`. When would `waitpid()` be useful?

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>


int main(){
        int rc = fork();
        if(rc == 0){ // child
                printf("Hello\n");

        } else {
                waitpid(rc, NULL, 0);
                printf("Goodbye\n");
        }
}
```

La funzione `waitpid()` sarebbe più utile in caso fosse presente più di un processo figlio e si volesse attendere la terminazione di uno di questi in particolare. Come la `wait()` il secondo argomento è un puntatore ad un intero dove inserirà lo stato del processo figlio terminato, il primo argomento è un intero e rappresenta il PID del processo da aspettare, e l'ultimo argomento rappresenta opzioni che attraverso delle flag che possono essere unite mettendole in or tra di loro.

## 2.6 Esercizio 7

Write a program that creates a child process, and then in the child closes standard output `STDOUT FILENO`. What happens if the child calls `printf()` to print some output after closing the descriptor?

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>


int main(){
        int rc = fork();
        if(rc == 0){
                close(STDOUT_FILENO);
                printf("Hello\n");
        }
}
```

Il processo figlio avendo chiuso lo standard output non è più in grado di stampare a schermo.

# 3 Gestione della Memoria[2]

I codici sono disponibili presso il seguente link Esercitazione del 19 Novembre 2024.

## 3.1 Allocazione di Memoria Dinamica

Scrivi un programma che crea un array dinamico di numeri interi di dimensione specificata dall'utente, riempie l'array con numeri casuali e stampa l'array. Assicurati di liberare la memoria allocata alla fine del programma.

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


int main(){
        srand(time(NULL));
        int x = 0;
        printf("Specificare la dimensione: \n");
        scanf("%d%*c", &x);

        int* array = malloc(x*sizeof(int));
        for (int i = 0; i < x; i++)
                array[i] = rand()%x;

    printArray(array, x);
        free(array);
}
```

Il programma richiedere un input dall'utente, e crea un array di dimensione pari a questo valore, riempito con interi di valore massimo pari a questo valore. Viene allocata memoria con `malloc()` e liberata con `free()`.

La funzione `printArray()` è utilizzata per verificare queste aggiunte, e verrà usata nei successivi esercizi:

```c
int printArray(int *array, int x){
        for (int i = 0; i < x; i++){
                if(i == 0) printf("[%d, ", array[i]);
                else if (i == x - 1) printf("%d]\n", array[i]);
                else printf("%d, ", array[i]);
        }
}
```

[2] *Dall'esercitazione in laboratorio del 19 Novembre 2024*

## 3.2 Gestione di Vettori Dinamici

Implementa una struttura dati per un vettore dinamico di interi. Includi funzioni per aggiungere un elemento, rimuovere un elemento e stampare tutti gli elementi del vettore. Assicurati che il vettore aumenti la sua capacità quando necessario.

Questo si realizza introducendo una struttura dati contenente un array `v`, la dimensione attuale dell'array `l`, e la dimensione massima per la memoria allocata di `v`, `lmax`:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define DEFAULT_DIM 4


typedef struct Vect{
        int* v;
        int l;
        int lmax;
}Vect;
```

Si definiscono quindi le funzioni di creazione del vettore, aggiunta di un elemento e rimozione di un elemento, dato un indice.

```
Vect* newVect(int lmax){
        int* newArray = malloc(lmax * sizeof(int));
        Vect* newVect = malloc(sizeof(Vect));
        newVect->v = newArray;
        newVect->l = 0;
        newVect->lmax = lmax;

        return newVect;
}
```

Per l'aggiunta se viene raggiunta la dimensione massima dell'array, allora viene re-allocato su una zona di memoria di dimensione doppia, avendo un inserimento ammortizzato a tempo costante. Si utilizza quindi la funzione `realloc()`:

```
void addVect(Vect* vect, int x){
        if(vect->l >= vect->lmax){
                vect->v = realloc(vect->v, vect->lmax * 2);
                vect->lmax *= 2;
        }
        vect->v[vect->l] = x;
        vect->l += 1;
}
```

Per la rimozione si effettua prima un controllo sull'indice e si effettua uno shift di una posizione verso sinistra di ogni elemento successivo all'indice dato, e si decrementa la lunghezza dell'array `l`:

```
void removeVect(Vect* vect, int i){
        if(vect->l <= i) printf("\nIndex Overflow\n");
        else if(i < 0)   printf("\nIndex Underflow\n");
        else{
        for (int j = i; j < vect->l-1; j++)
            vect->v[j] = vect->v[j + 1];
        vect->l--;
    }
}
```

Nella funzione main si controlla il funzionamento generando un vettore di dimensione di default ed aggiungendo un numero di elementi casuali determinati dall'utente. In seguito si rimuove un elemento di indice selezionato.

```
int main(){
        srand(time(NULL));
        Vect* v = newVect(DEFAULT_DIM);
        int dim = 0;
        printf("Specifica la dimensione dell'array: \n");
        scanf("%d%*c", &dim);
        for (int i = 0; i < dim; i++)
                addVect(v, rand()%dim);
        printVect(v);
        int del = -1;
        printf("Specificare l'indice dell'elemento da rimuovere: \n");
        scanf("%d%*c", &del);
        removeVect(v, del);
        printVect(v);
        free(v);
}
```

Si utilizza la funzione `printArray()` precedente per stampare il vettore:

```
int printArray(int *array, int x){
        for (int i = 0; i < x; i++){
                if(i == 0) printf("[%d, ", array[i]);
                else if (i == x - 1) printf("%d]\n", array[i]);
                else printf("%d, ", array[i]);
        }
}
```

## 3.3 Manipolazione della memoria

Scrivi un programma che alloca memoria per un array di interi, usa `memset()` per impostare tutti gli elementi a zero, quindi usa `memcpy()` per copiare il contenuto di un altro array nella memoria appena allocata.

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


int main(){
        int dim = 0;
    printf("Specifica la dimensione dell'array: \n");
        scanf("%d%*c", &dim);
        int* array = malloc(dim * sizeof(int));
        array = memset(array, 0, dim * sizeof(int));
        printArray(array, dim);
        memcpy(array, randArray(dim), dim * sizeof(int));
        printArray(array, dim);
}
```

La funzione `memset()` prende come argomento un puntatore alla zona di memoria da modificare, il valore da assegnare ad ogni elemento, e la lunghezza per cui deve effettuare questo procedimento. Quindi potrebbe assegnare `0` solamente ad una porzione dell'array se venisse specificato. La funzione `memcpy()` copia nella zona di memoria passata come primo argomento la zona di memoria passata come secondo argomento, quindi è necessario che la funzione `randArray()` restituisce un nuovo array. Inoltre come ultimo argomento prende una dimensione specificata per cui effettuare questa copia. Quindi anche per questa funzione è possibile copiare parzialmente una zona di memoria in un'altra.

La funzione `randArray()` restituisce un array contenente interi casuali, al massimo fino a `dim`, di dimensione specificata.

```c
int* randArray(int dim){
        srand(time(NULL));
        int* array = malloc(dim * sizeof(int));
        for(int i = 0; i < dim; i++)
                array[i] = random()%dim;
        return array;
}
```

## 3.4 Implementazione di `my_alloc()` e `my_free()`[3]

In questo esercizio bisogna implementare il comportamento delle funzioni `malloc()` e `free()`, senza utilizzare queste due funzioni. Bisogna creare due funzioni una che dato un intero che indica il numero di byte da allocare, restituisce un puntatore di tipo `void*` all'inizio della zona di memoria di dimensione passata. Ed un'altra funzione che prende un puntatore di tipo `void*` ad una zona di memoria allocata e la libera.

Quindi l'argomento della funzione `my_alloc()` è il numero di byte della zona di memoria, di tipo `size_t`, e restituisce un puntatore generico poiché non conosce il tipo di dati che sarà contenuto in questa zona di memoria. La funzione `my_free()` dovrebbe prendere un puntatore ad un'area di memoria generica e restituisce un valore che indica se questa operazione è andata a buon fine o meno.

Per implementare queste due funzioni si possono utilizzare una singola volta le funzioni `mmap()` e `munmap()`, durante l'esecuzione del programma. La prima funzione può essere utilizzata in due modi diversi, simulando il comportamento della malloc, poiché prende un numero di argomenti maggiore, permettendo comportamenti dati di default dalla malloc. Si possono definire i permessi dell'area di memoria da allocare, se è di sola lettura o scrittura, specificare se alcuni byte di quest'area possono essere eseguiti o indicare se questa zona può essere condivisa. È una funzione di più basso livello, con pià casi d'uso rispetto alla malloc, e si utilizza invece della malloc quando sono necessarie esigenze particolari. Un altro caso permette di creare una mappa con nome, un riferimento in memoria rispetto ad una file che si trova nel filesystem. Permette di semplificare il processo di interazione con un file nel filesystem, riservando un'area di memoria di dimensione pari a quella di questo file, con un certo offset opzionale. Si accede a questo file con la notazione array sulla zona di memoria che mappa il file.

Se c'è una connessone diretta tra questa zona di memoria ed il file su disco, in base alle opzioni della funzione, è possibile modificare anche il file su disco. Questo tuttavia ha un costo sull'utilizzo della memoria, che potrebbe impedire il suo utilizzo, in caso la memoria virtuale necessaria è minore del file che si vuole mappare. Inoltre in base al file si potrebbe voler modificare o leggere una piccola porzione del file, quindi invece di effettuare una mappatura completa, è conveniente accedere normalmente al file senza trasferirlo in memoria. Si potrebbe creare mappe anonime, che non hanno un file corrispettivo nel filesystem, questo corrisponde all'esecuzione di una `malloc()`, poiché non ha di default un file. La `munmap()` è la controparte di questa funzione, permette di liberare un'area di memoria allocata utilizzando la funzione `mmap()`.

Accedendo alle pagine di manuale corrispondenti tramite il comando:

```
corso@sistemi-operativi:~$ man map
```

Queste funzioni sono incluse nell'header `sys/mman.h`. Il primo argomento della funzione è l'indirizzo di memoria che si vorrebbe avere come l'indirizzo di partenza della memoria allocata. Questa è un'opzione molto sofisticata, non necessaria per applicazioni così semplici, quindi si può inserire il valore `NULL` per specificare che l'indirizzo di partenza della memoria non è rilevante, quindi è il kernel a scegliere automaticamente l'indirizzo, in modo che sia allineato con una pagina, ovvero ha un offset nullo per una pagina di memoria. Questo rappresenta solo un suggerimento, la

[3] *Soluzione Offerta dal Professore in Aula*

funzione potrebbe non accettarlo e quindi dovrebbe sceglierlo il kernel nel modo precedentemente descritto.

Un altro parametro di tipo `site_t` specifica il numero di byte di questa zona di memoria da allocare. Altri due parametri chiamati `prot` e `flags` di tipo intero specificano opzioni di accesso e flag aggiuntive per l'operazione. Un altro parametro `fd`, chiamato come un file descriptor, sempre di tipo intero, rappresenta il file che si vuole inserire in questa zona di memoria. Infine si può inserire un offset, per portare in memoria solamente una porzione del file, di tipo `off_t`.

Per i parametri `prot` si possono utilizzare diverse opzioni per specificare che le pagine possono essere eseguite, lette, scritte, oppure non sono accessibili:

- `PROT_EXEC`: Permette la sua esecuzione;
- `PROT_READ`: Permette l'accesso in lettura;
- `PROT_WRITE`: Permette l'accesso in scrittura;
- `PROT_NONE`: Non permette nessuna della precedenti.

Queste operazioni possono essere messe in or per utilizzare più opzioni contemporaneamente. Per il parametro `flags` sono possibili molte opzioni, le più interessanti per questo caso sono le opzioni per condividere la mappa ad altri processi, oppure per renderla privata ad un singolo processo:

- `MAP_PUBLIC`: La mappatura è accessibile da altri processi;
- `MAP_PRIVATE`: La mappatura non è accessibile ad altri processi;
- `MAP_ANONYMOUS`: La mappatura non richiede un file.

Se si utilizza le opzioni `prot` per scrivere e leggere, e si indica con il parametro `flags` che si tratta di una mappa privata, il comportamento di questa funzione è analogo al comportamento di una `malloc()`. Si utilizza un'ulteriore flag per indicare che si vuole creare una mappa anonima, senza specificare il file descriptor per il file su cui si vuole operare. Con questa flag il parametro `fd` viene ignorato, e convenzionalmente viene inserito il valore -1. Queste funzioni appartengono allo standard POSIX, su molti sistemi operativi basati sul kernel Linux, e sono in accordo rispetto a delle interfacce e prototipi di funzioni, che possono avere implementazioni anche molto differenti l'una con l'altra. Quindi in alcune implementazioni, è richiesto che `fd` sia uguale a -1. Dato che l'offset non viene utilizzato si pone pari a zero. La chiamata alla funzione `mmap()` è quindi:

```
memory_pool = (Block*) mmap(NULL, MEMORY_POOL_SIZE, PROT_READ |
↪  PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
```

Per la `munmap()` si hanno due argomenti, l'indirizzo di base della mappa che si vuole rimuovere, e come secondo argomento si prende la dimensione della zona di memoria da liberare. La funzione `free()` non ha bisogno di questo parametro, ma identifica automaticamente la dimensione della zona di memoria. Non appare a questo livello l'informazione necessaria per liberare l'area di memoria. Ciò avviene poiché alla chiamata di una `malloc()`, poco prima dell'indirizzo restituito inserisce

dei metadati relativi alla memoria che ha allocato, come la sua dimensione in byte. Questa dimensione viene letta ad un'eventuale `free()` ed è quindi in grado di liberare esattamente la memoria precedentemente allocata.

Per utilizzare una singola volta nell'intero programma queste due funzioni, un possibile approccio consiste nel realizzare una prima chiamata alla `mmap()` generando una grande area di memoria, utilizzabile poi nel resto del programma. Inserendo direttamente la `mmap()` dentro la `my_alloc()` comporterebbe multiple chiamata alla `mmap()`, ogni volte che viene effettuata una nuova allocazione. Questo essenzialmente aggira il problema, come se fosse inserita una `malloc()` all'interno della funzione `my_alloc()`.

Si utilizza una funzione per allocare questa zona di memoria. Su questa zona per allocare una porzione di memoria, in modo che la `my_free()` sia in grado di liberarla, bisogna implementare un meccanismo analogo alla `free()`, inserendo un header per salvare metadati relativi alla memoria richiesta. Si realizza una struttura dati di tipo `block` contenente la dimensione del blocco di memoria che segue il metadato, una variabile che indica se il blocco è libero ed il suo successivo blocco di memoria:

```
typedef struct Block{
    size_t size;        // dimensione blocco
    int free;           // blocco libero (1) o occupato (0)
    struct Block* next; // blocco successivo
}Block;
```

Inizialmente questa zona di memoria rappresenta un intero blocco, contenente nei primi 16 byte un puntatore al relativo `block` contenente i metadati. Quando si effettua una chiamata alla `my_alloc()` di dimensione minore della memoria allocata, ma abbastanza grande da impedire la creazione di un altro blocco di memoria, poiché sono richiesti almeno 16 byte per memorizzare solamente i metadati in `block*`. L'indirizzo restituito dalla `my_alloc()` è la posizione iniziale sommata ai 16 byte dei metadati. La funzione `my_free()` prende un indirizzo base, torna indietro di 16 byte, e legge il blocco dei metadati, impostando ad uno il parametro `free` della struttura, in modo che sia di nuovo utilizzabile.

Se si effettua un'allocazione di dimensione più piccola, allora è possibile realizzare un nuovo blocco di questa memoria creando un nuovo `block`, contenuto subito la zona più piccola allocata. In questo modo si divide il blocco iniziale in due, dove il primo avrà come metadato `size` la dimensione attuale e come parametro `next` l'indirizzo del successivo blocco. Questo blocco successivo avrà come metadato `size` la dimensione rimanente della memoria allocata libera. Ma quando viene liberato il primo blocco di memoria, saranno presenti due blocchi di memoria entrambi liberi, che dividono lo spazio totale. Questo può impedire allocazioni più grandi della dimensione di questi blocchi, nonostante complessivamente lo spazio totale possa contenerle.

Questo rappresenta il memoria della frammentazione interna, per risolverlo al completamento di una `my_free()` bisogna provare ad effettuare un unione tra blocchi liberi adiacenti, controllando dal blocco successivo nel parametro `next` dei metadati, se anch'esso è libero. Continuando nei blocchi successivi fino a quando non si incontra un blocco occupato.

Nonostante questo è possibile una frammentazione dove due blocchi di memoria libera sono divisi da un blocco occupato.

Bisogna effettuare un'operazione di deframmentazione spostando i blocchi nella zona, ma in questo scenario non è possibile effettuarlo poiché non si sta programmando a livello del kernel, quindi l'effetto sarà non trasparente al livello del programmatore. Dopo aver spostato un blocco ed unito i due liberi, l'indirizzo di memoria del blocco occupato cambia, e non sarà più possibile accedervi tramite l'indirizzo fornito dalla precedente chiamata alla `my_alloc()`. All'interno del sistema operativo sarebbe sufficiente modificare le istanze della page table per rendere questo cambiamento invisibile al programmatore, mantenendo la funzionalità dell'indirizzo fornito dalla `my_alloc()`.

All'inizio del programma si definiscono due macro, una per la dimensione della memoria, e la seguente per la dimensione minima di un blocco:

```
#define MEMORY_POOL_SIZE (1024* 1024* 1024)
#define MIN_BLOCK_SIZE sizeof(Block)
```

Si inizializzano due variabili puntatore a blocco per lo spazio di memoria e per la lista dei blocchi liberi:

```
static Block* memory_pool = NULL;
static Block* free_list = NULL;
```

La prima funzione inizializza lo spazio di memoria, chiamando un'unica volta nell'intero programma la funzione `mmap()`, controllando se ha fallito l'esecuzione ed inizializzando i valori dei metadati che si sta creando.

```
void init_memory_pool(){
    if(memory_pool == NULL){
        memory_pool = (Block*) mmap(NULL, MEMORY_POOL_SIZE, PROT_READ |
        ↪  PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);

        if(memory_pool == MAP_FAILED){
            pererror("mmap");
            exit(1);
        }

        memory_pool->size = MEMORY_POOL_SIZE - sizeof(Block);
        memory_pool->free = 1;
        memory_pool->next = NULL;
        free_list = memory_pool;
    }
}
```

Si definisce in seguito l'operazione per suddividere dei blocchi, dato un riferimento ad un blocco `block` e la dimensione della memoria occupata nel blocco `size`, inserendo in coda il nuovo blocco. Il puntatore di base del nuovo blocco si ottiene dall'indirizzo del blocco, sommando la dimensione della memoria da occupare passata come parametro `size` e la dimensione del blocco. La dimensione

di questo nuovo blocco è la dimensione rimasta, dopo aver rimosso dalla dimensione corrente del blocco la dimensione dei dati che si sta inserendo `size` e la dimensione del blocco. Lo stato di questo nuovo blocco è libero ed ha come successivo, il blocco successivo del corrente. Si aggiornano in fine i metadati del blocco passato, inserendo come successivo questo nuovo blocco.

```
void split_block(Block* block, size_t size){
    Block* new_block = (Block*)((char*)block + sizeof(Block) + size);
    new_block->size = block->size - sizeof(Block) - size;
    new_block->free = 1;
    new_block->next = block->next;

    block->size = size;
    block->free = 0;
    block->next = new_block;
}
```

Un'ulteriore funzione stampa il layout della memoria in un dato momento, ogni volta che viene chiamata un'allocazione o de-allocazione.

Segue la definizione della `my_alloc()`, rifiuta richieste di allocazioni dove la dimensione passata è minore o uguale a zero, poiché si tratta di una richiesta sbagliata. In seguito chiama `init_memory_pool()`, per creare se non è mai stata creata la zona di memoria. In seguito scorre la `free_list` cercando un blocco libero, di dimensione maggiore della richiesta. Se viene trovato, si controlla se è possibile frammentarlo, ed in caso viene chiamata la `split_blocks()`, altrimenti semplicemente si assegna il blocco corrente come occupato. In seguito viene restituito l'indirizzo di partenza della memoria allocata, aggiungendo la dimensione del blocco all'indirizzo del blocco corrente. Se non viene individuato alcun blocco viene restituito `NULL`, poiché si è esaurita la memoria di dimensione maggiore della dimensione specificata.

```
void* my_alloc(size_t size){
    if(size <= 0)
        return NULL;
    init_memory_pool();

    Block* current = free_list;
    while(current != NULL){
        if(current->free && current->size >= size){
            if(current->size > size + MIN_BLOCK_SIZE)
                split_block(current, size);
            else
                current->free = 0;
            printf("Memoria allocata, nuovo layout:\n");
            print_blocks(free_list);
            return (void*)((char*)current + sizeof(Block));
        }
        current = curent->next;
```

```
    }
    return NULL;
}
```

In seguito si definisce la funzione per unire tra loro due blocchi, quando si trovano due nodi liberi adiacenti, scorrendo la lista.

```
void merge_blocks(Block* block){
    while(block->next != NULL && block->next->free){
        block->size += sizeof(Block) + block->next->size;
        block->next = block->next->next;
    }
}
```

Quest'implementazione permette l'unione di blocchi adiacenti, controllando solamente a destra, se fossero presenti dei blocchi liberi alla sinistra, non verrebbero uniti tra di loro.

Per realizzare la `my_free()` si passa un puntatore, e si ottiene l'indirizzo al blocco corrispondente, sottraendo la dimensione di un blocco. Si aggiornano i suoi metadati e si tenta di unire blocchi liberi, chiamando la `merge_blocks()` sul blocco corrente:

```
void my_free(void* ptr){
    if(ptr == NULL)
        return NULL;

    Block* block = (Block*)((char*)block - sizeof(Block));
    block->free = 1;
    merge_blocks(block);
    printf("Memoria de-allocata, nuovo layout:\n");
    print_blocks(free_list);
}
```

Si definisce infine la funzione per liberare la memoria al termine del programma:

```
void cleanup_memory_pool(){
    if(memory_pool != NULL){
        if(munmap(memory_pool, MEMORY_POOL_SIZE) == -1)
            pererror("munmap");
        memory_pool = NULL;
        free_list = NULL;
    }
}
```

# 4 Programmazione Concorrente

I codici sono disponibili presso il seguente link Programmazione Concorrente.

## 4.1 Introduzione: Moltiplicazione tra Matrici

Se un problema è parallelizzabile e si può implementare utilizzando thread con una qualsiasi libreria, non necessariamente `pthread`. Se è presente un'architettura multicore o multiprocessore, allora è molto probabile rendere più veloce l'esecuzione del programma, ottenendo prestazioni migliori.

Si considera un programma in grado di effettuare la moltiplicazione tra due matrici riempite di interi casuali. Si ha una macro che definisce una dimensione massima per la matrice bidimensionale e si definiscono tre matrici in variabili globali. Una prima funzione riempie le matrici in modo casuale:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/wait.h>

#define MATRIX_SIZE 2000
    int a[MATRIX_SIZE][MATRIX_SIZE];
    int b[MATRIX_SIZE][MATRIX_SIZE];
    int result[MATRIX_SIZE][MATRIX_SIZE];

void fill_matrix(int matrix[MATRIX_SIZE][MATRIX_SIZE]){
    for (int i = 0; i < MATRIX_SIZE; i++)
        for (int j = 0; j < MATRIX_SIZE; j++)
            matrix[i][j] = rand() % 100;
}
```

Un ulteriore funzione effettua questa moltiplicazione, in modo seriale:

```
void multiply_matrices_single_threaded( int a[MATRIX_SIZE][MATRIX_SIZE], int
↪  b[MATRIX_SIZE][MATRIX_SIZE]){
    for (int i = 0; i < MATRIX_SIZE; i++)
        for (int j = 0; j < MATRIX_SIZE; j++){
            result[i][j] = 0;
            for(int k = 0; k < MATRIX_SIZE; k++)
                result[i][j] += a[i][k] * b[k][j]
        }
}
```

Nella funzione main si inizializzano queste due matrici. Inoltre si utilizza una funzione per in-strumentare il codice ed ottenere statistiche, in questo caso riguardanti i tempi di esecuzione. Si può utilizzare la funzione `gettimeofday()`, più precisa rispetto alla funzione `time()`, che restituisce

il valore in una variabile di tipo `time_val` del tempo corrente. Questa struttura contiene un campo contenente il time stamp, ed un'altra porzione che contiene il valore in microsecondi al momento dell'esecuzione della funzione. `time()` invece ha intervalli di tempo in secondi, e non ha informazioni sul tempo in microsecondi.

Intorno alla funzione di cui vogliamo conoscere queste statistiche sono presenti due chiamata alla funzione descritta, salvando i loro valori in due variabili diverse. In seguito si effettua una somma sulla stessa scale per determinare il tempo effettivo dell'esecuzione:

```
int main(){
    a = fill_matrix();
    b = fill_matrix();

    struct time_val start, end;

    gettimeofday(&start, NULL);
    multiply_matrices_single_threaded(a, b);
    gettimeofday(&end, NULL);

    double time_taken = (end.tv_sec - start.tv_sec) * 1e6;
    time_taken = (time_taken + (end.tv_usec - start.tv_usec)) * 1e-6;

    printf("Time taken: (single-threaded) %f seconds\n", time_taken);

    return 0;
}
```

Per svolgere questo programma sfruttando la programmazione concorrente, si utilizzano un certo numero di thread assegnati ad un certo numero di righe della matrice. Si realizza una struttura dati per contenere le informazioni relative ad i thread, e si assegna un certo numero di thread:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <sys/wait.h>

#define MATRIX_SIZE 2000
#define NUM_THREADS 4
    int a[MATRIX_SIZE][MATRIX_SIZE];
    int b[MATRIX_SIZE][MATRIX_SIZE];
    int result[MATRIX_SIZE][MATRIX_SIZE];

typedef struct {
    int (*a)[MATRIX_SIZE];
    int (*b)[MATRIX_SIZE];
```

```
    int (*result)[MATRIX_SIZE];
    int start_row;
    int end_row;
}ThreadData;
```

Il calcolo del prodotto tra matrici avviene quindi sui sottoinsiemi del problema contenuti nella struttura di tipo `ThreadData`:

```
void* multiply_matrices_multi_threaded(void *arg){
    ThreadData *data = (ThreadData*) arg;
    for (int i = data->start_row; i < data->end_row; i++)
        for (int j = data->start_row; j < data->end_row; j++){
            data->result[i][j] = 0;
            for(int k = data->start_row; k < data->end_row; k++)
                data->result[i][j] += data->a[i][k] * data->b[k][j]
        }
    return NULL;
}
```

Il problema viene diviso su questi thread, associando ad ognuno di questi thread un sottoinsieme delle righe della matrice originaria:

```
    pthread_t threads[NUM_THREADS];
    ThreadData *thread_data[NUM_THREADS];
    int rows_per_thread = MATRIX_SIZE/NUM_THREADS;
```

L'unica altra modifica rispetto al programma seriale è la presenza di un ciclo tra le due `gettimeofday()`, dove vengono avviati tutti i thread, inizializzando le loro strutture `thread_data`, ed un altro ciclo per effettuare un join su ognuno di questi thread:

```
    for(int i = 0; i < NUM_THREADS; i++){
        thread_data[i]->a = a;
        thread_data[i]->b = b;
        thread_data[i]->result = result;
        thread_data[i]->start_row = i * rows_per_thread;
        thread_data[i]->end_row = (i + 1) * rows_per_thread;
        if (i == NUM_THREADS - 1)
            thread_data[i]->start_row += MATRIX_SIZE % NUM_THREADS;

        pthread_create(&threads[i], NULL, multiply_matrices_multi_threaded,
        ↪  thread_data[i]);
    }

    for(int i = 0; i < NUM_THREADS; i++)
        pthread_join(threads[i], NULL);
```

Ci si aspetta che il tempo di esecuzione di questo programma sia velocizzato di un fattore pari al numero di thread, ma essendo presenti overhead, ed in generale nell'esecuzione di un qualsiasi programma parallelizzabile, non è possibile raggiungere questo incremento ideale. In questo caso inoltre il programma viene eseguito in un container docker quindi ha uno stack molto profondo, è quindi difficile ottenere le prestazioni volute. Utilizzando il comando `top`, durante l'esecuzione di questa funzione mostra come le percentuali di utilizzo della CPU sono del 400%, poiché rappresenta la somma dell'utilizzo di ogni core del processore, avendo quattro thread è quindi quattro volte l'utilizzo del programma seriale.

Essendo eseguito all'interno di un container docker, il risultato di questa esecuzione dipende dalla configurazione corrente. In caso il numero di core disponibili è minore del numero di thread descritti in questo file è necessario modificare la configurazione di docker. Nella Home directory è presente un file chiamato `.wslconfig` che specifica le risorse da assegnare, contiene un nome che indica la configurazione tra parentesi quadre, e due parametri `memory` e `processors`, questi se sono commentati indicano il valore di default:

```
[wsl2]
#memory=4GB
processors=4
```

Se non è presente è possibile crearlo con il nome della configurazione `[wsl2]`. In versioni successive di docker è possibile modificare questi valori con un'interfaccia grafica tra le impostazioni di docker. Per aggiornare questi valori bisogna terminare il container ed eseguire il comando `wsl --shutdown` per riavviare l'engine wsl con le relative configurazioni. Il tempo di esecuzione diminuendo il numero di core disponibili è migliore rispetto al programma non parallelizzato, ma decisamente peggiore rispetto alla configurazione con più core. Se si modifica il codice del programma per rendere il numero di thread pari al numero di core disponibili allora è possibile aumentare leggermente il tempo di esecuzione.

In applicazioni di cloud computing, dove sono presenti un numero molto elevato di core, generalmente le macchine virtuali disponibili utilizzano un numero di core maggiore di una divisione equa, poiché è improbabile che tutte le macchine virtuali utilizzino allo stesso tempo tutti i core presenti. Questo meccanismo di over-provisioning o sovradimensionamento permette di aumentare le prestazioni delle macchine virtuali presenti, permettendole di operare con più risorse di quante siano le risorse fisiche disponibili. Questa tecnica può essere effettuata anche sulla memoria principale o secondaria per ogni macchina virtuale.

## 4.2 Produttori e Consumatori

Un'architettura software molto usata è costituita da un agente produttore che realizza un certo tipo di informazione ed un consumatore che utilizza quest'informazione. Condividono una qualche struttura dati per condividere quest'informazione, accessibile ad entrambi gli agenti. Questo rappresenta un problema di sincronizzazione tra i due, tra i più classici e rappresenta un'architettura di base per sistemi moderni, per realizzare applicazioni moderni, come applicazioni basate su microservizi.

Una pila di 10 elementi interi è condivisa tra due thread: un produttore ed un consumatore.

1. Il produttore deve essere implementato secondo la seguente logica. In un ciclo infinito:

- Deve attendere una quantità di tempo casuale inferiore al secondo;
- Una volta scaduta l'attesa, se la pila è piena, deve attendere che qualche elemento venga rimosso dal consumatore;
- Quando si libera dello spazio nello stack, deve inserire un numero casuale di elementi (senza andare in overflow).

2. Il consumatore deve essere implementato secondo la seguente logica. In un ciclo infinito:

Deve attendere una quantità di tempo casuale inferiore al secondo Una volta scaduta l'attesa, se lo stack è vuoto, deve attendere che qualche elemento venga inserito dal produttore Quando lo stack non è vuoto, deve leggere un numero casuale di elementi, inferiore o uguale al numero di elementi presenti nello stack. Suggerimenti:

- Lo stack può essere implementato con un array di interi, un contatore di elementi già inseriti, e con due funzioni: `push()` e `pop()`;
- Alcune funzioni utili: `random()` e `usleep()`.

Si realizza una struttura dato chiamata `stack` che rappresenta la pila, contenente un array di interi, un intero che indica la dimensione complessiva, ed un ulteriore intero che specifica il numero di posizioni occupate nella pila:

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

#define SIZE 10

typedef struct Stack{
    int* a;    // array
    int size;  // dimensione dell'array
    int used;  // posizione occupate nell'array
}stack;

stack pila;
```

Si realizza una variabile globale pila, poiché è l'unica struttura dati utilizzata nel problema.

Si definiscono quindi le operazioni di inizializzazione della pila, di inserimento e rimozione in coda:

```c
void init(){
    pila.a = (int*) calloc(SIZE, sizeof(int));
    pila.size = SIZE;
    pila.used = 0;
}
```

```
void push(int v){
    if (pila.used < pila.size){
        pila.a[pila.used] = v;
        pila.used++;
    }
}

int pop(){
    int val = -1;
    if (pila.used > 0){
        val = pila.a[pila.used-1];
        pila.used--;
    }
    return val;
}
```

A differenza della funzione `malloc()`, la funzione `calloc()` inizializza tutti i valori della zona di memoria allocata a zero. Le funzioni di pop e push incluse sono semplificate, si utilizza il valore -1 di default per indicare che l'operazione di pop è stata eseguita su di una pila vuota, anche se questa dovrebbe poter gestire interi negativi.

Per poter gestire questi due agenti, bisogna creare due thread, assegnati a due funzioni diverse, una consumatore ed un'altra produttore. Per cui queste due funzioni devono restituire una variabile di tipo puntatore a void, ed accettano un argomento di tipo puntatore a void. Si potrebbe realizzare anche con una fork, ma lo spazio di memoria non sarebbe condiviso e quindi il problema sarebbe più difficile, poiché ogni inserimento o rimozione deve essere effettuato su una terza zona di memoria condivisa oppure su di un file. La programmazione multi-processo è quindi tendenzialmente più difficile di una programmazione multithread, poiché sono necessarie più funzioni del sistema operativo per realizzare le stesse operazioni.

Il produttore itera su un ciclo infinito ed aspetta un intervallo di tempo casuale, fornito da `usleep()` passando come argomento un intero casuale fino a $10^6$, poiché l'unità di misura è in microsecondi. Bisogna effettuare un cast di `1e6`, poiché è di tipo `double` e non è compatibile con l'operatore modulo. Bisogna stabilire quanti elementi da inserire in pila, devono essere in numero inferiore o uguale al numero di posizioni disponibili. Dato questo numero si itera e si inseriscono dei numeri casuali invocando la funzione `push()`. Bisogna inserire un controllo, per verificare se il consumatore deve consumare elementi dalla pila. Per impedire di ripetere continuamente questo ciclo su una pila piena, si attende fino a quando non si libera dello spazio, con un'istruzione condizionale. Il consumatore si implementa analogamente, con un ciclo infinito, che attende per un tempo casuale, e legge nella pila solamente se non è vuota.

Si inseriscono inoltre delle stampe per indicare le operazioni eseguite da questi due agenti.

Si inizializza la pila e si definiscono le due variabili contenenti i thread. Per implementare queste queste due funzioni come due thread si creano con la funzione |pthread_create()|. Non bisogna specificare né il secondo né il quarto parametro di questa funzione, poiché non si sono opzioni da

specificare o il numero di worker. Come primo parametro si inserisce il puntatore al thread, e come terzo la funzione da eseguire.

A questo punto l'esecuzione di questo programma non produce nulla, poiché il thread principale termina prima dell'esecuzione di questi due thread, terminando anche loro. Il thread principale deve attendere la terminazione dei figli, ma questi eseguono un ciclo infinito quindi non finiranno mai, bisogna essere consapevoli di questo, poiché il programma deve essere terminato con la sequenza "Ctrl + C". Si utilizza la funzione `pthread_join()`, specificando come primo parametro i thread di cui bisogna aspettare l'esecuzione ed un parametro nullo:

```
int main(){
    init();
    pthread_t cons, prod;
    pthread_create(&cons, NULL, consumatore, NULL);
    pthread_create(&prod, NULL, produttore, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);
    pthread_mutex_destroy(&pila);
}
```

Questi due thread possono accedere alla stessa area di memoria contemporaneamente e quindi bisogna inserire un lock per impedire che tentino di effettuare contemporaneamente operazioni sulla pila.

Per inserire questo lock si utilizzano due funzioni `lock()` e `unlock()`, passando come argomento un riferimento alla struttura dati che si vuole bloccare. Si avrà quindi per il produttore:

```
void* produttore(void* arg){
    while(1){
        usleep(random()%(int)1e6);
        pthread_mutex_lock(&pila);
        if(pila.used <= pila.size){
                int aggiunte = random()%(pila.size - pila.used);
                for( int i = 0; i < aggiunte; i++ ){
                    int add = random();
                    push(random());
                    printf("[Produttore]: Inserito %d\n", add);
            }
        }
        pthread_mutex_unlock(&pila);
    }

}
```

E per il consumatore:

```
void* consumatore(void* arg){
    while(1){
        usleep(random()%(int)1e6);
        pthread_mutex_lock(&pila);
        if(pila.used < pila.size){
                int rimozioni = random()%pila.used;
                for( int i = 0; i < rimozioni; i++ ){
                    printf("[Consumatore]: Rimosso %d\n", pop());
            }
        }
        pthread_mutex_unlock(&pila);
    }
}
```

## 4.3 Produttori e Consumatori su File[4]

Un file su disco ha il seguente formato:

```
<numero_record><record 1><record 2>...
```

Dove:

- `<numero_record>` è un intero rappresentante il numero di record attualmente presenti all'interno del file;
- `<record1><record2>...` sono ognuno un numero intero.

Il file è acceduto da due thread: un produttore ed un consumatore, ed è gestito come se fosse una pila: i nuovi elementi vengono accodati al termine del file, e la lettura (con contestuale rimozione) degli elementi avviene dall'ultimo elemento del file. Il file non deve contenere più di 10 record oltre all'indicatore iniziale del numero di record presenti.

I due thread, produttore e consumatore, hanno il seguente comportamento:

1. Il produttore, in un ciclo infinito:
   (a) Deve attendere una quantità di tempo casuale inferiore al secondo;
   (b) Una volta scaduta l'attesa, se la pila contenuta nel file è piena, deve attendere che qualche elemento venga rimosso dal consumatore;
   (c) Quando si libera dello spazio nella pila, deve inserire un numero casuale di elementi (senza andare in overflow rispetto alle dimensioni della pila) ed aggiornare il contatore all'inizio del file.
2. Il consumatore, in un ciclo infinito:
   (a) Deve attendere una quantità di tempo casuale inferiore al secondo;

[4] *Dall'esercitazione in laboratorio del 3 Dicembre 2024*

(b) Una volta scaduta l'attesa, se la pila è vuota, deve attendere che qualche elemento venga inserito dal produttore;

(c) Quando la pila non è vuota, deve leggere un numero casuale di elementi (inferiore o uguale al numero di elementi presenti nello stack), sostituirne il valore con il numero 0 ed aggiornare il valore all'inizio del file.

**Suggerimento**

Quando si esegue una `read()` o una `write()` su un file, viene spostato un cursore in avanti del numero di byte letti o scritti sul file. Ad esempio, se un file contenesse la stringa "ciaopino" e venisse effettuata una `read()` di 4 byte, questa leggerebbe "ciao". Un'eventuale seconda `read()` di 4 byte leggerebbe invece "pino". Allo stesso modo, se si eseguisse una prima lettura di 4 byte e successivamente una scrittura di 4 byte della stringa "anno", il file conterrebbe la stringa "ciaoanno" al termine dell'esecuzione delle `read()` e delle `write()`. È possibile spostare il cursore anche senza necessariamente effettuare una `read()` o una `write()`, utilizzando la funzione `lseek()`, usa il manuale per scoprire come usare `lseek()`.

**Soluzione**

Questo rappresenta un problema simile a quello mostrato in precedenza, quindi i due thread del produttore e del consumatore sono realizzati in modo molto simile:

```
void* producer(){
    while(!exitCondition){
        usleep((int)random()%(int)1e6);
        pthread_mutex_lock(&lock);
        int dim = getDim();
        int pushDim = random()%(MAX_DIM - dim + 1);
        printf("[Producer]: (%d) Insertion(s): ", pushDim);

        if(pushDim == 0) { printf("[]"); }
        for(int i = 0; i < pushDim; i++) {
            if(i == 0) { printf("[Push]: "); }
            int temp = random()%NUM_DIM;
            printf("[%d] ", temp);
            push(temp);
        }
        printf("\n[Producer]: ");
        printStack();
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}
```

```
void* consumer(){
    while(!exitCondition){
        usleep((int)random()%(int)1e6);
        pthread_mutex_lock(&lock);
        int dim = getDim();
        int popDim = random()%(dim + 1);

        printf("[Consumer]: (%d) Deletion(s):  ", popDim);
        for(int i = 0; i < popDim; i++) {
            if(i == 0) { printf("[Pop]:  "); }
            printf("[%d] ", pop());
        }
        printf("\n[Consumer]: ");
        printStack();
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}
```

La differenza sostanziale consiste nella diversa implementazione delle funzioni `pop()` e `push()`. Mentre nell'esercizio precedente queste accedevano ad una pila condivisa, in questo esercizio devono accedere ad un file esterno. Questo file contiene l'insieme dei record acceduti con politica LIFO. Le due funzioni dopo aver letto la dimensione dal primo intero, scorrono il cursore sul file per aggiungere alla fine un intero, per la `push()` oppure rimuovere un intero azzerandolo, per la `pop()`. Per facilità si realizza un'ulteriore funzione per ottenere la dimensione della pila sul file `getDim()`:

```
int getDim(){
    int fd = open("a.txt", O_RDONLY | O_CREAT, S_IRWXU);
    if(fd < -1) { error("[GetDim]: Error in open()\n"); }

    int dim = 0;
    if(read(fd, &dim, sizeof(int)) < -1) { error("[GetDim]: Error in read()\n");
    ↪  }

    close(fd);
    return dim;
}
```

Mentre le due funzioni `pop()` e `push()` sono:

```
int pop(){
    int dim = getDim() - 1;
    int fd = open("a.txt", O_RDWR);
    if(fd < -1) { error("[Pop]: Error in open()\n"); }
```

```
    // updates the dimension
    if(write(fd, &dim, sizeof(int)) < -1) { error("[Pop]: Error in write()\n"); }


    // reads the last element
    int k;
    lseek(fd, (dim + 1) * sizeof(int), SEEK_SET);
    if(read(fd, &k, sizeof(int)) < -1) { error("[Pop]: Error in read()\n"); }

    // sets the last element to zero, "removing" it
    int zero = 0;
    lseek(fd, (dim + 1) * sizeof(int), SEEK_SET);
    if(write(fd, &zero, sizeof(int)) < -1) { error("[Pop]: Error in write()\n");
    ↪  }

    numPops++;
    close(fd);
    return k;
}


void printStack(){
    int fd = open("a.txt", O_RDONLY | O_CREAT, S_IRWXU);
    if(fd < -1) { error("[PrintStack]: Error in open()\n"); }
    int dim = 0;
    if(read(fd, &dim, sizeof(int)) < -1) { error("[PrintStack]: Error in
    ↪  read()\n"); }
    printf("[PrintStack]: (%d) Item(s):", dim);
    for(int i = 0; i < dim; i++){
        int temp = 0;
        if(read(fd, &temp, sizeof(int)) < -1) { error("[PrintStack]: Error in
        ↪  read()\n"); }

        printf("[%d] ", temp);
        if(i == dim - 1) { printf("\n"); }
    }
    if(dim == 0) { printf("[]\n"); }
    close(fd);
}
```

Nella funzione principale si inizializzano le variabili globali e si attende un periodo di tempo prima di terminare la entrambi i thread. Si può definire un'ulteriore funzione di supporto, non richiesta, per stampare il contenuto del file ad ogni iterazione per facilitare la correzione:

```
void printStack(){
    int fd = open("a.txt", O_RDONLY | O_CREAT, S_IRWXU);
    if(fd < -1) { error("[PrintStack]: Error in open()\n"); }
    int dim = 0;
    if(read(fd, &dim, sizeof(int)) < -1) { error("[PrintStack]: Error in
    ↪  read()\n"); }
    printf("[PrintStack]: (%d) Item(s):", dim);
    for(int i = 0; i < dim; i++){
        int temp = 0;
        if(read(fd, &temp, sizeof(int)) < -1) { error("[PrintStack]: Error in
        ↪  read()\n"); }

        printf("[%d] ", temp);
        if(i == dim - 1) { printf("\n"); }
    }
    if(dim == 0) { printf("[]\n"); }
    close(fd);
}
```

## 4.4 Simulazione di uno Scheduler

Il seguente esercizio ha lo scopo di simulare lo scheduler di un sistema operativo e l'esecuzione di processi (job), nel caso semplificato in cui l'esecuzione di un processo non può essere interrotta e non è necessario accedere a risorse oltre la CPU.

Si consideri il seguente scenario. Si dispone di un computer dove:

- Il tasso di arrivi dei processi è pari a 10 processi al secondo. In altre parole, è come se venissero eseguite sul computer considerato 10 applicazioni al secondo;
- Il tempo di esecuzione di ogni processo è un valore casuale sempre inferiore a 1 secondo;
- Lo scheduler non supporta più di 1000 processi in coda. Ciò vuol dire che, se la coda è piena, non vengono ricevuti nuovi processi;
- Lo scheduler implementa la politica FIFO.

Scrivere un simulatore che implementi il comportamento dello scenario descritto e che calcoli il turnaround time medio al variare dei seguenti parametri:

- Durata della simulazione: il numero di job che arrivano all'interno del sistema è pari a 10, 20, 30;
- Numero delle CPU: il computer in considerazione può avere 1, 2, 4 CPU.

### Suggerimento

Si suggerisce di utilizzare come base di partenza l'implementazione del produttore-consumatore. La struttura dati rappresentante la coda di processi è disponibile nel file queue.h.

Prendere familiarità con le strutture dati definite in queue.h, in particolare con la struct Process. Quest'ultima può essere utilizzata per memorizzare informazioni utili quali l'ID del processo, il suo tempo di esecuzione, il tempo di arrivo all'interno del sistema, il tempo di avvio ed il tempo di completamento. Questa struct Process è utilizzata per costruire una lista bidirezionale che può essere acceduta con diverse funzioni, tra le quali: `enqueue(Queue* queue, Process* data)` e `dequeue(Queue* queue)`: la prima aggiunge un processo in coda alla lista, la seconda rimuove e restituisce il processo in testa alla lista. Per poter inizializzare la coda è necessario utilizzare la funzione `initializeQueue(Queue* queue)`.

È possibile implementare la simulazione estendendo il problema del produttore-consumatore, in particolare:

- Il thread produttore può essere implementato in modo tale da generare processi alla velocità indicata in precedenza;
- Il thread consumatore può rappresentare una CPU che estrae un processo dalla coda e lo esegue.

### Soluzione

Si considerano due tipi di thread, un thread produttore che può creare processi ad una frequenza di 10/20/30 processi al secondo, realizzata tramite una `usleep()` relativa all'intervallo di tempo tra due processi:

```
void* job(){
    while(!exitCondition){
        usleep((int)1e5/jobFrequency);
        pthread_mutex_lock(&lock);
        if(size >= MAX_QUEUE_SIZE){
            pthread_mutex_unlock(&lock);
            continue;
        }
        pthread_mutex_unlock(&lock);

        Process* p = newProcess();
        int tempSize = 0;

        pthread_mutex_lock(&lock);
        tempSize = ++size;
        enqueue(queue, p);
        pthread_mutex_unlock(&lock);
```

```
        printf("[Job]: Added New Process in Queue with PID (%d), Queue Size
        ↪  (%d)\n", pid, tempSize);
    }
    return NULL;
}
```

Il secondo tipo di thread rappresenta una CPU ed è in grado di rimuovere un processo dalla coda e simulare la sua esecuzione, sempre con una `usleep()` per il tempo descritto dentro al processo. Quando il thread CPU ha finito l'esecuzione di un processo aggiorna il turnaround time medio ed attende che la coda sia libera per poter rimuovere un ulteriore processo:

```
void* cpu(){
    while(!exitCondition){
        int tempSize = 0;
        pthread_mutex_lock(&lock);
        if(size <= 0){
            pthread_mutex_unlock(&lock);
            continue;
        }
        Process* exec = dequeue(queue);
        tempSize = --size;
        pthread_mutex_unlock(&lock);

        // assigns the process start time
        exec->start = getTime();
        printf("[CPU]: Removed Process from Queue with PID (%ld), Start Execution
        ↪  Time %s", exec->id, asctime(localtime(&(exec->start.tv_sec))));
        // waits for the process execution
        usleep((long)exec->exec_time);

        // assigns the process end time
        exec->end = getTime();
        printf("[CPU]:   Finished Execution of Process PID (%ld),   End Execution
        ↪  Time %s", exec->id, asctime(localtime(&(exec->end.tv_sec))));

        // updates the turnaround time (mutex locking inside the function)
        updateTTime(exec);

        free(exec);
    }
    return NULL;
}
```

La funzione `getTime()` assegna il timestamp corrente all'attributo di tipo `struct timeval` del processo, come il numero di microsecondi trascorsi dal primo Gennaio 1970. Utilizzato anche nella

funzione `newProcess()`. Non utilizza si utilizza un lock per questa funzione poiché la variabile globale `pid` contenente l'ultimo id di un processo generato viene acceduta solamente dal singolo thread produttore e non provoca una race condition. Si utilizzano delle variabili temporanee `temp*` per non dover utilizzare dei lock anche nelle stampe, questo infatti diminuirebbe l'efficacia dei thread, essendo abbastanza onerose rispetto a semplici assegnazioni. Così anche nella funzione `updateTTime()` per aggiornare il tempo di ritorno medio, dove i lock vengono chiamati all'interno della funzione.

Nella funzione principale, vengono testate diverse combinazioni di frequenze di arrivo e numero di CPU. Ad ogni iterazione vengono ri-inizializzate le variabili globali ed i vari thread, per realizzare una graduatoria in base al tempo medio di ritorno tra tutte queste combinazioni. Essendo il codice abbastanza lungo, si omette la sua scrittura completa, è comunque disponibile su scheduler.c come tutti gli altri esercizi.

## 4.5 Problema dei Filosofi a Cena

Cinque filosofi siedono ad una tavola rotonda con un piatto di spaghetti davanti e una forchetta a sinistra. Ci sono dunque cinque filosofi, cinque piatti di spaghetti e cinque forchette.

Si immagini che la vita di un filosofo consista di periodi alterni di mangiare e pensare, e che ciascun filosofo abbia bisogno di due forchette per mangiare, ma che le forchette vengano prese una per volta. Dopo essere riuscito a prendere due forchette il filosofo mangia per un po', poi lascia le forchette e ricomincia a pensare. Il problema consiste nello sviluppo di un algoritmo che impedisca lo stallo (deadlock) o la morte d'inedia (starvation). Il deadlock può verificarsi se ciascuno dei filosofi tiene in mano una forchetta senza mai riuscire a prendere l'altra. Il filosofo F1 aspetta di prendere la forchetta che ha in mano il filosofo F2, che aspetta la forchetta che ha in mano il filosofo F3, e così via in un circolo vizioso. La situazione di starvation può verificarsi indipendentemente dal deadlock se uno dei filosofi non riesce mai a prendere entrambe le forchette.

Implementare un programma basato sulla libreria pthread dove i cinque filosofi sono realizzati tramite 5 thread, e si cerchi di evitare situazioni di deadlock e starvation.

Per risolvere questo problema bisogna evitare di gestire le forchette come un array circolare, poiché si incontrerebbe una situazione dove ogni filosofo ha in mano la forchetta alla sua sinistra, o destra, un deadlock. Inoltre è importante l'ordine in cui vengono prese le forchette, e quindi l'ordine in cui vengono usati i lock. In quest'implementazione ogni forchetta ha il suo lock, le l'uso delle forchette ed il lock sono rappresentati come due array di cinque elementi. Ogni filosofo inoltre ha un suo ID che lo identifica, passato al momento della creazione dei thread, da 1 a 5. Se si tentasse di accedere sempre alla forchetta indirizzata dal proprio ID, si incontrerebbe una situazione di stallo, realizzato da un array circolare. Per cui ogni $i$-esimo filosofo quando si trova in procinto di mangiare determina l'indice delle due forchette che necessita, $i-1$ ed $i \mod 5$. Per impedire che ogni filosofo abbia in mano la forchetta di indice $i-1$ si impone che la prima forchetta che ogni filosofo controlla è quella di indice minore, tra $i-1$ e $i \mod 5$. In questo modo è garantito che non si incontri mai la situazione di stallo circolare. Poiché se il filosofo cui $i \mod 5 < i$ non riesce a prendere la forchetta di indice $i \mod 5$ comincia a pensare, poiché non ha in mano alcuna forchetta, e quindi la forchetta $i-1$ è libera per essere usata dal filosofo successivo nella tavola, che potrà mangiare ed in seguito liberare le sue due forchette. In questo modo si può continuare lungo l'intera tavola

facendo mangiare almeno un filosofo alla volta, per ritornare al filosofo $i$-esimo iniziale che vedrà la forchetta $i \mod 5$ libera e potrà mangiare, dato che la forchetta $i-1$ verrà sicuramente liberata, se non è già libera. Si considera quindi la seguente implementazione del thread filosofo:

```
// every ith-philosopher can access only fork[i - 1] and fork[i%5]
// there are only 5 forks with id 0~4, each corresponding to the jth lock
void* philosopher(void* arg){
        // the id is set before entering the loop
        int id = *((int*) arg);
        // to determine the philosopher next action (1) is eat, (0) is think
        int toEat = 1;
        // int toEat = random()%2; // alternate version
        // sets the waitTime
        struct timeval timeSinceLastAte;
        resetsTime(&timeSinceLastAte);
        while(!exitCondition){
                // check if died
                if(isDead(&timeSinceLastAte)){
                        fprintf(stderr, "[Philosopher %d]: Starved to
                        ↪  Death...\n", id);
                        exit(EXIT_FAILURE);
                }

                if(toEat){
                        // first check the lowest ranking fork (the one on their
                        ↪  left)
                        // circularity -> deadlock
                        int leftFork = id - 1;
                        int rightFork = id%5;
                        if(leftFork > rightFork){ swap(&leftFork, &rightFork); }

                        pthread_mutex_lock(lock + leftFork);
                        pthread_mutex_lock(lock + rightFork);
                        if(forkUsed[leftFork] || forkUsed[rightFork]){
                                // the forks are not free, better wait
                                pthread_mutex_unlock(lock + rightFork);
                                pthread_mutex_unlock(lock + leftFork);
                                continue;
                        }

                        // both of them are free! picks them up
                        forkUsed[leftFork] = 1;
                        forkUsed[rightFork] = 1;
                        pthread_mutex_unlock(lock + rightFork);
```

```
                        pthread_mutex_unlock(lock + leftFork);

                        eat(random()%(int)MAX_EAT, id);

                        // ate, put them down
                        pthread_mutex_lock(lock + leftFork);
                        pthread_mutex_lock(lock + rightFork);
                        forkUsed[leftFork] = 0;
                        forkUsed[rightFork] = 0;
                        pthread_mutex_unlock(lock + leftFork);
                        pthread_mutex_unlock(lock + rightFork);

                        // if successfully eaten, sets action to (0) to think,
                        ↪  and resets the wait time
                        toEat = 0;
                        resetsTime(&timeSinceLastAte);
                }else{
                        think(random()%(int)MAX_THINK, id);
                        toEat = 1;
                }
        }
        return NULL;
}
```

I nomi degli indici `leftFork` e `rightFork` non sono propriamente corretti, poiché come descritto la prima forchetta ad essere presa `leftFork` non coincide sempre con la forchetta alla sinistra del filosofo.

Utilizzando questa strategia tuttavia, se la tavola è sufficientemente ampia, ovvero sono presenti molti di più di cinque filosofi, il primo filosofo di questa catena potrebbe morire di inedia a causa della lunghezza. Per tavole più grandi quindi si potrebbe imporre che un filosofo deve scegliere casualmente quale forchetta tenta di prendere per prima. Ma in questo modo non è garantito che ogni filosofo sia in grado di mangiare prima di morire di inedia. Non esiste infatti una soluzione in grado di impedire sia lo stallo che la morte di inedia di uno dei due filosofi, bisogna sceglierla in base alle caratteristiche del problema corrente, e dalle prestazioni richieste.

## 4.6 Problema del Barbiere Sonnolento

Un barbiere possiede un negozio con una sola sedia da lavoro e un certo numero limitato di posti per attendere. Se non ci sono clienti il barbiere dorme. All'arrivo del primo cliente il barbiere si sveglia ed inizia a servirlo. Se dovessero sopraggiungere clienti durante il periodo di attività del barbiere, essi si mettono in attesa sui posti disponibili. Al termine dei posti di attesa, un ulteriore cliente viene scartato. Questa problematica è molto vicina al sistema di funzionamento degli helpdesk informatizzati dove l'operatore serve, uno per volta, tutti i clienti in coda oppure attende, senza effettuare alcuna operazione in particolare, l'arrivo di nuove chiamate. Una corretta

programmazione concorrente deve far "dormire" il barbiere in assenza di clienti, attivare il barbiere sul primo cliente al suo arrivo e mettere in coda tutti i successivi clienti tenendoli inattivi.

Questo problema è molto simile alla simulazione di uno scheduler, la differenza principale è la capacità del thread consumatore di aspettare dormendo l'arrivo di altri clienti. I clienti vengono realizzati utilizzando la stessa struttura dati `Process`. Per poter aspettare l'arrivo di altri clienti, senza effettuare alcuna azione, un'implementazione rudimentale utilizza la funzione `usleep()` per aspettare un periodo di tempo casuale, proporzionale al tempo di arrivo di un cliente:

```
void* barber(){
        Process* currentCustomer = NULL;
        while(!exitCondition){
                pthread_mutex_lock(&lock);
                if(waitSize == 0){
                        pthread_mutex_unlock(&lock);
                        printf("[Barber]: No Customer(s)... Sleeping!\n");
                        usleep(random()%(int)MAX_SLEEP);
                        continue;
                }
                currentCustomer = dequeue(queue);
                waitSize--;
                pthread_mutex_unlock(&lock);
                printf("[Barbar]: Customer(s)... Waking Up\n");
                currentCustomer->start = getTime();
                printf("[Barber]:   Serving Customer (%ld). Starting Haircut
                ↪  Time: %s", currentCustomer->id,
                ↪  asctime(localtime(&(currentCustomer->start.tv_sec))));

                usleep((int)currentCustomer->exec_time);

                currentCustomer->end = getTime();
                printf("[Barber]:    Served Customer (%ld). Finished Haircut
                ↪  Time: %s", currentCustomer->id,
                ↪  asctime(localtime(&(currentCustomer->start.tv_sec))));

                free(currentCustomer);
        }
        return NULL;
}
```

Una soluzione più corretta consiste nell'utilizzare segnale o semafori dalle librerie `signals.h` e `semaphore.h` per poter inviare segnali tra i thread, in caso di arrivo di clienti. Oppure si potrebbero utilizzare thread condizionali, forniti dalla stessa libreria `pthread.h` per effettuare queste operazioni.

Il thread produttore è essenzialmente lo stesso del simulatore di uno scheduler:

```
void* customer(){
        Process* customer = NULL;
        while(!exitCondition){
                usleep(random()%(int)MAX_ARRIVAL_TIME);
                pthread_mutex_lock(&lock);
                if(waitSize >= MAX_WAIT_SIZE){
                        pthread_mutex_unlock(&lock);
                        printf("[Customer]: Queue Full, Can't Enter\n");
                        continue;
                }
                pthread_mutex_unlock(&lock);
                customer = newCustomer();

                printf("[Customer]: Entered Customer (%ld), Wakey Wakey! Haircut
                ↪  Time (%.2f [sec]) and Arrival Time %s", customer->id,
                ↪  customer->exec_time/1e6,
                ↪  asctime(localtime(&(customer->arrival.tv_sec))));

                pthread_mutex_lock(&lock);
                enqueue(queue, customer);
                waitSize++;
                pthread_mutex_unlock(&lock);
        }
        return NULL;
}
```