ŚPIĄCY FRYZJER DOKUMENTACJA

Zadanie przygotowane przez Emanuel Korycki i Patryk Rybak.

SPIS TREŚCI

Opis implementacji kolejki – myQueue.c

```c
struct Queue *addToQue(struct Queue *que, int clientId){

    Que *new = malloc(sizeof(Que));
    new->id = clientId;
    new->next = NULL;

    if(que == NULL){
        que = new;
    }else{
        Que *temp = que;
        while(temp->next != NULL){
            temp = temp->next;
        }
        temp->next = new;
    }
    return que;
}
```

Funkcja addToQue tworzy nowy węzeł i dodaje go do kolejki que na podstawie przekazanych argumentów clientId. Jeśli kolejka jest pusta, nowy węzeł staje się pierwszym elementem kolejki.

```c
struct Queue *deleteFirstFromQue(struct Queue *que){
    Que *new = que->next;
    free(que);
    return new;
}
```

Funkcja deleteFirstFromQue usuwa pierwszy element z kolejki que, zwalniając pamięć zajmowaną przez ten węzeł.

```c
void printQue(struct Queue *que){
    Que *temp = que;
    while(temp != NULL){
        printf("%d | ", temp->id);
        temp = temp->next;
    }
    printf("\n");
}
```

Funkcja printQue wypisuje zawartość kolejki que, przechodząc przez kolejne węzły i wyświetlając wartość id każdego węzła.

Opis pliku nagłówkowego – myQueue.h

```c
typedef struct Queue{
    int id;
    struct Queue *next;
}Que;
```

Zdefiniowanie prostej struktury danych przetrzymującej id klienta i wskaźnik do następnej pozycji w kolejce.

```c
Que *barberQue;
Que *leftQue;
```

Inicjalizowane są wskaźniki barberQue i leftQue jako NULL, które będą służyć do śledzenia kolejek.

```c
struct Queue *addToQue(struct Queue *que, int clientId);
struct Queue *deleteFirstFromQue(struct Queue *que);
void printQue(struct Queue *que);
```

Inicjalizacja metod do obsługi naszej kolejki.

Opisanie wspólnych części rozwiązań

```
int NUMBER_OF_BARBERS = 1;   //numer of active hairdresser seats
int NUMBER_OF_SEATS_IN_WAITROOM = 5; //capacity of wait room
int NUMBER_OF_CLIENTS = 10; //number of threads
int TIME_OF_CUTTING = 5; //time it takes to cut a client

int INFO = 0;
```

Zmienne globalne ustawiające nasz program.

- NUMBER_OF_BARBERS – określa ilość wątków fryzjerów, które będziemy tworzyć
- NUMBER_OD_SEATS_IN_WAITROOM – określa liczbę miejsc w poczekalni
- NUMBER_OF_CLIENTS – określa liczbę wątków fryzjerów, które będziemy tworzyć
- TIME_OF_CUTTING – określa stały czas jednego strzyżenia
- INFO – określa czy mamy wyświetlać kolejki do fryzjera

```
int clientsInWaitingRoom = 0;
int currentlyCutting = 0;
int clientsLeft = 0;
int elementsInBarberQue = 0;
int nextClient = 0;
```

Zmienna globalne, które będą zmieniać wątki.

- clientsInWaitingRoom – określa liczbę klientów obecnie znajdujących się w poczekalni
- currentlyCuttin – określa id klienta, który jest obecnie obcinany przez fryzjera
- clientsLeft – zlicza liczę klientów, którym nie udało się dostać do fryzjera
- elementsInBarberQue – zlicza liczbę klientów zapisanych do kolejki do fryzjera
- nextClient – przechowuje następnego klienta w kolejce

```
pthread_mutex_t mutexWaitroom = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutexCurrentlyCutting = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutexClientsLeft = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutexQue = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutexNextClient = PTHREAD_MUTEX_INITIALIZER;
```

Stworzenie mutesów

- mutexWaitroom – odpowiada za zmienna clientsInWaitingRoom
- mutexCurrentlyCutting – odpowiada za zmienna currentlyCutting
- mutexClientsLeft – odpowiada za mienna clientsLeft
- mutexQue – odpowiada za operacje przeprowadzane w ramach kolei Que
- mutexNextClient – odpowiada za zmienną nextClient

```
void printInfo(){…
```

Odpowiada za wyświetlanie informacji o stanie programu w formie:

```
Resigned: 0     Wait Room: 1/5     [in:-]
```

Lub

```
Resigned: 3     Wait Room: 4/5     [in:3]
        BarberQue: 4 | 5 | 6 | 7 |
        LeftQue: 8 | 9 | 10 |
```

Jest ona wywoływana podczas każdej zmiany stanu programu.

```
void deleteFirstFromBarberQue(){
    pthread_mutex_lock(&mutexQue);
    barberQue = deleteFirstFromQue(barberQue);
    elementsInBarberQue--;
    pthread_mutex_unlock(&mutexQue);
}
```

Usuwa pierwszy element z kolejki barberQue.

```
int getFirstElementFromQue(struct Queue *que){
    pthread_mutex_lock(&mutexQue);
    int nextClient = que->id;
    pthread_mutex_unlock(&mutexQue);
    return nextClient;
}
```

Pobiera id pierwszego klienta z przekazanej w argumencie kolejki.

```
void addToQueSave(struct Queue **que, long clientId){
    pthread_mutex_lock(&mutexQue);
    *que = addToQue(*que, clientId);

    if(*que == barberQue)
        elementsInBarberQue++;

    pthread_mutex_unlock(&mutexQue);
}
```

Dodaje id klienta do odpowiedniej kolejki.

```
void doCutting(){
    sleep(TIME_OF_CUTTING);
}
```

Wykonanie operacji ścinania klienta.

```
void takeClientFromWaitRoom(){
    pthread_mutex_lock(&mutexWaitroom);
    clientsInWaitingRoom--;
    pthread_mutex_unlock(&mutexWaitroom);
}
```

Zmniejsza zmienna clientsInWaitRoom o 1.

```
void setCurrentlyCutting(long clientId){
    pthread_mutex_lock(&mutexCurrentlyCutting);
    currentlyCutting = clientId;
    pthread_mutex_unlock(&mutexCurrentlyCutting);
}
```

Ustawia zmienną currentlyCutting na id klienta.

```
void clientLeft(){
    pthread_mutex_lock(&mutexCurrentlyCutting);
    currentlyCutting = 0;
    pthread_mutex_unlock(&mutexCurrentlyCutting);
}
```

Resetuje zmienną currentlyCutting.

```
void initializeThreads(){…
```

Funkcjia inicjalizuje wątki:

```
pthread_t clients[NUMBER_OF_CLIENTS];
pthread_t barberThread[NUMBER_OF_BARBERS];
```

```
for(long i=0; i<NUMBER_OF_BARBERS; i++){
    if(pthread_create(&barberThread[i], NULL, barber, (void*) i)){
            perror("Failed to create barber thread!\n");
            exit(EXIT_FAILURE);
        }
}

for(long i=0; i<NUMBER_OF_CLIENTS; i++){
    randomSleep();
    //sleep(1);
    if(pthread_create(&clients[i], NULL, customer, (void*) i + 1)){
        perror("Failed to create customer thread!\n");
        exit(EXIT_FAILURE);
    }
}
```

Przy tworzeniu wątków fryzjera przekazujemy funkcję Barber, a przy tworzeniu wątków klienta przekazujemy funkcje client, jednak tym razem przekazujemy również jego unikalne id, zaczynając od 1.

```
for(long i=0; i<NUMBER_OF_BARBERS; i++){
    if(pthread_join(barberThread[i], NULL)){
        perror("Failed to join barber thread!\n");
        exit(EXIT_FAILURE);
    }
}

for(long i=0; i<NUMBER_OF_CLIENTS; i++){
    if(pthread_join(clients[i], NULL)){
        perror("Failed to join customer thread!\n");
        exit(EXIT_FAILURE);
    }
}
```

I na koniec niszczy utworzone mutexy.

```
if(pthread_mutex_destroy(&mutexWaitroom) != 0){
    perror("Failed to destroy mutex mutexWaitroom!\n");
}
if(pthread_mutex_destroy(&mutexCurrentlyCutting) != 0){
    perror("Failed to destroy mutex mutexCurrentlyCutting!\n");
}
if(pthread_mutex_destroy(&mutexClientsLeft) != 0){
    perror("Failed to destroy mutex mutexClientsLeft!\n");
}
if(pthread_mutex_destroy(&mutexQue) != 0){
    perror("Failed to destroy mutex mutexQue!\n");
}
```

W zależności od sposobu rozwiązania wykonuje też inne operacje, które będą opisane w odpowiedniej dla nich sekcji dokumentacji.

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

Funkcja główna, przyjmuje opcje programu, które mogą wyglądać następująco:

```
./run 10 5 -info
```

- ./run – jest to nazwa programu
- 10 – oznacza liczbę klientów (watków), które będziemy tworzyć

- 5 – oznacza czas każdego strzyżenia
- -info – (nieobowiązkowa) flaga, która zaznacza, że chcemy wyświetlić dokładne kolejki do fryzjera i osób, które się nie dostały

```
if(argc < 2){
    printf("Expected number of clients (threads) and time of cutting!\n");
    return EXIT_FAILURE;
}

if(argc < 3){
    printf("Expected time of cutting!\n");
    return EXIT_FAILURE;
}
```

Podaje informacje o braku odpowiednich parametrów programu.

```
if(isdigit(*argv[1])){
    NUMBER_OF_CLIENTS = atoi(argv[1]);
}else{
    printf("Wrong value. Expected a number of clients (threads)!\n");
    return EXIT_FAILURE;
}

if(isdigit(*argv[2])){
    TIME_OF_CUTTING = atoi(argv[2]);
}else{
    printf("Wrong value. Expected time of cutting!\n");
    return EXIT_FAILURE;
}

if(argc > 3 && (strcmp(argv[3], "-info") == 0 || strcmp(argv[3], "-INFO") == 0)){
    INFO = 1;
}
```

Wyświetlenie błędów w przypadku podania nieprawidłowych wartości parametrów.

```
printf("STATUS | CLIENTS %d | CUT TIME %d | INFO %d\n", NUMBER_OF_CLIENTS, TIME_OF_CUTTING, INFO);
```

Wyświetlenie ustawień programu.

```
initializeThreads();
```

Inicjalizacja wątków.

Rozwiązanie problemu przy użyciu semaforów – semaphores.c

```
sem_t *semClient; //signal client is ready
sem_t *semBarber; //wakes barber up
```

Stworzenie zmiennych globalnych semaforów.

Dodatkowa zawartość w funkcji initializeThreads():

```
sem_unlink("/semClient");
sem_unlink("/semBarber");

semBarber = sem_open("/semBarber", O_CREAT|O_EXCL, S_IRWXU, 0);
semClient = sem_open("/semClient", O_CREAT|O_EXCL, S_IRWXU, 0);
```

Utworzenie nazwanych semaforów. W systemie macOs na którym tworzone było rozwiązanie nienazwane semafory nie są wspierane, dlatego w naszym rozwiązaniu używamy nazwanych. Najpierw odłączamy semafory o podanych nazwach, a potem tworzymy je na nowo.

```
if(sem_close(semClient) != 0){
    perror("Failed to destroy semaphore semClient!\n");
}
if(sem_close(semBarber) != 0){
    perror("Failed to destroy semaphore semBarber!\n");
}
```

Zamknięcie semaforów na sam koniec funkcji initializeThreads().

```
void* barber(void* args){
    while(true){
        pthread_mutex_lock(&mutexWaitroom);
        while(clientsInWaitingRoom == 0){
            pthread_mutex_unlock(&mutexWaitroom);
            if(sem_wait(semBarber) != 0){
                perror("Sem wait semBarber error!\n");
            }
        }
        pthread_mutex_unlock(&mutexWaitroom);
        //take next client id
        int clientId = getFirstElementFromQue(barberQue);
        //signal to the nextClient that barber is ready
        pthread_mutex_lock(&mutexNextClient);
        nextClient = clientId;
        pthread_mutex_unlock(&mutexNextClient);
        if(sem_post(semClient) != 0){
            perror("Cond signal condClient error!\n");
        }
        takeClientFromWaitRoom();
        doCutting();
        clientLeft();
        printInfo();
    }
    return NULL;
}
```

Funkcja Barber odpowiada za strzyżenie klientów. Działa ona w nieskończonej pętli obcinając klientów jeżeli są w poczekalni (waitRoom). Poniżej wyjaśnienie działania:

- Jeżeli nie ma klientów w poczekalni to fryzjer czeka. Na czerwono została oznaczona sekcja z wykorzystaniem sem_wait()
- Kiedy pojawia się sygnał od klienta, że jest gotowy, to pobieramy jego id i wysyłamy mu wezwanie na fotel. (sekcja na zielono z wykorzystaniem sem_post())
- Następnie w funkcji takeClientFromWaitRoom() odejmujemy 1 klienta z poczekalni.
- Fryzjer wykonuje usługę
- Zwalnia się fotel fryzjera – pętla zaczyna się od nowa

```c
void* client(void* args){

    long clientId = (long) args;
    printf("Client %ld comes...\n", clientId);
    pthread_mutex_lock(&mutexWaitroom);
    if(clientsInWaitingRoom < NUMBER_OF_SEATS_IN_WAITROOM){
        clientsInWaitingRoom++;
        pthread_mutex_unlock(&mutexWaitroom);
        addToQueSave(&barberQue, clientId);
        printInfo();

        //signal client ready
        if(sem_post(semBarber) != 0){
            perror("Sem post semBarber error!\n");
        }

        //wait for client turn
        pthread_mutex_lock(&mutexNextClient);
        while(nextClient != clientId){
            pthread_mutex_unlock(&mutexNextClient);
            if(sem_wait(semClient) != 0){
                perror("Cond wait condClient error!\n");
            }
            pthread_mutex_lock(&mutexNextClient);
        }
        pthread_mutex_unlock(&mutexNextClient);

        //set id to currently cutting
        setCurrentlyCutting(clientId);

        //client goes for cutting
        //printf("TEST 1 CUST\n");
        deleteFirstFromBarberQue();
        printInfo();
    }else{
        pthread_mutex_unlock(&mutexWaitroom);
        pthread_mutex_lock(&mutexClientsLeft);
        clientsLeft++;
        pthread_mutex_unlock(&mutexClientsLeft);
        addToQueSave(&leftQue, clientId);
        printInfo();
    }
    return NULL;
}
```

Funkcja client obsługuje wątki klienta. Poniżej sposób działania

- Klient pobiera clientId z argumentu funkcji
- Klient sprawdza czy jest miejsce w poczekalni
- Jeżeli jest to:
    - Klient zajmuje miejsce w poczekalni (clientsInWaitRoom++)
    - Klient jest zapisany do kolejki do fryzjera
    - Wyświetlamy zaktualizowane informacje
    - Budzi fryzjera (sekcja na czerwono) – sem_post()
    - Czeka na swoją kolej (sekcja na zielono) – sem wait
    - Kiedy przychodzi jego kolej to siada na fotel (setCurrentlyCutting = clientId)
    - Usuwamy klienta z kolejki do fryzjera
    - Wyświetlamy zaktualizowane informacje
- Jeżeli nie jest
    - Zapisujemy do na liście odrzuconych klientów
    - Dodajemy klienta do listy klientów, którzy nie dostali się do fryzjera i odeszli
    - Wyświetlamy zaktualizowane informacje

Rozwiązanie problemu przy użyciu zmiennych warunkowych – conditio_variables.c

```
pthread_cond_t condBarber = PTHREAD_COND_INITIALIZER;
pthread_cond_t condClient = PTHREAD_COND_INITIALIZER;
```

Inicjalizacja zmiennych warunkowych.

```
if(pthread_cond_destroy(&condBarber) != 0){
    perror("Failed to destroy condition variable condBarber!\n");
}
if(pthread_cond_destroy(&condClient) != 0){
    perror("Failed to destroy condition variable condClient!\n");
}
```

Zniszczenie zmiennych warunkowych – na końcu funkcji initializeThreads().

```
void* barber(void* args){
    while(true){
        //if clientsWaiting = 0 then wait for client to show up
        pthread_mutex_lock(&mutexWaitroom);
        while(clientsInWaitingRoom == 0){
            pthread_mutex_unlock(&mutexWaitroom);
            pthread_mutex_lock(&mutexCurrentlyCutting);
            if(pthread_cond_wait(&condBarber, &mutexCurrentlyCutting) != 0){
            perror("Sem wait semBarber error!\n");
            }
            pthread_mutex_unlock(&mutexCurrentlyCutting);
            pthread_mutex_lock(&mutexWaitroom);
        }
        pthread_mutex_unlock(&mutexWaitroom);
        //take next client id
        int clientId = getFirstElementFromQue(barberQue);
        //signal to the next client that now is their turn
        pthread_mutex_lock(&mutexNextClient);
        nextClient = clientId;
        if(pthread_cond_signal(&condClient) != 0){
            perror("Cond signal condClient error!\n");
        }
        pthread_mutex_unlock(&mutexNextClient);

        takeClientFromWaitRoom();
        doCutting();
        clientLeft();
        printInfo();
    }

    return NULL;
}
```

Funkcja Barber odpowiada za strzyżenie klientów. Działa ona w nieskończonej pętli obcinając klientów jeżeli są w poczekalni (waitRoom). Poniżej wyjaśnienie działania:

- Jeżeli nie ma klientów w poczekalni to fryzjer czeka. Na czerwono została oznaczona sekcja z wykorzystaniem sem_wait()
- Kiedy pojawia się sygnał od klienta, że jest gotowy, to pobieramy jego id i wysyłamy mu wezwanie na fotel. (sekcja na zielono z wykorzystaniem sem_post())
- Następnie w funkcji takeClientFromWaitRoom() odejmujemy 1 klienta z poczekalni.
- Fryzjer wykonuje usługę
- Zwalnia się fotel fryzjera – pętla zaczyna się od nowa

```c
void* client(void* args){

    long clientId = (long) args;

    printf("Client %ld comes...\n", clientId);
    //check if client can get into wait room
    pthread_mutex_lock(&mutexWaitroom);
    if(clientsInWaitingRoom < NUMBER_OF_SEATS_IN_WAITROOM){
        clientsInWaitingRoom++;
        pthread_mutex_unlock(&mutexWaitroom);
        //add client to the barberQue
        addToQueSave(&barberQue, clientId);
        printInfo();

        //signal client ready
        if(pthread_cond_signal(&condBarber) != 0){
            perror("Cond signal condBarber error!\n");
        }

        //wait for client turn
        pthread_mutex_lock(&mutexNextClient);
        while(nextClient != clientId){
            if(pthread_cond_wait(&condClient, &mutexNextClient) != 0){
                perror("Cond wait condClient error!\n");
            }
        }
        pthread_mutex_unlock(&mutexNextClient);

        //set id to currently cutting
        setCurrentlyCutting(clientId);
        //client goes for cutting
        deleteFirstFromBarberQue();
        printInfo();

    }else{
        //note that client left
        pthread_mutex_unlock(&mutexWaitroom);
        pthread_mutex_lock(&mutexClientsLeft);
        clientsLeft++;
        pthread_mutex_unlock(&mutexClientsLeft);
        addToQueSave(&leftQue, clientId);
        printInfo();
    }
    return NULL;
}
```

Funkcja client obsługuje wątki klienta. Poniżej sposób działania

- Klient pobiera clientId z argumentu funkcji
- Klient sprawdza czy jest miejsce w poczekalni
- Jeżeli jest to:
    - Klient zajmuje miejsce w poczekalni (clientsInWaitRoom++)
    - Klient jest zapisany do kolejki do fryzjera
    - Wyświetlamy zaktualizowane informacje
    - Budzi fryzjera (sekcja na czerwono) – sem_post()
    - Czeka na swoją kolej (sekcja na zielono) – sem wait
    - Kiedy przychodzi jego kolej to siada na fotel (setCurrentlyCutting = clientId)
    - Usuwamy klienta z kolejki do fryzjera
    - Wyświetlamy zaktualizowane informacje
- Jeżeli nie jest
    - Zapisujemy do na liście odrzuconych klientów
    - Dodajemy klienta do listy klientów, którzy nie dostali się do fryzjera i odeszli
    - Wyświetlamy zaktualizowane informacje