

Trabalho Prático 1: FSPD

Lucas Fonseca Mundim - 2015042134
2022/1

Introdução

Este trabalho prático de FSPD tem como objetivo exercitar conhecimentos de pthreads, mais especificamente do uso de `pthread_cond_t`, `pthread_t` e `pthread_mutex_t` para criar uma abstração de ThreadPool, exercitando o conteúdo visto até agora na disciplina de forma prática.

Detalhes da implementação

Para a execução do trabalho prático, foi necessário implementar uma `ThreadPool` e uma estrutura para manter as tarefas que seriam recebidas. As tarefas seriam executadas no modelo `FIFO`, portanto uma estrutura do tipo `Queue` era a ideal para a utilização. Foi definida uma utilização minimalista de ambas as estruturas tendo em vista que o escopo do projeto é bem reduzido. Abaixo é possível ver alguns trechos importantes:

- `ThreadPool`:

A screenshot of a code editor with a dark background and light-colored text. The code is in C++ and shows the initialization of a thread pool. It includes comments and function calls for creating threads. The code is as follows:

```
// cria threadpool
pthread_t threads[max_threads];
int j;
for(j = 0; j<min_threads; j++){
    thread_count++;
    pthread_create(&threads[j], &attr, &start_thread, NULL);
}
```

Como nunca existirão mais do que `max_threads` threads na `ThreadPool`, o tamanho máximo da coleção de threads foi definido como este valor, com `min_threads` threads já inicializadas para evitar overhead de inicialização. A `ThreadPool` sempre terá pelo menos essa quantidade de threads esperando tarefas.

- Queue:

```
void push(task_descr_t task){
    pthread_mutex_lock(&mutex_queue);
    task_queue[task_count] = task;
    task_count++;
    pthread_mutex_unlock(&mutex_queue);
    pthread_cond_signal(&cond_queue);
}

task_descr_t pop(){
    if(task_count == 0){
        task_descr_t empty = {
            .ms = -1,
            .pid = -1
        };
        return empty;
    }

    task_descr_t task = task_queue[0];

    for(int i = 0; i < task_count - 1; i++){
        task_queue[i] = task_queue[i + 1];
    }
    task_count--;
    pthread_mutex_unlock(&mutex_queue);

    return task;
}
```

Para a implementação da fila, foi definida uma estrutura extremamente simples utilizando **Array** para evitar a complicação de envolver mais **structs** no projeto. O mais importante era que as funções **push()** e **pop()** precisavam ser **thread safe**. Para isso foi utilizado um **pthread_mutex_t**

nomeado `mutex_queue`, impedindo acesso concorrente às seções críticas desses métodos. Também é importante chamar a atenção para o primeiro uso de `pthread_cond_t`, presente no método `push()`. Ao adicionar uma nova tarefa na fila, `pthread_cond_signal()` é executado para sinalizar à uma thread em espera.

Para a inicialização de threads, foi construído o método `start_thread()` que cuida de todo o processo necessário para o ciclo de vida daquela thread:

- Definição de **thread ID**:

```
pthread_mutex_lock(&mutex_tid);
int tid = curr_tid;
curr_tid++;
pthread_mutex_unlock(&mutex_tid);

// anuncia inicio da thread
printf("TB %d\n", tid);
```


Ao ser iniciada, a thread incrementa um contador **thread safe** para definir seu ID e anuncia sua criação.

- Início do loop de tarefas e espera pela primeira tarefa:

```
// espera novas tarefas
pthread_mutex_lock(&mutex_queue);
while(task_count == 0){
    pthread_cond_wait(&cond_queue, &mutex_queue);
}
```

Enquanto não houverem tarefas enfileiradas, a thread fica em espera com `pthread_cond_wait()`. Isso evita o consumo de CPU enquanto estiver *idle*.

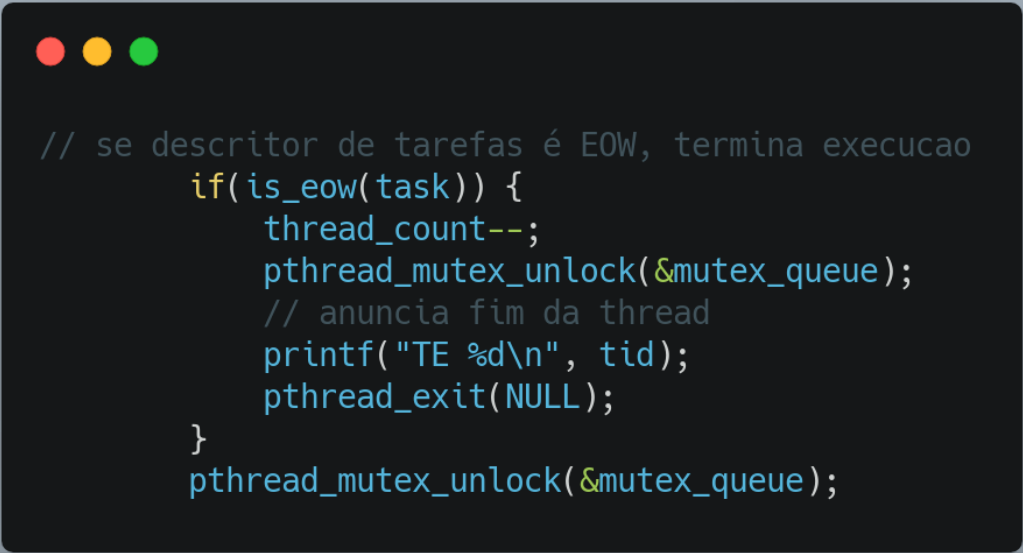
- Recuperação da definição de tarefa:



```
// retira descritor de tarefas
task_descr_t task;
task = pop();
```

Simplesmente é chamado o método `pop()` para recuperar o primeiro item da fila. Note que a falta de `lock` no método `pop()` é devido à sua existência no passo anterior.


- Verificação de **EOW**:



```
// se descritor de tarefas é EOW, termina execucao
if(is_eow(task)) {
    thread_count--;
    pthread_mutex_unlock(&mutex_queue);
    // anuncia fim da thread
    printf("TE %d\n", tid);
    pthread_exit(NULL);
}
pthread_mutex_unlock(&mutex_queue);
```


É feita a verificação de **EOW** na tarefa recuperada. Se for **EOW**, a thread anuncia seu encerramento e

o faz. EOW foi definido conforme sugerido:



```
task_descr_t eow = {  
    .ms = 0,  
    .pid = 0  
};
```

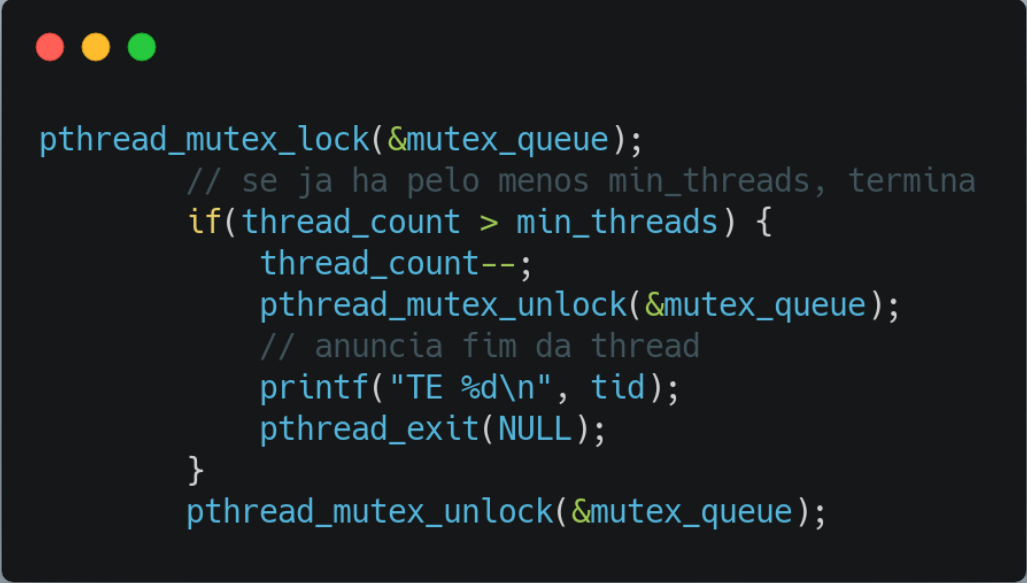
- Execução da tarefa:



```
if(task.pid > 0){  
    processa(&task);  
}
```

Se a tarefa for uma tarefa válida, ela é executada.

- Encerramento da thread caso necessário:



```
pthread_mutex_lock(&mutex_queue);  
    // se ja ha pelo menos min_threads, termina  
    if(thread_count > min_threads) {  
        thread_count--;  
        pthread_mutex_unlock(&mutex_queue);  
        // anuncia fim da thread  
        printf("TE %d\n", tid);  
        pthread_exit(NULL);  
    }  
    pthread_mutex_unlock(&mutex_queue);
```

Caso existam, além da thread corrente, pelo menos `min_threads`, a thread atual anuncia seu encerramento e o faz.

A criação de threads extras (entre `min_threads` e `max_threads`) se faz no ato da recepção de uma nova tarefa: caso seja criada uma nova tarefa e a quantidade de threads em espera é menor que o máximo, uma nova thread é criada.

Ao final, ao receber um `E0F` (`-1` no lugar de `pid`), a fila é preenchida com `max_threads E0W` que seguem por encerrar todas as threads restantes.

Conclusão

Algumas decisões de projeto com o fim de simplificar a implementação foram tomadas com o escopo do projeto, o que por fim foi uma boa decisão, deixando a implementação mais simples de ser depurada utilizando `lldb` e testada. A aplicação rudimentar de threads utilizando `pthread.h` é importante para entendermos o que outras bibliotecas mais robustas e modernas fazem "por baixo dos panos" e, de fato, compreender o que é paralelismo e o porquê das leis de Amdahl e Gustafsson.
