Trabalho Prático 3

Data Publicação: 21-04-2023 **Data Conclusão: 10-06-2023**

Entrega: A entrega da resolução é realizada na página da disciplina no Moodle juntando, num único ficheiro compactado, o **código fonte**, o **Makefile**. Existirão aulas práticas para a realização parcial deste trabalho com entrega na respetiva aula.

Objetivos: Familiarização com o ambiente UNIX/LINUX; Conceção de programas baseados no paradigma cliente/servidor utilizando *sockets* como mecanismo de comunicação entre processos; Conceção de programas concorrentes com base em múltiplas tarefas; Sincronismo entre múltiplas tarefas em POSIX; Utilização de sinais UNIX; Consolidação da programação ao nível de sistema.

Livro: A resolução deste trabalho pressupõe a utilização do livro R. Arpaci-Dusseau, A. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, August, 2018.

I. Realize os seguintes exercícios

Durante a realização dos exercícios propostos utilize o comando `man` no terminal (e.g. **`man fork`**) de forma a esclarecer dúvidas sobre as funções C, para obter informações sobre as chamadas de sistema, como por exemplo, os seus argumentos, os valores de retorno e como verificar as situações de erro.

Para a resolução de cada questão comece por criar uma pasta contendo os ficheiros C com a resolução do exercício e um ficheiro `Makefile` (Incluindo, entre outras, as regras **`all`**, e **`clean`**) que permita a compilação da solução do exercício. A compilação dos códigos na UC de SO é realizada com as flags `-Wall` e `-Werror`.

Na sua resolução deve considerar o tratamento de erros das chamadas de sistema utilizadas.

1. Num cenário onde múltiplas tarefas trabalham em conjunto pode ser necessário esperar que todas terminem de realizar um determinado conjunto de ações (cheguem a um determinado ponto de execução do seu código) antes de poderem prosseguir o seu processamento. Esta sincronização pode ser realizada através do mecanismo de barreira onde cada tarefa espera que todas as outras cheguem ao mesmo local. Após o grupo de tarefas estar reunido é permitido que todas continuem a sua execução.

   Assim, uma barreira é iniciada com o número de tarefas que devem encontrar-se na barreira. As tarefas realizam as suas ações e esperam na barreira até todas as tarefas chegarem a este mecanismo. Quando a última tarefa chegar, à barreira, todas as tarefas prosseguem a sua execução.

   Pretende-se que devolve este mecanismo de sincronização, **sem utilizar o `pthread_barrier_t` da biblioteca de `PThreads`**, e seguindo a seguinte API:

```
typedef struct {
    // a definir com os atributos e mecanismos de sincronismo
    // necessários à sua implementação
} sot_barrier_t;

int sot_barrier_init    (sot_barrier_t *barrier, int numberOfThreads);
int sot_barrier_destroy (sot_barrier_t *barrier);
int sot_barrier_wait    (sot_barrier_t *barrier);
```

2. Pretende-se uma função que determine o maior valor existente num vetor preenchendo, de seguida, todos os elementos do vetor com esse valor. Ilustra-se, de seguida, o comportamento desta função:

   Se consideramos o vetor `values` com os valores indicados:

   ```
   values[] = { 3, 4, 5, 78, 89, 6, 9, 1, 0 };
   ```

   depois de utilizada a função o vetor fica com a seguinte constituição:

   ```
   values[] = { 89, 89, 89, 89, 89, 89, 89, 89, 89 };
   ```

   Considerando que o vetor pode ter dimensões elevadas a função deve ser realizada em paralelo, com recurso a múltiplas tarefas, para otimizar o seu processamento. A função pretendida tem a seguinte assinatura:

   ```
   void find_larger_and_fill_parallel(int v[], int dim, int numberOfThreads);
   ```

   e realiza os seguintes passos:

   i. Divide o trabalho pelo número de tarefas indicado no argumento `nthreads`;
   ii. Cada tarefa processa a sua parte do vetor calculando o valor máximo (local) nessa partição;
   iii. Todas as tarefas colaboram na determinação do valor máximo global de todo o vetor;
   iv. Todas as tarefas têm de se sincronizar entre si garantindo que todo o vetor foi processado e o valor máximo foi determinado. Este sincronismo é realizado através de uma barreira (usar o mecanismo `sot_barrier_t` desenvolvido na questão anterior);
   v. De seguida cada tarefa preenche a sua parte do vetor com o valor máximo de todo o vetor e
   vi. Antes da função terminar espera que todas as tarefas terminem a sua execução.

3. Desenvolva o mecanismo de sincronismo `countdown_t` que permite que uma ou mais tarefas esperem que um conjunto de operações, que estão a ser realizadas por outras tarefas, terminem. O mecanismo é iniciado com um valor inteiro maior que zero. A função `countdown_wait()` bloqueia até que o valor do mecanismo chegue a zero através da evocação da primitiva `countdown_down()`. Depois do valor do mecanismo chegar a zero desbloqueia todas as tarefas em espera e as chamadas subsequentes ao `countdown_wait()` retorna de imediato (ver figura 1).

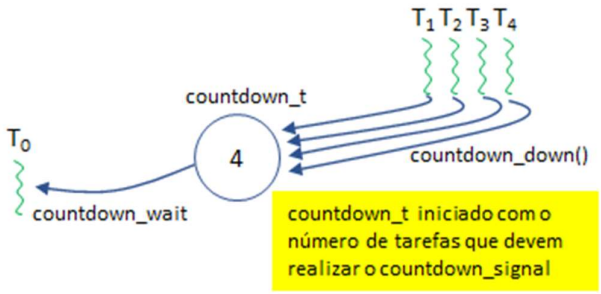

Figura 1 - Exemplificação do funcionamento do mecanismo `countdown_t`

A implementação deve respeitar a seguinte API:

```
typedef struct {
    // a definir com os atributos e mecanismos de sincronismo
    // necessários à sua implementação
} countdown_t;

int countdown_init    (countdown_t *cd, int initialValue);
int countdown_destroy (countdown_t *cd);
int countdown_wait    (countdown_t *cd);
int countdown_down    (countdown_t *cd);
```

II. Servidor de multiplicação de matrizes

4. As soluções de código concorrente estruturado com base em múltiplas ações, implica lidar, explicitamente, com a criação e terminação de tarefas para executarem cada uma dessas ações. A constante criação de novas tarefas, tem um custo associado, embora inferior à criação de novos processos, não é desprezável penalizando o desempenho. Por outro lado, a criação de tarefas por cada ação pode conduzir a um número excessivo de tarefas, que em simultâneo tentam executar-se, levando a taxa de ocupação dos processadores aos 100% e a um número elevado de troca de contexto entre essas tarefas.

   Uma alternativa possível baseia-se na utilização de um *thread pool*. Nesta abordagem, as ações são submetidas numa fila e associada a esta fila existe um conjunto de tarefas, previamente criadas, em que cada uma têm por objetivo retirar uma ação da fila, executá-la e voltar a ficar disponível para novas ações. As tarefas do *pool* não terminam e são reutilizadas na execução das diversas ações submetidas no *pool* (ver figura 2).

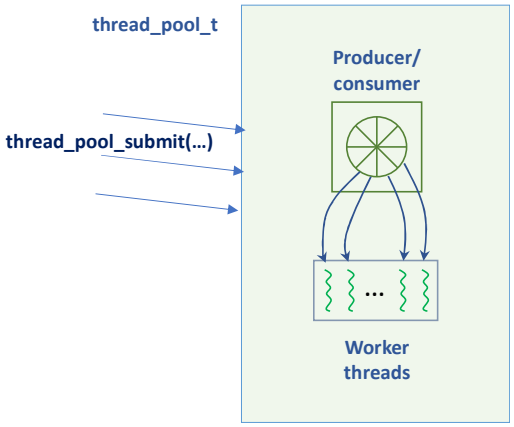

Figura 1 - Estrutura ilustrativa do *thread pool* a desenvolver

Desenvolva o seu *thread pool* seguindo a seguinte interface:

```
typedef void (wi_function_t)(void *);
typedef struct {
    // a definir com os atributos e mecanismos de sincronismo
    // necessários à sua implementação
} threadpool_t;

int threadpool_init    (threadpool_t *tp, int queueDim, int nthreads);
int threadpool_submit  (threadpool_t *tp, wi_func_t func, void *args);
int threadpool_destroy (threadpool_t *tp);
```

A função `threadpool_init()` inicia um *thread pool* com uma fila para trabalhos de dimensão `queueDim` e com um conjunto de *worker threads* indicado no argumento `nthreads`.

A função `threadpool_submit()` é utilizada para submeter ao *thread pool* um trabalho a realizar. O trabalho é definido através de uma função C que recebe um argumento do tipo ponteiro para `void`.

A função `threadpool_destroy()` deve ser chamada no fim da utilização do *thread pool* e deve considerar os seguintes pontos:

- O *pool* deve parar de aceitara submissão de novos trabalhos;
- Os trabalhos, previamente, submetidos devem ser todos executados;
- As tarefas de suporte do *pool* (*worker threads*) devem terminar de forma graciosa após todos os trabalhos terem sido executados;
- A função só retorna depois de todas as *worker threads* terem terminado.

Teste todas as funcionalidades do seu *thread pool* através de um programa de teste.

5. Realizou no trabalho passado a multiplicação de matrizes com base em múltiplas tarefas. Realize agora, uma nova versão, baseada no ***thread pool*** desenvolvido. Utilize o mecanismo **`countdown_t`**, desenvolvido na questão 1, para que a tarefa coordenadora se sincronize com o fim do processamento de todas as unidades de trabalho (*work items*). Compare as várias versões desenvolvidas. A função deve respeitar a seguinte assinatura:

```
Matrix * matrix_multiplication_with_pool_threads (Matrix *m1, Matrix *m2,
                                                  int nPartitions, threadpool_t *tp);
```

6. **O trabalho anterior terá uma classificação máxima de 15 valores**. Esta questão é opcional e serve para valorizar o seu trabalho. A resolução pode ser feita totalmente ou parcialmente, ficando a avaliação dependente do contributo adicional apresentado. Este ponto será avaliado com um máximo de 5 valores.

   Considere o servidor desenvolvido na última alínea do segundo trabalho prático (servidor que disponibiliza o serviço de multiplicação de matrizes). Este servidor aceita ligações dos clientes, tanto, através de *sockets* TCP, como, através de *sockets* UNIX. Reformule o servidor, desenvolvido, de forma que passe a incluir os seguintes pontos:

   a. Na abordagem anterior o servidor processava em concorrência os pedidos dos clientes criando uma tarefa por cada ligação. Estas tarefas terminavam após cessar a interação com um cliente. A terminação de todas estas tarefas implica a utilização da primitiva `phtread_join` para que os recursos das tarefas fossem eliminados. Por outro lado, se existisse um elevado número de ligações o servidor acabaria por criar demasiadas tarefas saturando o sistema. Finalmente, o servidor está sempre a despender tempo na criação e terminação de tarefas.

      De forma a reduzir o número de tarefas criadas e respetiva eliminação, passam a existir um conjunto fixo de tarefas (*Client Threads*) responsáveis pela interação com os clientes (*thread pool*). Estas tarefas obtêm a ligação com o cliente (descritor do *socket*) a partir de um *buffer* partilhado (Produtor/Consumidor). Após terminarem, o atendimento do cliente, regressam ao *buffer* partilhado esperando por uma nova ligação (as tarefas são reutilizadas para atenderem outro cliente). As tarefas responsáveis por aceitarem as ligações dos clientes (*Accept Threads*) colocam o descritor dos *sockets* no *buffer* partilhado (ver Figura 2).

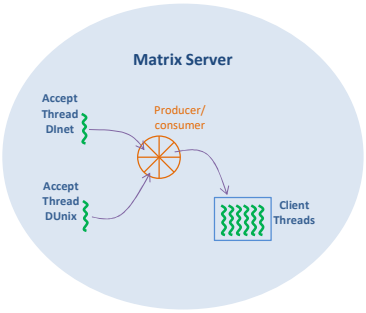

Figura 2

   b. Adicione ao servidor o registo de informação estatística relativa ao número de ligações recebidas, o total de operações realizadas e a dimensão média das matrizes.

   c. Adicione ao servidor uma tarefa **`printStatistics`** responsável pela apresentação periódica (na consola de segundo em segundo) da informação estatística mantida no servidor.

   d. **Adicione** ao servidor a possibilidade de desencadear a sua terminação quando for premida uma tecla (e.g. tecla '`T`'). Ao terminar, o servidor deve começar por não aceitar mais ligações, esperar que as ligações, anteriormente, aceites sejam todas processadas e depois terminar de forma ordeira todas as tarefas do servidor.

   e. A **multiplicação** das matrizes é realizada através de múltiplas tarefas (realizada no 2º trabalho) que terminam depois de concluírem uma operação. Adote a abordagem seguida na alínea a) (*thread pool*) para que o servidor possua um conjunto de tarefas reservadas para suportar todas as operações de multiplicação de matrizes.

Esta proposta de trabalho deixa em aberto algumas questões que constituem opções a serem tomadas pelos alunos. Os testes realizados na verificação da correção do trabalho constituem, também, um ponto de avaliação.

Bibliografia de suporte

1) Bibliografia de suporte disponível na página comum do Moodle:
   a) Slides utilizados nas aulas.
   b) Exemplos fornecidos.
   c) Exemplos realizados nas aulas.
2) R. Arpaci-Dusseau, A. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, august, 2018. Available: http://pages.cs.wisc.edu/~remzi/OSTEP/. [Accessed: 06-04-2022].

Bom trabalho,
Diogo Cardoso, *Nuno Oliveira*