Trabalho Prático 3

Data Publicação: 07-05-2024 **Data Conclusão: 09-06-2024**

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, November, 2023 (Version 1.10).

I. Realize os seguintes exercícios

Durante a realização dos exercícios propostos utilize o comando man no terminal (e.g. **`man pthread_mutex_init`**) 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.

1. Considere os códigos seguintes.
   a. Diga, justificando a sua resposta, se o programa da Figura 1 produz sempre o mesmo valor. Caso não produza apresente as modificações necessárias para o corrigir.

```c
void *th1 (void *arg) {
  int *pt = (int *)arg;
  for (int i=0 ; i<10000000; ++i)
    (*pt) += 2;
  return NULL;
}

void *th2 (void *arg) {
  int *pt = (int *)arg;
  for (int i=0 ; i<10000000; ++i)
    (*pt) -= 3;
  return NULL;
}

void *th3 (void *arg) {
  int *pt = (int *)arg;
  for (int i=0 ; i<10000000; ++i)
    (*pt)++;
  return NULL;
}
```

```c
int main() {
  int count = 0;
  pthread_t t1, t2, t3;

  pthread_create(&t1, NULL, th1, &count);
  pthread_create(&t2, NULL, th2, &count);
  pthread_create(&t3, NULL, th3, &count);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_join(t3, NULL);

  printf("Total = %d\n", count);
}
```

Figura 1

   b. Diga, justificando a sua resposta, se o programa da Figura 2 produz sempre o mesmo valor. Caso não produza apresente as modificações necessárias para o corrigir.

```c
void *th1 (void *arg) {
  int *pt = (int *)arg;
  for (int i=0 ; i<10000000; ++i)
    (*pt) += 2;
  return NULL;
}

void *th2 (void *arg) {
  int *pt = (int *)arg;
  for (int i=0 ; i<10000000; ++i)
    (*pt) -= 3;
  return NULL;
}

void *th3 (void *arg) {
  int *pt = (int *)arg;
  for (int i=0 ; i<10000000; ++i)
    (*pt)++;
  return NULL;
}
```

```c
int main() {
  int count = 0;
  pthread_t t1, t2, t3;
  int a = 0, b = 0, c = 0;

  pthread_create(&t1, NULL, th1, &a);
  pthread_create(&t2, NULL, th2, &b);
  pthread_create(&t3, NULL, th3, &c);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_join(t3, NULL);

  count = a + b + c;

  printf("Total = %d\n", count);
}
```

Figura 2

2. Considere a existência de um armazém automatizado onde vários robôs móveis realizam o carregamento dos camiões de transporte. Existe um local onde os robôs se dirigem para carregarem as suas baterias. O local de carregamento possui vários pontos de carga, por exemplo 10 (identificados de 0 a 9). Quando um robô necessita de carregar as suas baterias dirige-se ao local de carregamento e utiliza um dos pontos de carga ou espera até existir um livre. Considere que os robots são simulados por tarefas e que utilizam um gestor de acesso aos pontos de carga.

   a. Implemente uma solução para o gestor de acesso com as primitivas `reserveChargePoint` (que devolve o identificador do ponto de carga atribuído) e `freeChargePoint` (que liberta o ponto de carga atribuído pela primitiva anterior). Utilize os mecanismos de sincronismo que achar mais adequados.

   b. Considere a existência de robôs, que dado as suas funções, necessitam de obter o mais depressa possível um ponto de carga, preterindo, eventualmente, outros robôs já em espera. Adicione uma nova primitiva para a reserva prioritária de um ponto de carga (`reserveChargePointPriority`). Implemente esta nova versão do gestor de acesso.

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. As tarefas que usem a função `countdown_wait()` bloqueiam até que o valor do mecanismo chegue a zero. As tarefas que evocam a primitiva `countdown_down()` não bloqueiam e o valor do mecanismo é decrementado. Quando o valor do mecanismo chegar a zero todas as tarefas em espera são desbloqueadas e as chamadas subsequentes ao `countdown_wait()` retornam de imediato com a indicação de erro. A Figura 3 ilustra o comportamento do mecanismo.

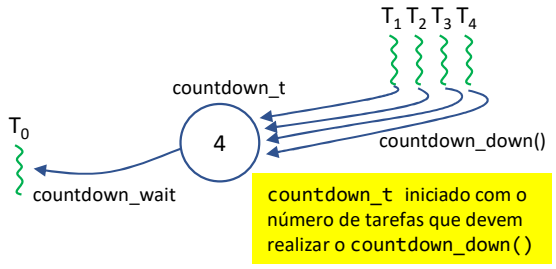

Figura 3 - Exemplificação do funcionamento do mecanismo `countdown_t`

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. Processamento estatístico de vetores

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 4).

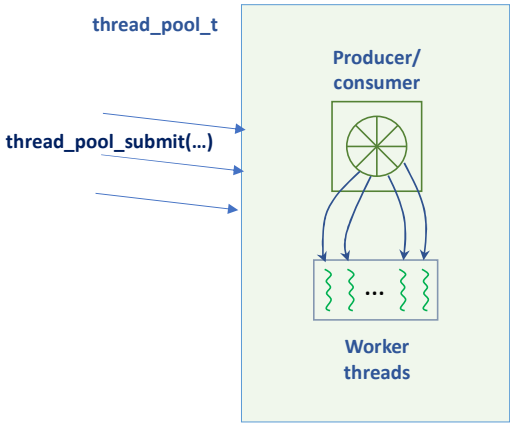

Figura 4 - Estrutura ilustrativa do *thread pool* a desenvolver

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_function_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 anterior uma função, com base em múltiplas tarefas, para extrair de um vetor (`array`) de números inteiros o subvetor com os elementos compreendidos num determinado intervalo de valores. Realize, uma nova versão baseada no ***thread pool*** desenvolvido na questão 4. Utilize o mecanismo **`countdown_t`**, desenvolvido na questão 3, 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:

```
int vector_get_in_range_with_thread_pool (int v[], int v_sz, int sv[],
                                          int min, int max, 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 (presta o serviço de estatística sobre vetores). 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.

      Altere o servidor de forma que o atendimento dos clientes seja realizado através do **thread pool** que implementou na questão 4. Desta forma passam a existir um conjunto fixo de tarefas (número de tarefas existentes no *thread pool*) responsáveis pela interação com os clientes.

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

   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 **operação sobre vetores** é realizada através de múltiplas tarefas (realizada no 2º trabalho) que terminam depois de concluírem uma operação. Adote a abordagem da alínea a) (*thread pool*) para que o servidor possua um conjunto de tarefas reservadas para suportar todas as operações sobre vetores.

   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.

III. Questões de escolha múltipla

1. Indique, para cada uma das afirmações, se a utilização de múltiplas tarefas, num programa a executar-se num sistema operativo UNIX, é uma razão válida.

Para ter a execução de dois troços de código concorrentes com espaços de endereçamento separados num mesmo processo.	
Para poder realizar operações I/O em simultâneo com outras operações num mesmo processo.	
Para poder executar dois programas (ficheiros executáveis) diferentes em concorrência e de uma forma mais rápida.	
Para dividir o processamento por ações concorrentes de forma a maximizar a utilização de toda a capacidade de processamento do *hardware*	

2. Considere que a função `handleClient()` processa uma ligação com o cliente indique se as afirmações seguintes são verdadeiras ou falsas

`` int main () { int s = tcp_socket_server_init(PORT); while (1) { int ns = tcp_socket_server_accept(s); handle_client(ns); } return 0; } ```	Servidor disponível através de um *socket* no domínio internet atendendo múltiplos clientes em concorrência.	
	Servidor disponível através de um *socket* no domínio internet atendendo múltiplos clientes em sequência.	
`` void `thHandleCient (void *arg) { int ns = *((int *)arg); handle_client(ns); return NULL; } int main () { int s = tcp_socket_server_init(PORT); pthread_t th; while (1) { int ns = tcp_socket_server_accept(s); int *ps = malloc(sizeof(int)); *ps = ns; pthread_create(&th, NULL, thHandleClient, ps); pthread_detach(th); } return 0; } ```	Servidor disponível através de um *socket* no domínio internet atendendo múltiplos clientes em concorrência.	
	Servidor disponível através de um *socket* no domínio internet atendendo múltiplos clientes em sequência.	

Bibliografia de suporte

1) Bibliografia de suporte disponível na página comum do Moodle:
   a) Slides utilizados nas aulas; Exemplos fornecidos; Exemplos realizados nas aulas.
2) R. Arpaci-Dusseau, A. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, November, 2023 (Vers 1.10) [Ch 1 - 6]. Available: http://pages.cs.wisc.edu/~remzi/OSTEP/. [Accessed: 19-02-2024].

Bom trabalho,
Diogo Cardoso, *Nuno Oliveira*