

Ficha 4 - Threads

Extraia o conteúdo do ficheiro `ficha-threads-ficheiros.zip` para um diretório de trabalho à sua escolha. Execute o comando `make` para criar os ficheiros executáveis.

I

Analise o programa contido em `ex1.c`. A função `myprint` simula a escrita num dispositivo “lento”. Esta função foi implementada de forma a realçar o risco de conflitos no acesso concorrente a recursos partilhados; no entanto, mesmo com as funções normais de escrita, este tipo de problemas também poderão ser verificados, embora de forma menos evidente.

1 - Considere o seguinte extrato de código do ficheiro `ex1.c`:

```
int main()
{
    char *buf;
    pthread_t tid1, tid2;

    buf = (char *) malloc(256);
    sprintf(buf, "%d: Ola ++++++\n", getpid());

    /*
    Irá ser criada uma nova thread e executada a função mythread(buf) nessa thread. A variável buf
    contém o endereço do bloco de memória onde é guardada a string.
    */
    pthread_create(&tid1, NULL, mythread, buf);

    //...

void *mythread(void *arg) {
    char *str1 = (char *) arg;
    while(1) {
        sleep(1);

        myprint(str1);
    }
}
```

1.1 - Execute o programa. Justifique o comportamento observado (*strings* “misturadas”) com base no descrito acima.

1.2 Qual o objetivo da instrução `pthread_join(tid1, NULL)`?

1.3 -Tendo em conta as relações ilustradas, justifique a necessidade de fazer duas chamadas à função `malloc` no código em `ex1.c`. O que aconteceria caso se reutilizasse o bloco de memória alocado inicialmente?

II

1 - Analise o programa fornecido no ficheiro `par.c` (também fornecido na última página da ficha).

1.1 - Teste o programa e registe os seus tempos de execução. Altere o valor `N_ITER` de forma que o tempo de processamento seja de **aproximadamente** 120 segundos. Não perca demasiado tempo na afinação do tempo (um erro de alguns segundos é perfeitamente aceitável). Deverá manter o mesmo valor de `N_ITER` nas alíneas seguintes.

1.2 - Altere o programa apresentado de forma a dividir o processamento dos dados (contidos no bloco de memória apontado pela variável `dados`) por duas *threads*, que deverão ser executadas concorrentemente. Na primeira *thread*, deverá ser aplicada a função `operacao_muito_demorada` à primeira metade do vetor; na segunda *thread*, deverá ser aplicada a mesma função à segunda metade do vetor. Note que

```
operacao_muito_demorada(dados, NUM_ELEM);
```

é equivalente a

```
operacao_muito_demorada(dados, NUM_ELEM/2);
```

```
operacao_muito_demorada(dados+NUM_ELEM/2, NUM_ELEM/2);
```

O programa só deverá terminar quando ambas as *threads* terminarem. Para implementar este comportamento, deverá utilizar a função `pthread_join`.

Não altere o código correspondente às medições dos tempos de execução. Teste o seu programa e comente os resultados. Garanta que a impressão obtida é a mesma que obteve com a versão inicial do programa.

Notas:

- Não deverá alterar a função `operacao_muito_demorada`; assuma que não tem acesso ao código da função.
- Sugestão de resolução:
Implemente uma função, a ser usada como função principal de cada *thread*, que deverá receber a seguinte informação:
a) um apontador para o primeiro elemento da parte relevante do vetor `dados`;
b) um inteiro com o número de elementos a ser processado pela *thread*.
Uma vez que a função `pthread_create` apenas consegue lidar com funções com um único parâmetro de entrada, deverá definir o seguinte tipo de dados:

```
typedef struct {  
    double *ptr;  
    int n;  
} targs_t;
```

Por sua vez, esta função deverá chamar a função `operacao_muito_demorada`, passando-lhe os argumentos recebidos.

1.3 - Repita o procedimento da alínea anterior, mas desta vez dividindo o processamento do vetor por 50 *threads*. Comente os resultados.