Ficha 3 – Posix *threads*

2025

{patricio.domingues, vitor.carreira, miguel.frade, rui.ferreira, nuno.costa, gustavo.reis, carlos.machado, leonel.santos, miguel.negrao, carlos.grilo}@ipleiria.pt

1 Introdução

A criação de processos (através da primitiva `fork`) é uma forma de obter execução concorrente de uma determinada tarefa. No entanto, do ponto de vista do sistema operativo, os processos apresentam algumas limitações:

- A criação de um processo é dispendiosa porque todo o contexto do processo pai é herdado pelo processo filho (a memória e os descritores do processo pai são copiados para o processo filho);
- É necessário usar comunicação entre processos (Inter Process Communication - IPC - na designação anglo-saxónica) para passar informação entre dois ou mais processos.

As *threads* são fluxos de execução concorrentes dentro de um processo e pretendem solucionar os problemas referidos. A criação de uma *thread* pode ser 10 a 100 vezes mais rápida[1] que a criação de um processo, sendo por vezes apelidadas de processos leves. Além disso, todas as *threads* dentro de um processo partilham a mesma memória, o que simplifica a partilha de informação entre elas. No entanto, esta simplificação não elimina o problema da sincronização (matéria abordada na Ficha 4).

Plataforma	**fork()**	**pthread_create()**
IBM 332 MHz 604e 4 CPUs/node 512 MB Memory AIX 4.3	92,4	8,7
IBM 375 MHz POWER3 16 CPUs/node 16 GB Memory AIX 5.1	173,6	9,6
INTEL 2.2 GHz Xeon 2 CPU/node 2 GB Memory RedHat Linux 7.3	17,4	5,9

Tabela 1: Tabela comparativa do tempo real (em segundos) entre a criação de 50 mil processos e 50 mil *threads* (retirado de [2])

Todos os processos possuem pelo menos uma *thread*, designada por *main thread*, que pode executar o mesmo código que qualquer outra *thread*. No entanto, esta *thread* é especial porque se comporta como um processo, ou seja, quando acaba a sua execução, termina o processo sem permitir que as outras *threads* cheguem ao fim. Dentro de um processo, todas as *threads* partilham:

- As instruções a executar (segmento de código);
- Os dados na memória virtual do processo;
- Os descritores;
- Funções para tratamento de sinais (*signal handlers*);
- A diretoria corrente;
- Os IDs de utilizador e de grupo.

Mas cada *thread* possui individualmente:

- `Thread ID` (TID) – identificador da *thread* (diferente do identificador do processo - `pid`);
- Conjunto de registos (incluindo o contador do programa e da pilha);
- Pilha – estrutura que armazena, entre outros dados, as variáveis locais, os parâmetros de entrada de uma função e o valor de retorno;
- `errno` – variável onde é armazenado o código de erro resultante de uma chamada ao sistema;
- Conjunto de sinais pendentes ou bloqueados;
- Prioridade – a prioridade de uma *thread* é relativa às restantes *threads* do processo.
- Variáveis de tipo Thread-Local Storage (TLS).

A Figura 1 ilustra as diferenças entre processos e *threads* no que diz respeito ao conjunto de registos e pilha.

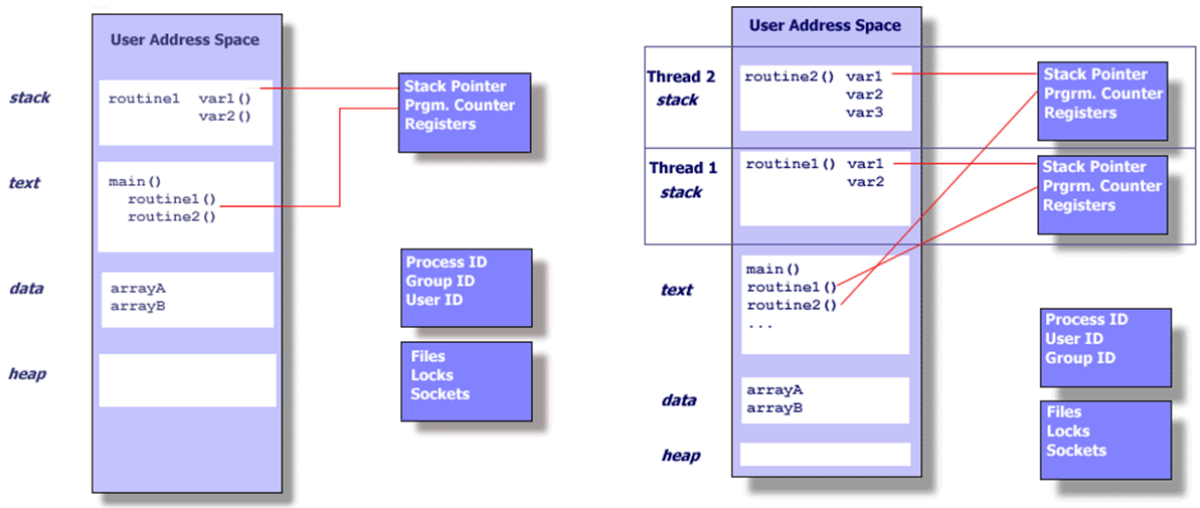

Figure 1: Divisão do espaço de endereçamento de um processo (esquerda) e um processo com duas *threads* (direita) [2]

Na próxima secção desta ficha é abordada a API definida na norma `POSIX 1003.1c` das *threads*, também designada por `pthreads`. Note que, apesar do *standard* C11 já especificar uma API para *threads*, iremos utilizar a API definida na norma POSIX.

> ***NOTA***
>
> *Nas distribuições Ubuntu as páginas do sistema man referentes às `pthreads` encontram-se disponíveis através dos módulos `manpages-posix` e `manpages-posix-dev`. Caso não estejam instalados, a instalação desses módulos pode ser realizada da seguinte forma:*

```
user@ubuntu $ sudo apt-get install manpages-posix manpages-posix-dev
```

1.1 *Threads* em Linux

Através do comando `ps` é possível listar também os dados associados às *threads* existentes num sistema Linux. Assim, a execução de `ps -eLf` acrescenta os campos `lwp` (ID da *thread*) e `nlwp` (número de *threads* do processo) à listagem. Na Figura 2 pode ver-se um exemplo dessa listagem, onde é possível verificar que, associado ao processo `init` com o PID 1, existe também associada uma *thread* com o TID 1.

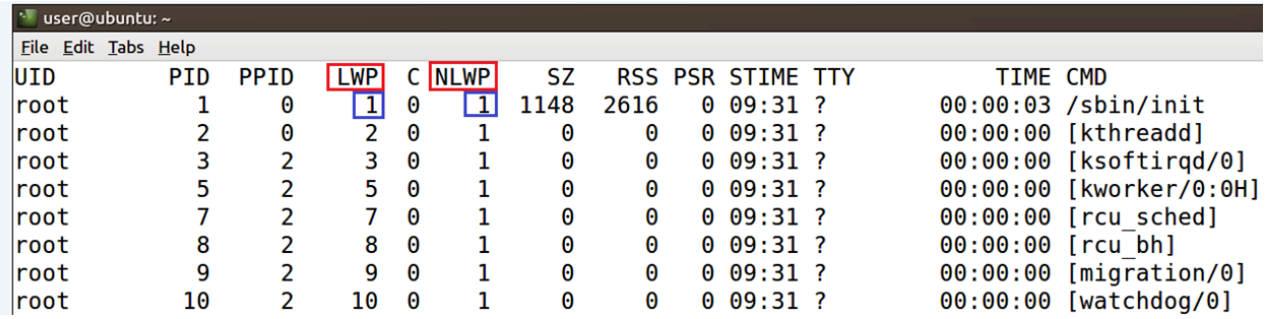

Figure 2: Saída parcial do comando **ps -eLf**

Na Figura 3 é mostrada uma listagem do mesmo comando, mas devidamente filtrada para que apenas apareçam processos `lxterminal`. Pode ver-se que o processo com PID 2513 tem 3 *threads* criadas.

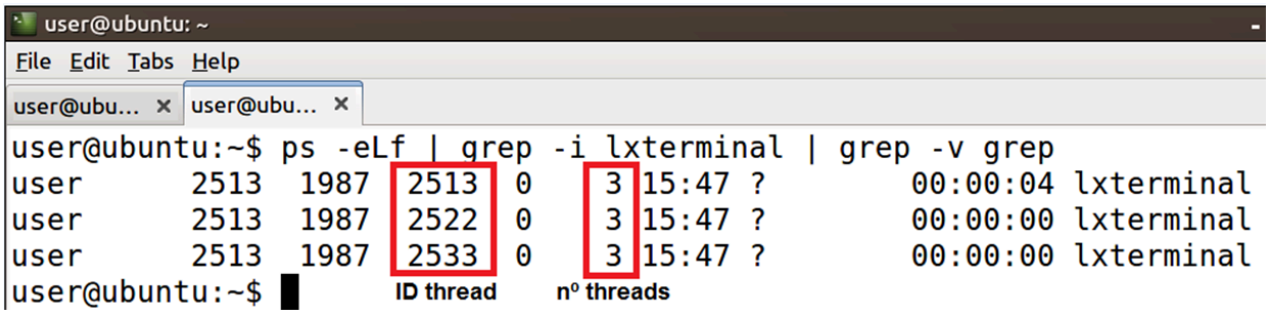

Figure 3: Saída parcial do comando **ps –eLf** com identificação da coluna `lwp` (ID *thread*) e `nlwp` (nº de *threads*)

1.1.1 Lab 1

Utilize a linha de comandos para obter o número de *threads* que um determinado processo tem associado. Por exemplo, se usarmos a informação da Figura 2 e quiséssemos obter a informação pretendida para o processo `init` (ou `systemd`), o output deveria ser “1”.

1.1.2 Lab 2

Utilizando o projeto fornecido, os conhecimentos da ficha anterior e a linha de comandos do Lab 1, elabore um programa que mostre o número de *threads* associadas ao processo criado pela execução do programa.

Execução do programa e respetivo output:

```
user@ubuntu $ ./lab2
Nome do programa: ./lab2
PID do processo atual: 5508
Numero de threads do processo atual: 1
```

2 Principais funções da biblioteca POSIX *threads*

Uma *thread* é um fluxo de execução dentro de um processo. Sempre que é criado um processo, é criada automaticamente uma *thread* (a *main thread*). Quando um processo termina, as *threads* associadas ao processo são terminadas.

2.1 Criar e executar uma thread

Para criar *threads* adicionais usamos a função `pthread_create`.

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attributes, void *(*start_routine)(void
        *), void *arg);
```

A função `pthread_create` permite criar uma nova *thread* que irá executar a função passada no terceiro parâmetro. A função recebe os seguintes quatro parâmetros:

- `thread` – ponteiro onde será armazenado o identificador único da *thread*;
- `attributes` – estrutura com os atributos da *thread* a criar (prioridade, tamanho inicial da pilha, etc.). Caso se pretenda utilizar os valores pré-definidos, passa-se como argumento o valor `NULL`. Para obter uma lista de atributos consulte a página do manual: `man pthread_attr_init`.
- `start_routine` – ponteiro para a função que a *thread* irá executar. A função a executar deve possuir o seguinte protótipo: `void * function(void *);`
- `arg` – ponteiro para o argumento passado à função `start_routine`. A função que a *thread* irá executar recebe um único argumento. Caso pretenda passar múltiplos argumentos deve utilizar uma estrutura.

Valores de retorno

Sucesso – devolve o valor zero.

Insucesso – devolve um valor positivo contendo o código de erro que seria atribuído à variável `errno`. No entanto, o valor não é atribuído à variável `errno`, apesar desta variável ser *thread-safe*. Este comportamento aplica-se, salvo exceções indicadas, a todas as funções da biblioteca `POSIX threads`.

> ***NOTA***
>
> *Uma vez que as funções da biblioteca `POSIX threads` não atribuem o código de erro à variável `errno`, a utilização das macros `ERROR` ou `WARNING` para tratamento de erros torna-se inapropriada, dado que a função associada a esta macro consulta o valor da variável de erro `errno` para imprimir o erro que ocorreu. Caso atribua o resultado à variável `errno`, pode continuar a utilizar as macros `ERROR` e `WARNING`.*

2.2 Identificador de uma thread

Cada *thread* possui um identificador único dentro de um processo (ID) atribuído quando a *thread* é criada (função `pthread_create`). Uma *thread* pode consultar o seu ID através da função `pthread_self`.

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Valores de retorno

Sucesso – devolve o identificador (ID) da *thread* atual.

Insucesso – não se aplica.

> ***NOTA***
>
> *O identificador devolvido por `pthread_self` não corresponde necessáriamente ao TID do processo, que pode ser obtido com `gettid`. O ID devolvido por `pthread_self` é único para cada thread apenas dentro de um dado processo. Para mais informação, consultar as páginas de manual pthread_self(3) e gettid(2).*

Exemplo

Listing 1: Criação de uma thread

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <pthread.h>
#include "debug.h"

#define C_ERROR_PTHREAD_CREATE 1

void *hello(void *arg);

int main(int argc, char *argv[]) {
    (void)argc;
    (void)argv;

    pthread_t tid;

    if ((errno = pthread_create(&tid, NULL, hello, NULL)) != 0) {
        ERROR(C_ERROR_PTHREAD_CREATE, "pthread_create() failed!");
    }

    exit(0);
}

void *hello(void *arg) {
    (void)arg;
    printf("My name is Thread, Posix Thread = [%lu]\n",
           (unsigned long)pthread_self());
    return NULL;
}
```

2.3 Término de uma thread

Uma *thread* pode terminar de três formas:

- Implicitamente – quando a função `start_routine` executada pela *thread* chega ao fim (quando faz `return`);
- Abruptamente – quando uma das *threads* do processo chama a função `exit`. Neste caso, para além de terminar o processo, terminam também todas as *threads* associadas ao mesmo;
- Explicitamente – utilizando a função `pthread_exit`.

2.3.1 Lab 3

Compile o programa do Lab 3, usando um ficheiro `makefile`.

> ***NOTA***
>
> *Use os ficheiros fornecidos no Lab 3 e complete o makefile devidamente. Execute o programa e explique a saída, associando-a a uma das três opções anteriores.*

2.3.2 Função pthread_exit

Apesar de a forma preferencial de terminar uma *thread* ser através de um `return NULL`, existe uma função (`pthread_exit`) que também permite implementar essa funcionalidade.

```
#include <pthread.h>
void pthread_exit(void *retval);
```

A função `pthread_exit` termina a execução da *thread* atual. A função recebe apenas um parâmetro:

- `retval` – ponteiro para o valor de retorno (*status*) da *thread*. Não é válido devolver o endereço de uma variável local. Caso não se pretenda devolver um valor, deve utilizar-se o argumento `NULL`.

Exemplo

Listing 2: Término explícito de uma thread

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <errno.h>
#include "debug.h"

#define C_ERROR_PTHREAD_CREATE 1

void *hello(void *arg);

int main(int argc, char *argv[]) {
    (void)argc;
    (void)argv;

    pthread_t tid;

    if ((errno = pthread_create(&tid, NULL, hello, NULL)) != 0) {
        ERROR(C_ERROR_PTHREAD_CREATE, "pthread_create() failed!");
    }

    exit(0);
}

void *hello(void *arg) {
    (void)arg;

    printf("My name is Thread, Posix Thread = [%lu]\n",
           (unsigned long)pthread_self());

    pthread_exit(NULL);

    /* Boas práticas de programação */
    return NULL;  // Forma preferencial para terminar a thread
}
```

2.4 Pontos de junção

As listagens anteriores, quando executadas, não produzem qualquer resultado. Este comportamento resulta do facto do processo que as criou terminar imediatamente antes das *threads* criadas serem executadas. É importante não esquecer que todas as *threads* criadas por um processo (*main thread*) terminam quando este termina. Utilizando a função `pthread_join` é possível implementar pontos de junção na execução concorrente de várias *threads*. Com esta função, uma *thread* pode esperar pelo término de uma outra *thread* (semelhante ao `waitpid` nos processos). No entanto, ao contrário dos processos, não existe nenhuma forma de esperar por uma qualquer *thread*. como acontece nos processos com as funções `wait` e `waitpid(-1,…)`.

2.4.1 Função pthread_join

```
#include <pthread.h>
int pthread_join(pthread_t th, void **return_ptr);
```

A função `pthread_join` bloqueia a *thread* atual até que a *thread* identificada pelo parâmetro `th` termine a sua execução. Parâmetros recebidos:

- `th` – identificador da *thread* pela qual se vai esperar;
- `return_ptr` – endereço do ponteiro que irá armazenar o valor devolvido (*status*) pela *thread* `th` (consultar função `pthread_exit`). Caso não se pretenda armazenar o valor devolvido, utiliza-se o argumento `NULL`.

Valores de retorno

Sucesso – devolve o valor zero.

Insucesso – devolve um valor positivo contendo o código de erro que seria atribuído à variável `errno`.

Exemplo

Listing 3: Junção de threads

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <pthread.h>
#include "debug.h"

#define C_ERROR_PTHREAD_CREATE 1
#define C_ERROR_PTHREAD_JOIN 2

void *hello(void *arg);

int main(int argc, char *argv[]) {
    (void)argc;
    (void)argv;

    pthread_t tid;

    if ((errno = pthread_create(&tid, NULL, hello, NULL)) != 0) {
        ERROR(C_ERROR_PTHREAD_CREATE, "pthread_create() failed!");
    }

    printf("Created thread's TID = [%lu]\n", (unsigned long)tid);

    if ((errno = pthread_join(tid, NULL)) != 0) {
        ERROR(C_ERROR_PTHREAD_JOIN, "pthread_join() failed!");
    }

    exit(0);
}

void *hello(void *arg) {
    (void)arg;
    printf("My name is Thread, Posix Thread = [%lu]\n",
           (unsigned long)pthread_self());
    return NULL;
}
```

2.4.2 Lab 4

Compile e execute o programa do Lab 4 e explique a sua saída. Explique o modo de funcionamento do programa relativamente às três opções do término de *threads*.

2.4.3 Função pthread_detach

O sistema operativo mantém uma estrutura de dados de controlo por cada *thread* criada. A estrutura possui, entre outros dados, informação relativa ao estado de execução da *thread* (em execução, suspensa, terminada, etc.) e o valor devolvido pela *thread* aquando do seu término. Por omissão, essa estrutura permanece em memória até que a função `pthread_join` seja chamada por outra *thread*. No entanto, podemos alterar este comportamento através da função `pthread_detach`. Neste caso, assim que uma *thread* chega ao fim, a estrutura de controlo que lhe estava atribuída é libertada.

```
#include <pthread.h>
int pthread_detach(pthread_t th);
```

A função recebe um único parâmetro:

- `th` – identificador da *thread* pela qual não se pretende esperar.

Valores de retorno

Sucesso – devolve o valor zero;

Insucesso – devolve um valor positivo contendo o código de erro que seria atribuído à variável `errno`.

2.5 Exemplo da utilização de *threads*

O programa abaixo utiliza *threads* para processar, de forma concorrente, o cálculo do fatorial de um vetor de inteiros.
São usados dois vetores:

- `valores[MAX]` – para armazenar os valores a calcular;
- `resultados[MAX]` – para armazenar os resultados do cálculo do fatorial.

Cada *thread* recebe um ponteiro para a posição (índice) dos vetores que tem que processar, através de uma estrutura (porque se está a passar mais do que um parâmetro), calculando o fatorial respetivo.

Listing 4: Cálculo do fatorial de uma sequência de números recorrendo a threads

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <pthread.h>
#include <sys/time.h>
#include "debug.h"

#define C_ERROR_PTHREAD_CREATE 1
#define C_ERROR_PTHREAD_JOIN 2
#define MAX 5

void *fatorial(void *arg);

struct ThreadParams {
    unsigned long valor;
    unsigned long resultado;
};

int main(void) {
    struct ThreadParams threadParams[MAX];

    // inicializar números para cálculo do fatorial (de 1 a 5)
    // "configurar" os parâmetros a passar às threads
    for (int i = 0; i < MAX; i++) {
        threadParams[i].valor = i + 1;
    }

    struct timeval tIni, tEnd;
    // inicia contagem de tempo
    printf("A efetuar cálculos...\n");
    gettimeofday(&tIni, NULL);

    // criar as threads para cálculo e passar "o(s) parâmetro(s)"

    pthread_t tids[MAX];

    for (int i = 0; i < MAX; i++) {
        if ((errno = pthread_create(&tids[i], NULL, fatorial,
                                    &threadParams[i])) != 0) {
            ERROR(C_ERROR_PTHREAD_CREATE, "pthread_create() failed!");
        }
    }

    // esperar que todas as threads terminem para mostrar resultados
    for (int i = 0; i < MAX; i++) {
        if ((errno = pthread_join(tids[i], NULL)) != 0) {
            ERROR(C_ERROR_PTHREAD_JOIN, "pthread_join() failed!\n");
        }
    }

    // termina contagem de tempo
    gettimeofday(&tEnd, NULL);

    // mostrar resultado da execução das threads
    printf("Tempo: %d segundos\n", (int)(tEnd.tv_sec - tIni.tv_sec));
    printf("Resultados:\n");
```

```
60      for (int i = 0; i < MAX; i++) {
61          printf("\t%lu!: %lu\n", threadParams[i].valor,
62                 threadParams[i].resultado);
63      }
64
65      exit(0);
66  }
67
68  void *fatorial(void *arg) {
69      // cast para o tipo de dados original
70      struct ThreadParams *params = (struct ThreadParams *)arg;
71
72      // obter valor para cálculo do fatorial
73      unsigned long valor = params->valor;
74
75      // calcular fatorial
76      unsigned long aux = valor;
77      while (--valor > 1) {
78          aux *= valor;
79      }
80
81      // guardar valor calculado no vetor de resultados
82      params->resultado = aux;
83
84      // esperar um pouco (apenas para pergunta do próximo Lab)
85      sleep(params->valor);
86
87      return NULL;
88  }
```

2.5.1 Lab 5

Compile e execute o programa do Lab 5 e explique a sua saída, principalmente em relação ao tempo que irá aparecer como sendo o tempo que todas as *threads* demoraram a terminar a sua tarefa.

2.5.2 Lab 6

O que aconteceria caso a função `pthread_join` fosse colocada no segundo ciclo `for`, logo após a função `pthread_create`? Verifique o tempo que iria ser utilizado para realizar o mesmo cálculo e retire as respetivas conclusões.

3 Exercícios

Nota: na resolução de cada exercício deve utilizar o template de exercícios que inclui uma `makefile` bem como todas as dependências necessárias à compilação.

3.1 Para a aula

1. Escreva um programa que deve criar três processos filho, sendo que cada processo filho deve criar duas *threads*. As *threads* devem escrever o PID do processo pai, o PID do seu próprio processo e o seu TID.

2. Escreva um programa que crie N *threads* que devem somar uma determinada quantidade a uma variável partilhada (global) não protegida, originando assim uma *race condition*. A soma deve ser realizada através de um ciclo onde se deve incrementar o valor unitariamente até perfazer o valor pretendido. Para forçar a

ocorrência deste tipo de problemas use a função `sched_yield()` no ciclo de incremento.

> ***NOTA***
>
> *Ambos os parâmetros, número de threads e quantidade a somar, devem ser enviados da linha de comandos, usando para isso a ferramenta* `gengetopt`*,*

Exemplo da saída de três execuções do programa que cria 15 *threads*, em que cada *thread* faz 20 incrementos unitários à variável partilhada:

```
G_shared_counter = 135 (expecting 300)
G_shared_counter = 126 (expecting 300)
G_shared_counter = 122 (expecting 300)
```

3.2 Exercícios extra-aula

3. Escreva um programa que crie duas *threads* para processar as diretorias existentes na diretoria `/proc`. Uma das *threads* deve processar as diretorias cujo nome represente um número par, ao passo que a outra *thread* processa as diretorias cujo nome represente um número ímpar. Cada *thread* deve mostrar mostrar o nome da diretoria e se este é par ou ímpar.

Escreva uma única função, que deve ser usada pelas duas *threads*, passe-lhe os parâmetros adequados e utilize as funções `opendir`, `readdir`, `closedir` e `stat` para obter a informação existente na diretoria `/proc`. De seguida, mostra-se um exemplo do output do programa sem (à esquerda) e com (à direita) utilização adequada da função `sched_yield()` [2]:

```
thread IMPAR: 1
thread IMPAR: 3
…
thread IMPAR: 16091
thread IMPAR: 16139
…
thread PAR: 2
thread PAR: 8
…
thread PAR: 16054
thread PAR: 16088
```

```
thread IMPAR: 1
thread PAR: 2
thread PAR: 8
thread IMPAR: 3
…
thread IMPAR: 16091
thread PAR: 16054
thread IMPAR: 16139
thread PAR: 16088
```

---

4. Escreva um programa que crie, preencha (com valores aleatórios entre 0 e 9) e calcule a multiplicação de duas matrizes de tamanho 5x5. Como o cálculo de cada elemento da matriz resultante não depende do cálculo dos outros, recorra a uma *thread* para calcular cada elemento da matriz resultante. Por fim, imprima o resultado. Segue-se um exemplo da multiplicação de duas matrizes 2x2:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}$$

5. Escreva um programa que, recorrendo à utilização de *threads*, ordene o conteúdo de um vetor de inteiros utilizando o algoritmo QuickSort. O tamanho do vetor a ordenar é definido como argumento de entrada (use o `gengetopt` com a opção **–n**) e o seu conteúdo deve ser preenchido aleatoriamente.

4 Bibliografia

[1] D. Butenhof, “Programming with POSIX threads”, Addison-Wesley.

[2] https://computing.llnl.gov/tutorials/pthreads/

[3] http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixthreads.html

[4] K. Robbins and S. Robbins, “Unix Systems Programming”, Prentice-Hall (capítulos 12 e 13).

---

1. As primeiras implementações de *threads* (na biblioteca `pthread`) para Linux, criam um processo por cada *thread*, embora a memória seja partilhada. Neste caso, a vantagem da rapidez na criação das *threads* não existe. No entanto, em versões da *kernel* superiores à 2.6, este comportamento foi alterado permitindo ao programador tirar partido do aumento de desempenho. A biblioteca de *threads* usada nas atuais distribuições de Linux recentes é a `NPTL` (`Native Posix thread Library`) que se encontra integrada na biblioteca do C (glibc).↩

2. `sched_yield()` causes the calling thread to relinquish the CPU. The thread is moved to the end of the queue for its static priority and a new thread gets to run.↩