Ficha 3 – Posix Threads

Tópicos abordados:
– Criação de *threads*
– Identificadores de *threads*
– Término e ponto de junção de *threads*

Duração prevista: 1 aula

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

[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* -- https://sourceware.org/git/?p=glibc.git;a=tree;f=nptl;hb=HEAD).

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 *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);
- A "maioria" dos dados (alguns apontadores de controlo, tais como o *program counter* – PC – não são partilhados);
- Os descritores;
- Funções para tratamento de sinais (*signal handlers*);
- A diretoria corrente;
- Os ID's 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.

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

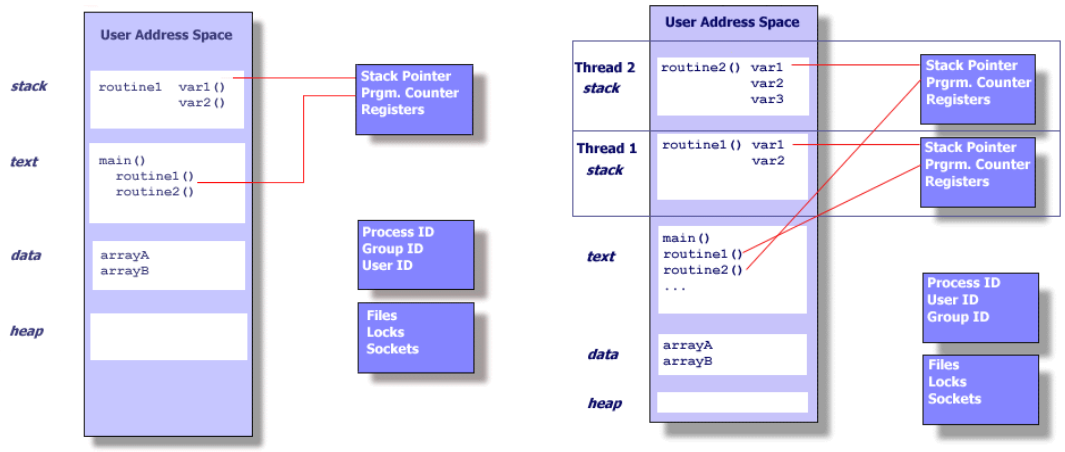

Figura 1 – A divisão do espaço de endereçamento de um processo (esquerda) e um processo com duas threads (direita) [2]

Na próxima secção desta ficha é abordada a API definida na norma POSIX 1003.1c das *threads*, também designada por *pthreads*. Note que o *standard* C11 já especifica uma API para *threads*, mas esta ainda não é atualmente suportada pela GLIBC e, consequentemente, pelo GCC.

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 feita da seguinte forma:

```
sudo apt-get install manpages-posix manpages-posix-dev
```

1.1. Threads no 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* (*systemd*) com o PID 1, existe também associada uma *thread* com o TID 1.

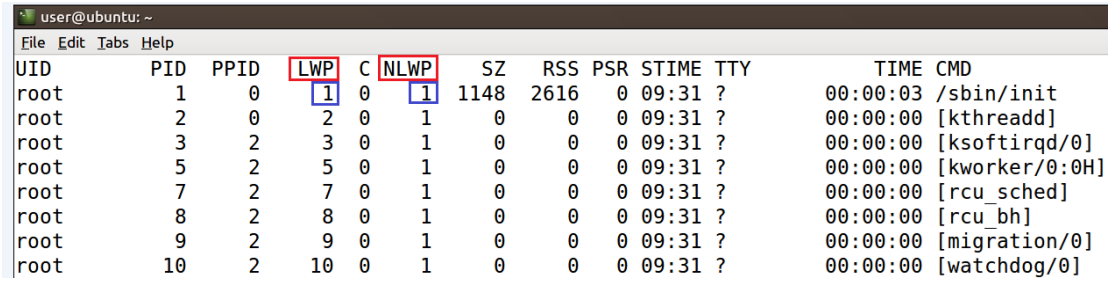

Figura 2: saída (parcial) do comando ***ps -eLf***

Na Figura 3 é mostrada uma listagem do mesmo comando, mas devidamente filtrada para que apenas apareçam processos "lxterminal". Aqui pode ver-se que cada processo tem 3 *threads* criadas:

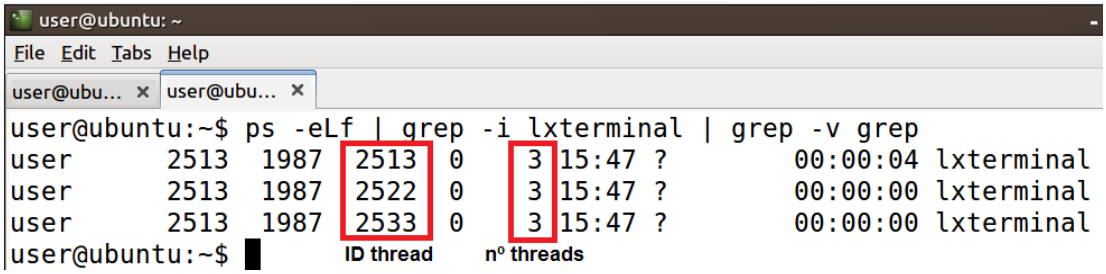

Figura 3: saída (parcial) do comando ***ps -eLf*** *com identificação da coluna* ***lwp*** *(ID thread) e* ***nlwp*** *(nº de threads)*

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" então o *output* deveria ser "1".

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*. 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 por parâmetro. A função recebe os seguintes 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 deverá 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 deverá utilizar uma estrutura.

2.1.1 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 ocorrido. Caso atribua o resultado à variável *errno*, poderá continuar a utilizar as macros ERROR e WARNING.

2.2. Identificador de uma *thread*

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

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

2.2.1 Valores de retorno

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

Insucesso – não se aplica.

2.2.2 Exemplo

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <errno.h>
#include "debug.h"
#define C_ERRO_PTHREAD_CREATE 1

void *hello(void *arg);

int main(int argc, char *argv[])
{
  pthread_t tid;
  (void)argc; (void)argv;
  if ((errno = pthread_create(&tid, NULL, hello, NULL)) != 0) {
    ERROR(C_ERRO_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;
}
```

Listagem 1 – Criação de uma thread

Lab 3

Sem recurso a *makefile*, compile apenas o código da Listagem 1.

Nota: para compilar este *Lab*, bem como os restantes *Labs*, é necessário efetuar a *linkagem* do código com a biblioteca *pthread*.

```
Exemplo:   $ gcc –o lab3 lab3.o -lpthread
                         lab3.c
```

2.3. Término de uma *thread*

Uma *thread* pode terminar de três maneiras:

- Implicitamente – quando a função (`start_routine`) executada pela *thread* chega ao fim (faça `return`);
- Abruptamente – quando uma das *threads* do processo chamar 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`.

Lab 4

Compile o programa do *Lab 4*, usando agora um ficheiro *makefile* e adicionando a referência à biblioteca **pthread**:

```
LIBS = -lpthread
```

Nota: use os ficheiros fornecidos no *Lab 4* e complete o *makefile* devidamente.

Execute o programa e explique a saída, associando-a a uma das 3 opções anteriores.

2.3.1 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 um único parâmetro:

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

2.3.2 Exemplo

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

#define C_ERRO_PTHREAD_CREATE 1

void *hello(void *arg);

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

  (void)argc;(void)argv;

  if ((errno = pthread_create(&tid, NULL, hello, NULL)) != 0) {
    ERROR(C_ERRO_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 praticas de programacao */
  return NULL; // Forma preferencial para terminar a thread
}
```

Listagem 2 – Término explícito de uma 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*.

2.4.1.1 Valores de retorno

Sucesso – devolve o valor zero.

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

2.4.1.2 Exemplo

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

#define C_ERRO_PTHREAD_CREATE 1
#define C_ERRO_PTHREAD_JOIN 2

void *hello(void *arg);

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

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

  printf("Processo: TID da thread criada = [%lu]\n",
                                        unsigned long) tid);

  if ((errno = pthread_join(tid, NULL)) != 0) {
    ERROR(C_ERRO_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;
}
```

Listagem 3 – Junção de threads

Lab 5

Compile e execute o programa do *Lab 5* e explique a sua saída.

Explique o modo de funcionamento do programa relativamente às três opções do término de *threads*.

2.4.2 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 retornado 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;

2.4.2.1 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 seguinte utiliza *threads* para processar, de forma concorrente, o cálculo do fatorial de um vetor de inteiros.

São usados 2 vetores:

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

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

```
#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_ERRO_PTHREAD_CREATE  1
#define C_ERRO_PTHREAD_JOIN  2
#define MAX 5
void *fatorial(void *arg);

struct ThreadParams
{
  unsigned long *ptrValor;
  unsigned long *ptrResultado;
};

int main(int argc, char *argv[])
{
  struct timeval tIni, tEnd;
  int i;
  pthread_t tids[MAX];
  unsigned long valores[MAX];
  unsigned long resultados[MAX];
  struct ThreadParams threadParams[MAX];

  (void)argc;(void)argv;

  // inicializar números para cálculo do fatorial (de 1 a 5)
  // "configurar" os parâmetros a passar às threads
  for (i = 0; i < MAX; i++){
    valores[i] = i + 1;
    // para cada thread passa-se apenas a posição a calcular
    threadParams[i].ptrValor = &valores[i];
    threadParams[i].ptrResultado = &resultados[i];
  }
```

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

  // criar as threads para cálculo e passar "o(s) parâmetro(s)"
  for (i = 0; i < MAX; i++) {
    if ((errno = pthread_create(&tids[i], NULL, fatorial,
                                &threadParams[i])) != 0) {
      ERROR(C_ERRO_PTHREAD_CREATE, "pthread_create() failed!");
    }
  }

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

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

  // mostrar resultado da esecução das threads
  printf("Tempo: %d segundos\n", (int)(tEnd.tv_sec - tIni.tv_sec));
  printf("Resultados:\n");
  for (i = 0; i < MAX; i++) {
    printf("\t%lu!: %lu\n", valores[i], resultados[i]);
  }

  exit(0);
}

void *fatorial(void *arg)
{
  // cast para o tipo de dados original
  struct ThreadParams *params = (struct ThreadParams *) arg;

  // obter valor para cálculo do fatorial
  unsigned long valor = *params->ptrValor;

  // calcular fatorial
  unsigned long aux = valor;
  while (--valor > 1) {
    aux *= valor;
  }

  // guardar valor calculado no vetor de resultados
  *params->ptrResultado = aux;

  //esperar um pouco (apenas para pergunta do próximo Lab)
  sleep(*params->ptrValor);

  return NULL;
}
```

Listagem 4 – Cálculo do fatorial de uma sequência de números recorrendo a threads

Lab 6

Compile e execute o programa do *Lab 6* 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.

Lab 7

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 efetuar o mesmo cálculo e retire as respetivas conclusões.

3. Exercícios

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

3.1. Para a aula

1. Elabore o programa que deve criar três processos filhos, 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. Elabore 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 efetuada 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. Elabore um programa que crie duas *threads* para processar (mostrar o número e se este é par ou ímpar) as diretorias existentes na diretoria `/proc`. Uma das *threads* deverá processar as diretorias cujo nome represente um número par, ao passo que a outra *thread* processa as diretorias cujo nome represente um número ímpar.

Elabore uma única função, que deve ser usada pelas duas *threads*, passe-lhe os parâmetros adequados e utilize as funções `opendir`, `readdir`, `closedir`, `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. Faça uma aplicação que crie, preencha (com valores aleatórios entre 0 e 9) e calcule a multiplicação de duas matrizes de tamanho 5x5. Como cada elemento da matriz resultante não depende 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:

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

5. Elabore um programa que, recorrendo à utilização de *threads*, ordene o conteúdo de um vetor de inteiros utilizando o algoritmo QuickSort (http://en.wikipedia.org/wiki/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.

[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.

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).