FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO
MIEIC - 2013/2014
SISTEMAS OPERATIVOS
Trabalhos Práticos

TRABALHO Nº 2

Gerador de números primos *multithreaded*

Objetivos do trabalho

Proporcionar a familiarização com a programação de sistema, em ambiente Unix/Linux, envolvendo, principalmente, o desenvolvimento de aplicações *multithread* e a utilização de mecanismos de comunicação e de sincronização entre *threads*.

Especificação

Pretende-se desenvolver um programa que determine os números primos entre 2 e N, usando o método conhecido por "peneira de Eratóstenes", cujo algoritmo geral se apresenta a seguir:

- Fazer uma lista com todos os números, de 2 até N.
- Repetir
    - Retirar o 1º número, X, da lista. Este número é primo.
    - Retirar da lista todos os múltiplos de X.

  até que a lista fique vazia.

O programa *multithreaded*, **primes**, deve aceitar o valor de N como argumento da linha de comando e, no final, mostrar os números primos resultantes. Cada iteração do ciclo acima indicado deverá ser feita por *threads* diferentes, funcionando "em *pipeline*". Cada *thread*, deverá receber do seu antecessor uma sequência de números e proceder à filtragem desta sequência, retirando-lhe o 1º número e todos os múltiplos deste, e passando a sequência obtida ao *thread* seguinte, para ser filtrada por este. O 1º número da sequência, retirado por cada *thread*, deve ser guardado numa zona de memória comum, acessível a todos os *threads*. No final, os números guardados nessa zona de memória deverão ser os números primos entre 2 e N.

O programa deve, genericamente, fazer o seguinte:

- alocar dinamicamente espaço para a zona de memória comum onde será guardada a lista de números primos; use a seguinte estimativa do número de valores primos a guardar: 1.2*(N / ln N) ;
- preparar a lista inicial de números a filtrar ("peneirar");
- lançar em execução os *threads* que vão proceder à filtragem, como abaixo indicado;
- aguardar que a determinação dos números primos termine;
- ordenar a sequência de números guardados na zona de memória comum (usar `qsort`) e apresentá-la no ecrã.

O programa deve criar um *thread* inicial que prepara a lista de números a processar e uma sequência de *threads*-filtro, cada um dos quais vai processando a lista de números que lhe vai sendo transmitida, gerando uma lista de números filtrados ("peneirados"). A comunicação entre os *threads* da sequência deve ser feita através de filas circulares (uma por cada dois *threads* consecutivos), cujo acesso deve ser devidamente sincronizado. Cada *thread*-filtro terá, por isso, uma fila de entrada e uma fila de saída. O processamento consistirá, genericamente, em retirar o 1º número da fila de entrada, X, guardando-o na região de memória comum, e em escrever na fila de saída os números que vão chegando através da fila de entrada que não sejam múltiplos de X.

Mais especificamente, o *thread* inicial deve fazer o seguinte:

- colocar o inteiro 2 (o 1º número primo) na zona de memória comum;
- se N for superior a 2
    - criar uma fila circular de saída na qual colocará, futuramente, a sequência de números abaixo indicada;
    - criar um *thread*-filtro que vai processar os números escritos na fila circular, passando o endereço desta fila como parâmetro do *thread* criado;
    - colocar na fila os números ímpares entre 3 e N (para além do número 2, apenas estes podem ser números primos); após estes números deverá escrever o número 0 (zero) para indicar o fim da sequência;

- se não
  - sinalizar um semáforo que indica o fim da determinação dos números primos**,** terminando a seguir; este semáforo deverá ser inicializado pelo *thread* principal do programa.

Cada *thread*-filtro deve ir lendo os números que vão chegando através da sua fila de entrada e processando-os de acordo com as seguintes regras:

- se o 1º número da fila de entrada for maior que **`(int) (sqrt(N))`**
  - vai escrevendo na zona de memória comum os números que lhe vão chegando através da fila de entrada, até que surja o valor 0 (zero); todos estes números, exceto o zero final, são seguramente primos (porquê ?);
  - no fim, sinaliza o semáforo anteriormente referido e termina;
- se não
  - toma nota do 1º valor da sequência (este número é primo);
  - cria uma fila de saída;
  - cria o *thread*-filtro seguinte da sequência, passando-lhe como parâmetro o endereço desta fila (que será a fila de entrada deste novo *thread*);
  - vai filtrando os números da sua fila de entrada que sejam múltiplos do 1º valor da sequência, deixando o resultado na fila de saída; a sequência de valores escritos na fila de saída deve ser terminada pelo valor 0 (zero);
  - escreve na zona de memória comum o 1º valor da sequência.

Na zona de memória comum, para além do espaço para a lista de números primos, é necessário guardar o índice do próximo número primo a inserir nessa lista. Os *threads* devem sincronizar o acesso a estas variáveis, de forma a garantir que a sua atualização é feita de forma correta.

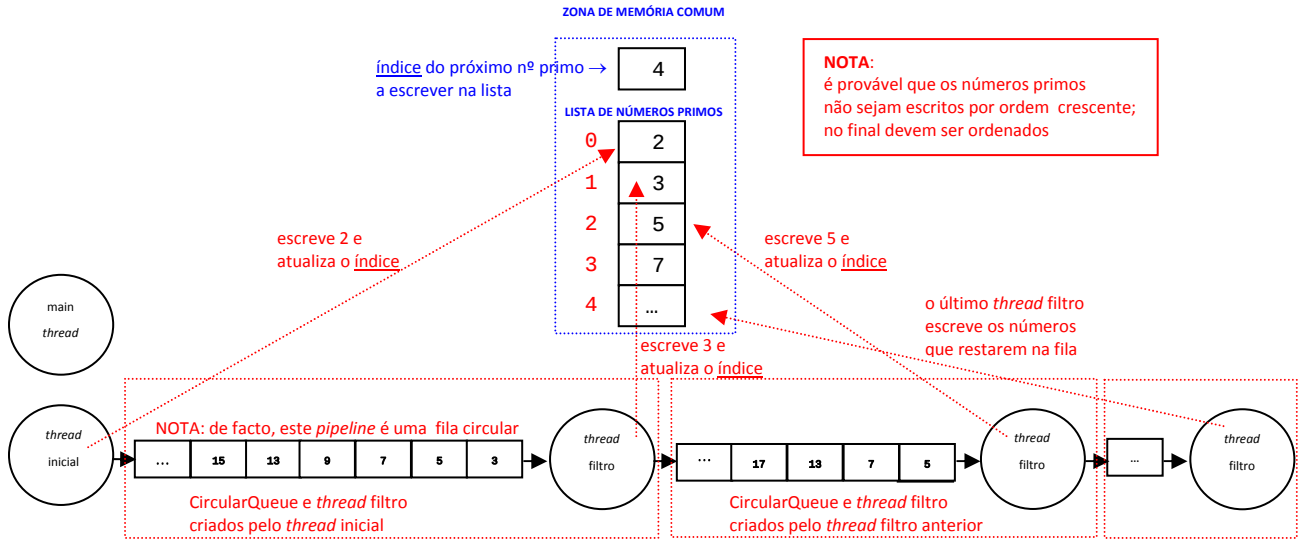
Notas:

1) Em anexo é fornecido parte do código necessário para criação e manipulação de uma fila circular, recorrendo a mecanismos de sincronização que impedem a escrita na fila quando ela estiver cheia ou a leitura de elementos quando ela estiver vazia, segundo o paradigma do produtor-consumidor.

2) Sugere-se que comece por desenvolver um programa de teste do funcionamento de uma fila circular, criando uma fila partilhada por dois *threads*, um que coloca na fila a sequência de números 1,2,3,4,5, …0 (terminada por zero) e outro que retira os números colocados na fila e os apresenta no ecrã.

Entrega do trabalho

- Data limite para a entrega do trabalho: 25/05/2014, às 23:55h.
- Oportunamente serão publicadas algumas regras para a entrega do trabalho na página de "Sistemas Operativos", no Moodle da Universidade do Porto.



ANEXO

Código para a criação e manipulação de uma fila circular com acesso sincronizado

As filas circulares para comunicação entre dois *threads* devem ser implementadas por uma *struct*, **`CircularQueue`**, que é manipulada por 4 funções que constituem a sua *API* :

- `void queue_init(CircularQueue **q, unsigned int capacity)`
- `void queue_put(CircularQueue *q, QueueElem value)`
- `QueueElem queue_get(CircularQueue *q)`
- `void queue_destroy(CircularQueue *q)`

Apresenta-se a seguir a definição de **`CircularQueue`**, a descrição da funcionalidade de cada uma das 4 funções, bem como o código de `queue_init()`. O código das restantes funções deve ser desenvolvido no âmbito deste trabalho.

```
//-------------------------------------------------------------------------------------------
// Type of the circular queue elements

typedef unsigned long QueueElem;

//-------------------------------------------------------------------------------------------
// Struct for representing a "circular queue"
// Space for the queue elements will be allocated dinamically by queue_init()

typedef struct
{
  QueueElem *v;     // pointer to the queue buffer
  unsigned int capacity; // queue capacity
  unsigned int first;    // head of the queue
  unsigned int last;     // tail of the queue
  sem_t empty;           // semaphores and mutex for implementing the
  sem_t full;            // producer-consumer paradigm
  pthread_mutex_t mutex;
} CircularQueue;

//-------------------------------------------------------------------------------------------
// Allocates space for circular queue 'q' having 'capacity' number of elements
// Initializes semaphores & mutex needed to implement the producer-consumer paradigm
// Initializes indexes of the head and tail of the queue
// TO DO BY STUDENTS: ADD ERROR TESTS TO THE CALLS & RETURN a value INDICATING (UN)SUCESS

void queue_init(CircularQueue **q, unsigned int capacity) // TO DO: change return value
{
  *q = (CircularQueue *) malloc(sizeof(CircularQueue));
  sem_init(&((*q)->empty), 0, capacity);
  sem_init(&((*q)->full), 0, 0);
  pthread_mutex_init(&((*q)->mutex), NULL);
  (*q)->v = (QueueElem *) malloc(capacity * sizeof(QueueElem));
  (*q)->capacity = capacity;
  (*q)->first = 0;
  (*q)->last  = 0;
}

//-------------------------------------------------------------------------------------------
// Inserts 'value' at the tail of queue 'q'

void queue_put(CircularQueue *q, QueueElem value)
{
  // TO DO BY STUDENTS
}

//-------------------------------------------------------------------------------------------
// Removes element at the head of queue 'q' and returns its 'value'

QueueElem queue_get(CircularQueue *q)
{
  // TO DO BY STUDENTS
}

//-------------------------------------------------------------------------------------------
// Frees space allocated for the queue elements and auxiliary management data
// Must be called when the queue is no more needed

void queue_destroy(CircularQueue *q)
{
  // TO DO BY STUDENTS
}
//===========================================================================================
// EXAMPLE: Creation of a circular queue using queue_init()
#define QUEUE_SIZE 10  // TO DO: test your program using different queue sizes
...
CircularQueue *q;
queue_init(&q,QUEUE_SIZE);
...
```