Relatório do EP2
MAC0422 – Sistemas Operacionais – 2s2017

Anderson Andrei da Silva (8944025),
Bruno Boaventura Scholl (9793586),
Victor Seiji Hariki (9793694).

1 Introdução

1.1 O problema

Uma das várias modalidades de ciclismo realizada em velodromos é a corrida por pontos. O objetivo deste EP será simular essa modalidade. Todos os detalhes estão descritos no arquivo **ep2.pdf** presente na pasta *documentation* que se encontra no diretório do projeto.

1.2 Disposição dos arquivos

Os arquivos estão dispostos da seguinte maneira:

- Na **raiz do diretório** se encontram : Makefile, LEIAME e as pastas descritas a seguir:
- Na pasta **execsrc** se encontram dispostos todos os arquivos que gerarão executáveis:
  - ep2.c : Arquivo que contém a main() deste exercício programa.
- Na pasta **src** se encontram todos os arquivos em c que serão utilizados mas não gerarão executáveis:
  - rider.c : Ctém as funções e structs relactionadas à representação de um ciclista
  - velodrome.c : Contém funções mais gerais, que necessitam de um conhecimento global do estado da corrida
- Na pasta **include** se encontram todos os arquivos de header (.h) que serão utilizados nos outros arquivos:
  - rider.h
  - typedef.h
  - velodrome.h
- Na pasta **documentation** se encontram :
  - ep2.pdf : o enunciado deste exercício programa;
  - relatorio-ep2-trio.pdf : este relatório;
  - commits : um arquivo em texto puro com o histórico de commits do repositório do projeto;

  - apresentacao.pdf : a apresentação do projeto em slides.

- Na pasta **bin** se encontra o binário *ep2*.
- Na pasta **obj** se encontram os arquivos *.o* criados pela compilação.
- Na pasta **testsrc** se encontra um código criado para testar o programa que torna visível a corrida, *graphical.c* .
- Na pasta **testbin** se encontra o binário do código de testes, *graphical*.

1.3 Algoritmos

Para a resolução do problema, foram criados e utilizados pelo grupo os seguintes algoritmos:

1.3.1 Ciclistas

Os ciclitas serão representados cada um por uma thread, tendo como características específicas:

- int id;
- pthread_t rider_t;
- Velodrome velodrome;
- int speed;
- bool broken;
- bool finished;
- int score;
- int total_time;
- int total_dist;
- int lane;
- int step_time;
- uint turn;
- sem_t turn_done;
- int *overtake;

O conjunto de todos eles será representado por um vetor de threads. Pontos como a movimentação de cada um deles será melhor descrita na seção 1.6 .

Por serem o método de representação dos ciclistas sua função principal é ”correr”e o algoritmo segue principalmente o seguinte pseudocódigo:

- set initial speed
- wait start
- if lap completed

    - choose new speed
    - check if scores
    - if lap is multiple of 15
        * decide if breaks
- go forward, left or right
    - if is a rider in front
        * wait rider in front, left or right do its turn
        * checks max speed possible and reduce if needed
- notify rider behind semaphore
- notify global barrier
- wait global barrier

1.3.2 Velódromo

Para simular a corrida, dado um vetor de threads, cada thread será disparada a cada ciclo que representam 60 ms reais. De acordo com suas características, em cada volta desempenharão seu rendimento. Cada uma é disparada por vez, com ordem estipulada especificamente pelo SO e o que diferenciará o rendimento individual de cada uma serão suas características, que serão contabilizadas e agregadas a cada ciclo.

Assim, teremos então diferentes desempenhos por ciclos e em uma visão geral, teremos a simulação da corrida.

O gerenciador de tudo que foi descrito à cima , assim como dos cicilistas, pontuações e etc é o **velódromo**, que possui as seguintes características:

- int length; *// Length of velodrome, in meters*
- uint rider_cnt; *// Total number of riders*
- uint a_rider_cnt; *// Number of active riders*
- uint lap_cnt; *// Turn count*
- int **pista;
- struct Rider *riders;
- pthread_t coordinator_t;
- pthread_barrier_t start_barrier;
- sem_t velodrome_sem;
- int round_time; *// How much time passes in one barrier round*
- sem_t *arrive;
- sem_t *continue_flag; *// Flag to pass barrier*
- int *placings; *// Remaining riders*

- int *s_indexes; *// Stack of placings by round*
- uint **placings_v;
- sem_t rand_sem; *// Random generator semaphore*
- sem_t score_sem; *// Score's semaphore*
- struct Rider *a_score; *// Score's rider controller*

E tem por seu algoritmo principal o seguinte pseudocódigo:

- Alocate the struct Velodrome
- Start the semphores (blocked)
- Alocate the track
- Alocate the stack of scores
- Create the riders
- Crate the placings
- Start the riders
  - Start the global barrier - *pthread_barrier_init*
  - Create the threads - *pthread_create*
  - Create the coordinator
  - Wait the threads - *pthread_barrier_wait*

1.4 Barreiras de sincronização

Foi utilizada uma barreiras de sincronização para tratar a largada da corrida. Temos de esperar que todas as threads sejam devidamente carregadas, como se os ciclistas estivem se posicionando na pista, para que a corrida possa começar. Assim sendo, a barreira sincroniza a criação e a preparação de todas as threads para a largada.

Para tal implementação foi utilizada a biblioteca **pthread.h** que possibilita o uso de threads e barreiras para elas:

- **pthread_barrier_init** : Inicializa uma barreira;
- **pthread_barrier_wait** : Faz a barreira barrar as threads.

1.5 Semáforos

Serão utilizados para controlar vários trechos considerados pelo grupo como seções críticas. Tais semáforos, seguidos das situações que eles tratam, são respectivamente:

- **sem_rand** : Utilizamos a geração de números aleatórios nas decisões de movimentação dos ciclistas e o que pode ocorrer é que mais de uma thread execute tal trecho ao mesmo tempo e assim obtenham o mesmo valor, o que não é desejado.
- **velodromo_sem** : Utilizado para controlar o acesso das threads à pista, evitando assim que mais de uma thread escreva na pista simultaneamente.

- **arrive** e **continue_flag** : Estes são vetores de semáforos, utilizados como barreiras do disparo das threads (sem serem barreiras propriamente ditas, como a da seção anterior). O grupo implementará a execução das threads conforme será descrito nas próximas seções. Mas para tal, a cada execução do ciclo, a ordem da execução das threads é desconhecida pelo grupo pois depende do SO. O que pode então resultar na execução sempre da mesma thread, pois ela pode ser muito rápida e terminar sua tarefa antes que outra seja selecionada, podendo assim ser escolhida para execução novamente.

  O grupo fará uso de uma thread ”coordenadora”e a mesma para bloquear e liberar as threads precisa de dois semáforos. Ela marca a chegada das threads com o primeiro semáforo (arrive), fazendo com que as outras esperem até que todas cheguem, e libera com o segundo (continue_flag), permitindo assim suas execuções. O fato de serem vetores de semáforos se deve ao fato de cada posição do vetor ser responsável por uma thread específica.

  O vetor de semáforos arrive também será utilizado para tratar casos nos quais vários ciclistas estarão na mesma pista. Assim o ciclista da frente é o limite de movimentação do anterior. Sendo assim, as threads que estão atrás tem que esperar a da frente se mover. Assim sendo, utilizaremos semáforos nesse ponto para poder impedir a possível movimentação (seleção do sistema daquela determinada thread) de ciclistas que estão impossibilitados por tal motivo. Logo, a movimentação só é liberada quando o semáforo permitir, que fará isso após a movimentação dos ciclistas da frente.

- **score_sem** Utilizado para controlar a marcação e exibição da pontuação e classificação dos ciclistas ao decorrer da execução do programa. Pode ocorrer de várias threads escreverem ao mesmo tempo na pilha que controla tais pontos, o que não é desejado.

- **sem_print** Utilizado nos cinco casos de print que permitem acompanhar o adamento da corrida. São os casos de exibição da classificação de todos os ciclistas a cada volta; a pontuação acumulada por cada ciclista a cada 10 voltas (em ordem decrescente); a ocorrência da ”quebra”de algum ciclista; a exibição do status final da corrida (pontuação final, instante de finalização da prova, exibição de todos os ciclistas ”quebrados”e em qual volta) e o modo debug.

  Todos esses cinco casos tem como seção crítica a exibição de suas mensagens, o que não podem ser feitas por várias threads ao mesmo tempo.

Para tais implementações foi utilizada a biblioteca **semaphore.h** que possibilita o uso de semáforos:

- **sem_post** : Libera o semáforo, sinalizando a permissão para a entrada na seção crítica;

- **sem_wait** : Bloqueia o semáforo, sinalizando o bloqueio da entrada na seção crítica.

1.6 Threads

As threads representarão os riders (ciclitas) e terão suas características como já descrito na seção 1.3.1. Ao ser criada, uma thread receberá suas características e se posicionará na pista, aguardando o iniício da corrida. Cada execução será feita dentro de um ciclo que representa 60 ms (em geral, mas os dois últimos ciclos representarão 20 ms). A cada ciclo os dados de cada thread é atualizado, de forma a simular seu desempenho na corrida. Para igualar as condições de todas as threads, essas passarão por barreiras e semáforos (descritos na seção 1.4 e 1.5).

A função principal de cada thread é ”correr”, e para tal, a função que simula tal movimento verifica todas as condições possíveis para os casos de aceleração e redução de velocidade, ultrapassagem, pontuação a cada sprint, bonificação (no caso de um corredor der uma volta completa em todos os outros) e até mesmo a ”quebra”dos ciclistas.

Para melhor controlar todas as threads teremos uma além da quantidade de ciclistas, que executará o papel de coordenador, assim como visto em algoritmos em aula. Essa thread ficará responsável por barrar e liberar a execução das threads de forma que não ocorram conflitos por condições de corrida. Ele tem por seu algoritmo principal o seguinte pseudocódigo:

- While the amount of active riders greater than zero
  - For all riders
    - if !(rider broken) and !(rider finished)
- Start the semaphores (blocked)
- Allocate the track
- Allocate the stack of scores
- Create the riders
- Crate the placings
- Start the riders
  - Start the global barrier - *pthread_barrier_init*
  - Create the threads - *pthread_create*
  - Create the coordinator
  - Wait the threads - *pthread_barrier_wait*

Para os casos de aumento e redução de velocidade, os ciclitas podem mudar entre 30, 60 e até 90 Km/h. Será computado distâncias percorridas a cada 1 metro. Assim sendo, ciclistas que estiverem a 30 Km/h e assim se mantiverem, ao completar o primeiro ciclo não terão percorrido 1m completo, e a nível de implementação, permancerão no mesmo ponto onde estavam no início do ciclo e só se moverão ao término do segundo, tendo então se movimentado por 1m.

Corredores que ”quebrarem”são imediatamente removidos da corrida, e a thread que o representa é destuída, mas o restante de seus dados continuam salvos para futuras exibições.

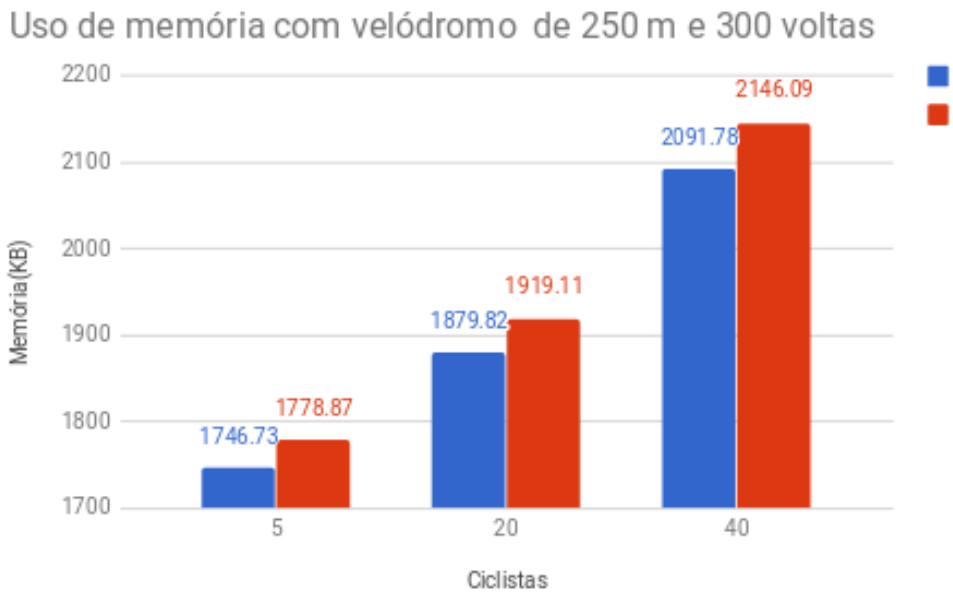
Serão exibidas informações das threads conforme requisitado no enunciado deste trabalho, assim como existirá um modo *debug* também especificado.

2 Resultados

Os testes foram feitos de forma automatizada utilizando bash script. Um script com dezoito loops de trinta testes, variando as entradas e a quantidade de ciclistas de acordo com o que foi pedido no enunciado. Os resultados obtidos foram tratados e geraram os seguintes gráficos (com média e intervalo de confiança calculados automaticamente na geração dos mesmos) :

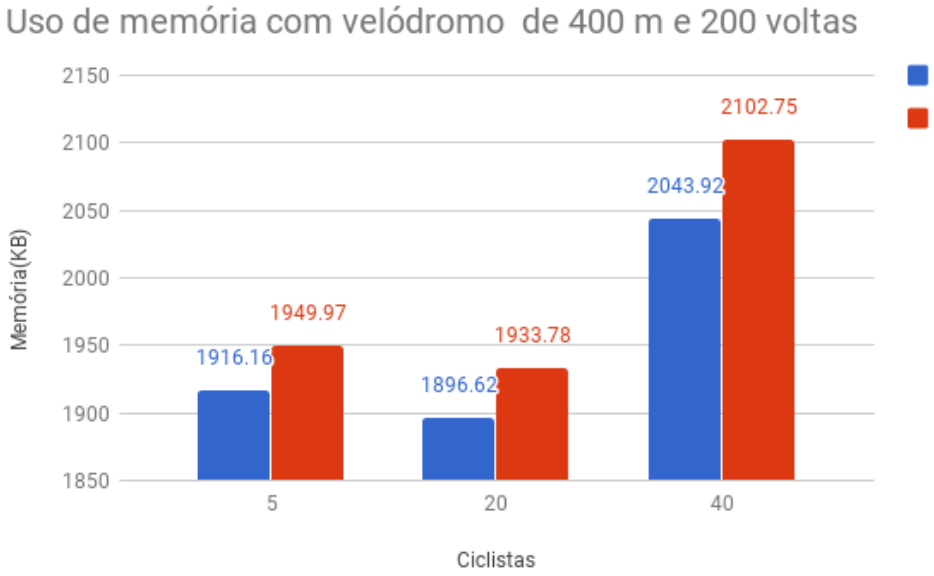
- **Impacto no uso de memória**
  Observação: As barras vermelhas representam o limite superior e as azuis o limite inferior do intervalos de dados medidos.

– *Figura 1*



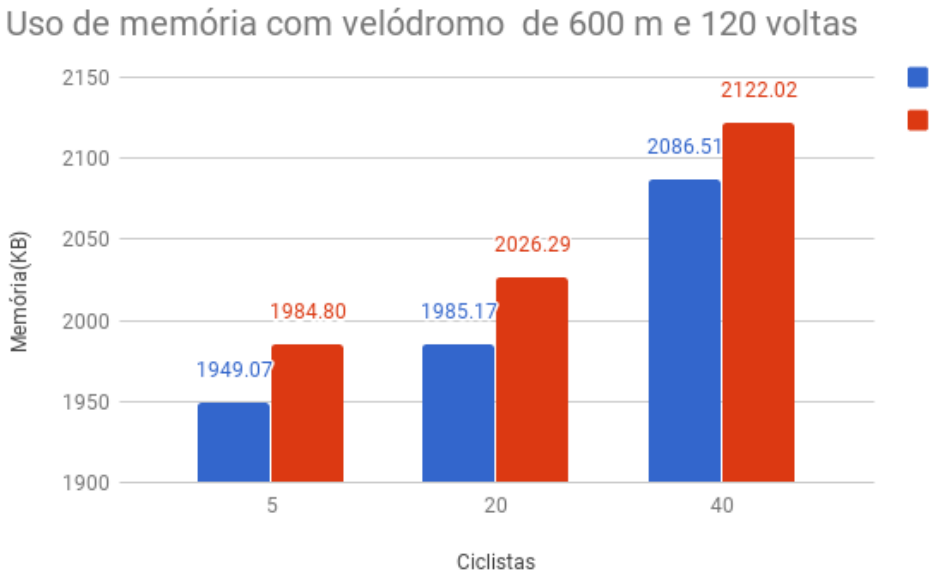
Conclusão : Conforme aumentou a quantidade de ciclistas as execuções consumiram mais memória, como esperado.

– *Figura 2*



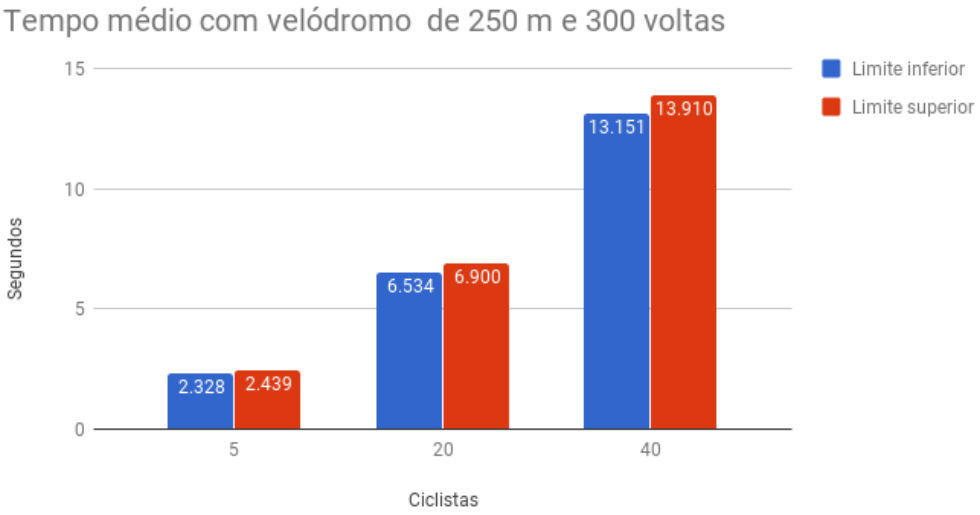
Conclusão : Era esperado um comportamento parecido com o da figura 1, porém para o caso de 20 ciclistas pode-se obeservar que foi consumido menos memória que no menor caso (5 ciclistas). O grupo não conseguiu identificar o motivo para tal comportamento.

– *Figura 3*



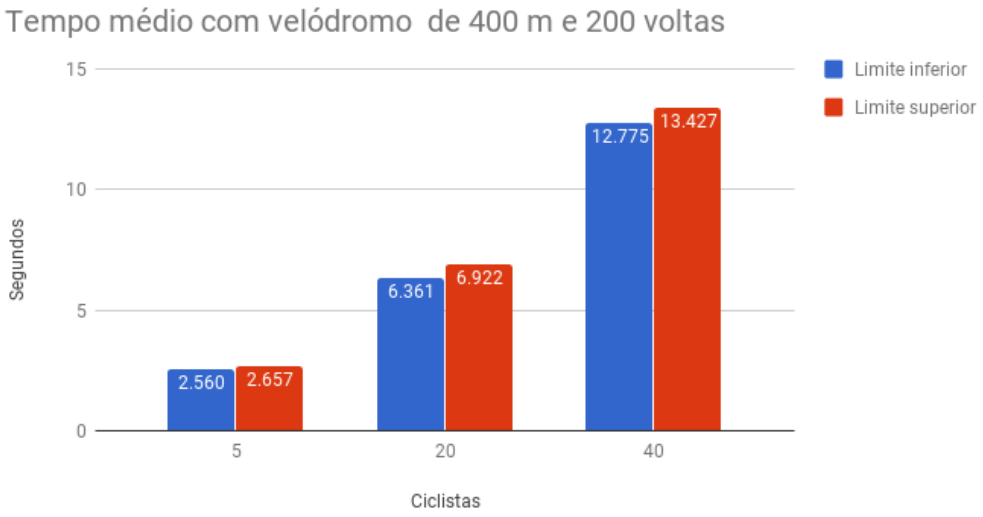
Conclusão : Assim como na figura 1 ocorreu o esperado, o crescimento do consumo de memória foi proporcional ao aumento da quantidade de ciclistas.

- **Impacto no tempo de execução**
  - *Figura 4*



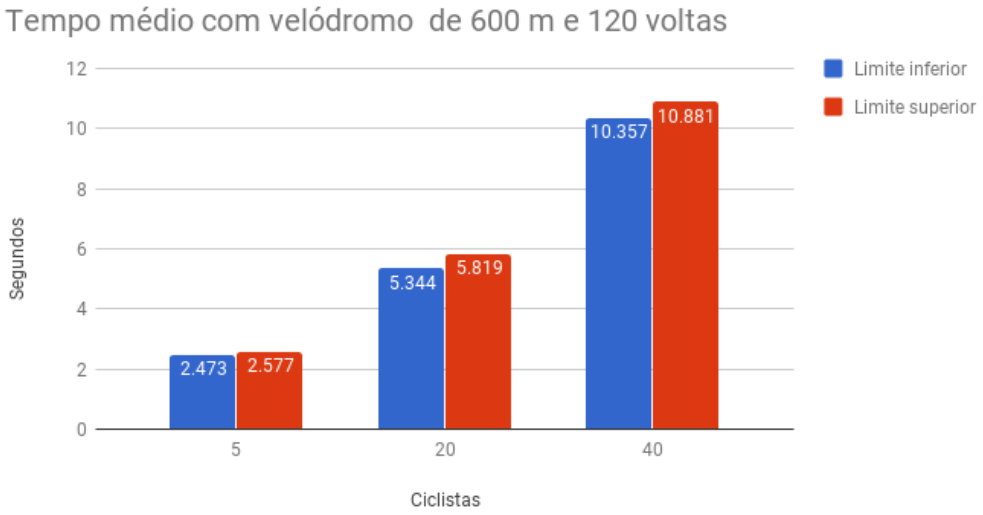
Conclusão : Conforme aumentou a quantidade de ciclistas as execuções levaram mais tempo para terminar, como esperado.

  - *Figura 5*



Conclusão : Era esperado um comportamento parecido com o da figura 4, porém para o caso de 20 ciclistas pode-se obeservar que levou-se menos tempo que no menor caso (5 ciclistas). Isso é provavelmente devido ao maior número de voltas para o primeiro teste.

– *Figura 6*



Conclusão : Assim como na figura 4 ocorreu o esperado, o crescimento do tempo médio de execução foi proporcional ao aumento da quantidade de ciclistas.

Observação: Foi notado pelo grupo que casos em que o nº de ciclistas for maior ou igual ao comprimemto da pista podem gerar deadlocks, por causa de uma situação de ”filósofos famintos”, no qual cada ciclista espera que o que está na frente dele se mova. Não desenvolvemos uma solução prática para o problema, mas uma solução possível seria a adição de um timeout.

Conclusão geral: A quantidade de ciclistas (threads) é a principal influência na duração da execução do programa e no consumo memória para tal. Diferente do número de voltas e comprimento da pista, nos quais o grupo não achou relação direta com o tempo e memória gastos.

3 Divisão das tarefas de implementação

Todos os integrantes fizeram parte de todas as tarefas. Conforme foram dividas houveram responsáveis por cada uma delas, mas todos verificaram, validaram e auxiliaram no desenvolvimento de

todas. Foram estipuladas as seguintes tarefas :

- **Relatório** : Responsável Anderson;
- **Slides** : Responsável Anderson;
- **Código** : Todos os integrantes fizeram código, ficando responsáveis por:
  - **Rider** : Reponsável Bruno;
  - **Velódromo** : Responsável Victor;
  - **Pontuação** : Resposável Anderson.

Na pasta **documentation** presente no diretório do projeto consta o histórico de *commits* do repositório.

4 Referências

O grupo não utilizou de livros texto para consultas e sim manuais do sistema e fórums como ”stack overflow”e etc.