Relatório

Simulador de Corridas

2018285164-Eduardo F. F Cruz 2019216314-Gonçalo M. Barroso

Introdução

Neste trabalho prático, desenvolvemos um sistema que permite simular, de forma simplificada, uma corrida de carros, semelhante a Fórmula 1. Este sistema foi conseguido através da utilização de processos e threads que comunicam entre si por meio de IPCs como *Shared Memory*, *Pipes* ou *Message Queues.* Tendo em conta a necessidade dos vários processos e threads lerem e/ou escreverem em memória partilhada ou para os *pipes*, foi fundamental garantir que esta leitura e escrita era realizada de forma sincronizada e eficiente. Para isso, e para evitarmos qualquer tipo de espera ativa, utilizámos vários mecanismos de sincronização, nomeadamente semáforos, *pthread_mutex*'s e variáveis de condição.

Funcionamento do Programa

Inicialmente, no processo Race Simulator (main), são importados os dados de configuração, do ficheiro config.txt, e armazenados numa struct config. Estes dados não sofrem alterações durante a execução do programa e são acessíveis a todos os processos e threads deste sistema, dado que, aquando da sua criação, os processos filhos herdam todas as variáveis globais do processo pai. De seguida criamos e inicializamos a Shared Memory que consiste numa struct mem_struct, num array de struct car e num array de struct team. Todos os dados necessários ao bom funcionamento da corrida serão guardados em memória partilhada. Ainda no processo main, criamos semáforos Posix binários para sincronizar a escrita no ficheiro de log, a escrita e leitura de variáveis associadas a estatísticas de corrida, etc. Alocamos espaço para os file descriptors dos unnamed pipes (cada equipa possuí um unnamed pipe), criamos o named pipe utilizado para receber comandos, criamos a Message Queue utilizada para transmitir as avarias aos carros e fazemos com que todos os processos ignorem os sinais SIGINT, SIGTSTP e SIGUSR1, tendo em vista que apenas um processo (Race Simulator) irá lidar com o sinal SIGINT e SIGTSTP e outro (Race Manager) com o SIGUSR1. São criados o processo Race Manager e Malfunction Manager, e o processo main passa a aguardar por eventuais sinais, até ser terminado. O processo Race Manager cria um processo para cada equipa e através do *select,* o processo fica bloqueado a aguardar receber comandos pelo named pipe ou alguma mensagem pelos unnamed pipes. É neste processo, através da leitura dos pipes que toda a corrida é gerida.

Existem duas variáveis em memória partilhada, race_state, que permitem aos processos saber se a corrida está a decorrer ou não, e stop_race que permite saber se foi recebido um SIGINT ou um SIGUSR1, e dependendo do seu valor, terminar a corrida assim que possível. O processo Malfunction Manager permanece bloqueado até a corrida ser iniciada. Isto é conseguido através do uso de variáveis de condição, que neste caso, aquando de uma alteração na variável race_state é realizado um broadcast para que todos os processos bloqueados nesta condição, verifiquem se reúnem condições para serem desbloqueados. Os processos Team Manager de cada equipa dependem também desta variável, entre outras, para iniciarem a gestão da BOX, assim como as car threads para iniciarem a corrida.

A criação das threads dos carros é realizada pelo processo Team Manager de cada equipa. Cada processo tem um counter de car threads que criou até ao momento e encontra-se bloqueado por uma variável de condição, enquanto o valor desse counter for igual ou superior ao número de carros escritos em memória partilhada para essa equipa. Ao receber um comando válido no named pipe, o processo Race Manager, caso estejam reunidas condições, obtém o lock de um mutex e escreve os dados recebido em memória partilhada. Após escrever os dados (alterar o estado da corrida ou adicionar um carro à SHM), é feito pthread_cond_broadcast, para que os processos a aguardar por esta variável de condição atestem a

sua condição e, por exemplo no caso de ter sido adicionado um carro, todos os processos Team Manager verificam se o carro foi adicionado na sua equipa e eu caso positivo, criam a respetiva thread para simular esse carro. Como já foi mencionado, a thread fica bloqueada até a corrida começar. Durante algumas fases da execução do programa, nomeadamente durante a fase em que a corrida está a decorrer, há certas variáveis em memória partilhada que podem ser acedidas sem que haja a necessidade de sincronizar esse acesso com semáforos binários ou pthread_mutexes, dado que os outros processos que poderiam efetuar escrita nessas variáveis, estão bloqueados enquanto a corrida decorre. De notar que, por questões de eficiência, cada equipa possuí um pthread_mutex que os carros associados a essa equipa utilizam para escrever no unnamed pipe atribuído a cada equipa. Não é necessário sincronizar a escrita e leitura da Message Queue dado que esta não é responsabilidade do programador e sim do sistema operativo.

Para conseguirmos um término de todos os processos e threads de forma controlada, e a limpeza e destruição de todos os mecanismos de sincronização ou memória alocada, utilizamos a variável stop_race, já mencionada anteriormente. Esta variável foi protegida através da implementação do caso conhecido de readers/writers, sem starvation e através de semáforos Posix, para que vários processos ou threads possam ler a variável ao mesmo tempo, enquanto esta não está a ser escrita, implementação que contribuí para a eficiência do programa. Esta variável força o término da corrida assim que todos os carros chegarem à meta e, dependendo do valor da stop_race, ou todos os processos e threads bloqueados são desbloqueados e terminados de forma ordenada, ou voltam ao estado em que estavam antes de receberem o comando “START RACE!”, permitindo a adição de mais carros e o início de uma nova corrida.

Mecanismos de Sincronização

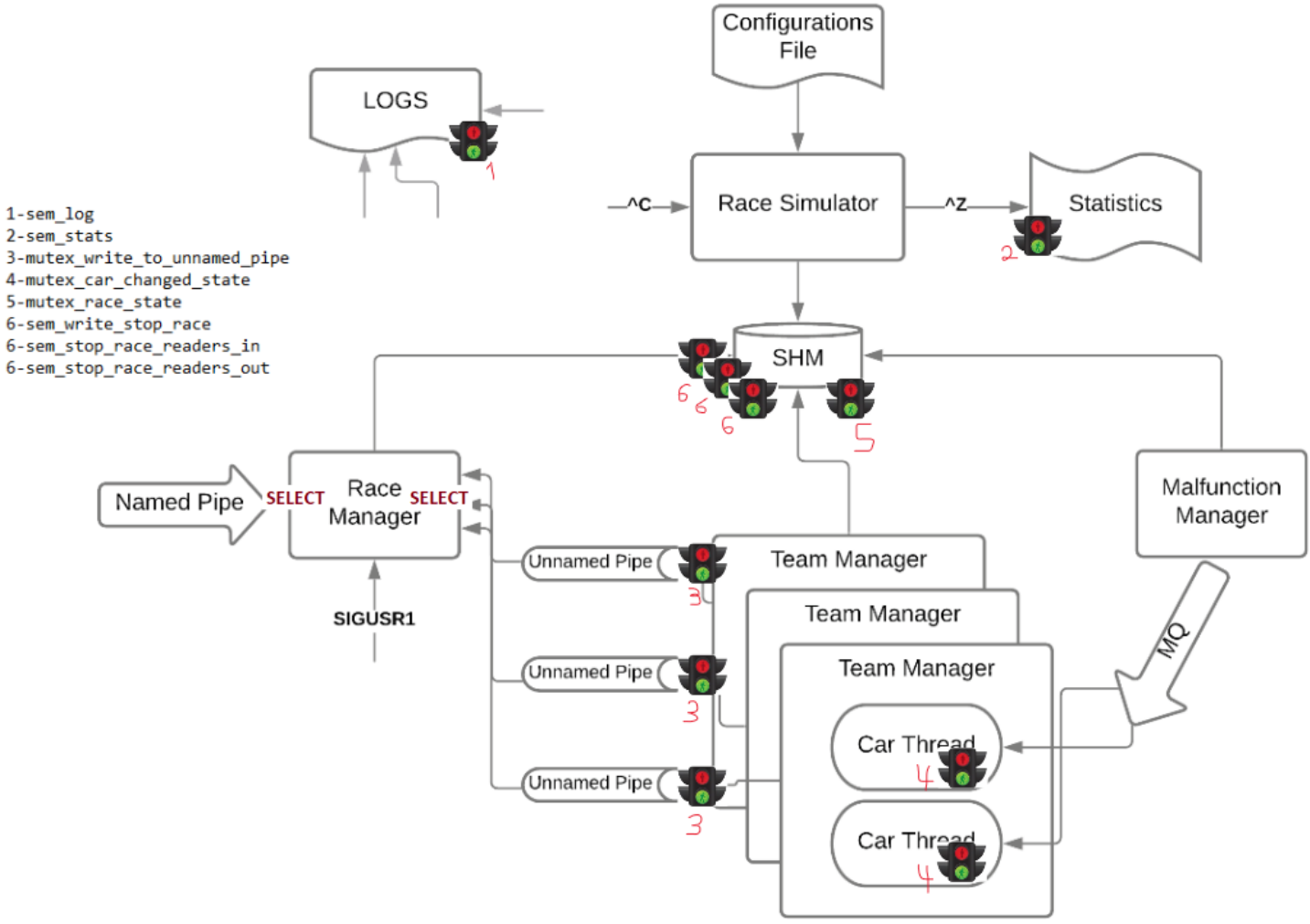

Figura 1- Arquitetura do Programa (pthread_cond_var não se encontram representadas)

Horas despendidas no projeto: Eduardo-Aproximadamente 30h; Gonçalo-Aproximadamente 30h.