UNIVERSIDADE da MADEIRA

Sistemas Operativos

Projeto Prático 2017/2018 – 2ª fase

Simulação de uma montanha-russa

Docentes

Eduardo Marques | Luís Gaspar

(emarques@uma.pt | lgaspar@staff.uma.pt)

Discentes

Cláudio Sardinha | Joaquim Perez | Luís Freitas

(2030215 | 2029015 | 2029715)

Índice

1. Protocolo de comunicação entre o Monitor e o Simulador

Em termos de comunicação, o Monitor e o Simulador vão comunicar entre si através de *sockets*, através da função mostrada abaixo. A família de protocolos escolhidos para esta conexão é o AF_UNIX, que indica que se trata de uma comunicação local (num sistema UNIX). Para além disso, o tipo da *socket* é indicado – *SOCK_STREAM.* Para averiguar situações de erro, na função verificarErro, é verificado o resultado da função *socket*, sendo que, quando é devolvido o valor de -1, trata-se de uma situação errónea.

```
socket_t criarSocket()
{
    socket_t s = socket(AF_UNIX,SOCK_STREAM,0);  //é criado um socket descriptor
    verificarErro(s);                            //o programa só continua caso não ocorra erro no passo anterior
    return s;
}
```

Figura 1: Criação do socket

Daí, são criadas estruturas (uma para o Monitor e outra para o Simulador) – sockaddr_un – onde será armazenado o endereço do espaço partilhado entre o Monitor e o Simulador.

```
sockaddr_un criarLigacaoSocket()
{
    sockaddr_un socket_addr;                    //
    socket_addr.sun_family=AF_UNIX;             //família do tipo AF_UNIX
    strcpy(socket_addr.sun_path,UNIXSTR_PATH);  //é indicado o endereço do espaço

    return socket_addr;
}
```

Figura 2: Criação do espaço partilhado das sockets

De seguida, por parte do Monitor, à sua *socket* é feito o *bind* para o dado endereço presente na respetiva estrutura de dados, para depois então ser esperada (através do *listen*) que o cliente (Simulador) faça a tentativa de conexão com o servidor (Monitor). Quando tal é feita, o Monitor trata de aceitá-la, recorrendo à função *accept*. Como em

qualquer outra parte do código, é sempre verificado, passo a passo, a ocorrência de erros (interpretada pelo resultados das determinadas funções). As etapas referidas anteriormente podem ser analisadas na figura abaixo.

```
client_t esperarPorCliente()
{
    socket_t s = criarSocket();                                  // é criada a socket
    sockaddr_un s_addr = criarLigacaoSocket();                   // estrutura de endereço do Monitor
    unlink(UNIXSTR_PATH);                                        // é apagado o ficheiro partilhado

    int bindstate = bind(s,(sockaddr*) &s_addr,sizeof(sockaddr_un));   // é atribuído o endereço à socket
    verificarErro(bindstate);                                    // o programa só prossegue caso não ocorram erros

    int listenstate = listen(s,1);                               // espera pela conexão com 1 cliente (Simulador)
    verificarErro(listenstate);                                  // o programa só prossegue caso não ocorram erros

    sockaddr_un client_addr;                                     // estrutura de endereço do cliente (Simulador)
    int clientlen = sizeof(client_addr);
    client_t client; = accept(s,(sockaddr*) &client_addr,&clientlen);  // quando ocorra, aceita a conexão com o cliente (Simulador)
    verificarErro(client);                                       // o programa só prossegue caso não ocorram erros

    return client;
}
```

Figura 3: Conexão Monitor-Simulador (do ponto de vista do Monitor)

Da mesma forma, do ponto de vista do Simulador (o cliente), é criada a *socket* com a indicação para o mesmo endereço do que o Monitor. Evidentemente, daí é feita a tentativa de conexão (que será aceite) com o Monitor, através da função *connect*.

```
socket_t criarSocketCliente()
{
    socket_t s = criarSocket();                                  //é criada a socket
    sockaddr_un s_addr = criarLigacaoSocket();                   //estrutura do endereço do cliente (Simulador)

    int connectstate = connect(s,(sockaddr*) &s_addr,sizeof(sockaddr_un)); // é feita a tentativa de conexão
    verificarErro(connectstate);                                 // o programa só prossegue caso não ocorram erros

    return s;
}
```

Figura 4: Conexão Monitor-Simulador (do ponto de vista do Simulador)

Na prática, vão ser enviadas mensagens do Simulador para o Monitor, sendo que o primeiro irá fazer *read* (lendo mensagens por parte do Simulador) e o segundo a operação *write* (mais propriamente, o envio das mesmas para o Monitor). De forma genérica, cada mensagem será interpretada pelo Monitor e registada a ocorrência indicada nessa mesma mensagem. Este comportamento por parte de ambos pode ser observado nas figuras abaixo.

```
void escreverNoMonitor(char message[])
{
    write(s,message,BUFFER_SIZE);          // é enviada uma mensagem para o Monitor
}
```

Figura 5: Envio de mensagens por parte do Simulador (para o Monitor)

```
char* lerDoSimulador()
{
    char * mensagem = (char *) malloc(sizeof(char)*BUFFER_SIZE);   // alocação de um array de chars
    read(simulador,mensagem,BUFFER_SIZE);                          // lê a mensagem enviada por parte do simulador
    return mensagem;                                               // retorna a dada mensagem
}
```

Figura 6: Receção das mensagens enviadas pelo Simulador

2. Descrição da implementação

2.1. Renomeações e funções auxiliares

Para facilitar a implementação e a sua manutenção (para não falar da leitura do código), foram feitas renomeações de determinadas funções e estruturas de dados para nomes mais simplificados e elegantes. Com o mesmo objetivo e também para aumentar a reutilização do código, foram feitas funções auxiliares.

2.1.1. Renomeações

No que toca às renomeações, para a parte da comunicação, o tipo int foi renomeado para ambos socket_t e client_t, de forma a ser mais apelativo. Do mesmo modo, em relação a *mutexes* (pthread_mutex_t) e a semáforos (sem_t), estes foram renomeados para mutex_t e semaforo_t, respetivamente. Para além disso, as variáveis do tipo pthread_t – as tarefas – foram renomeadas tomando agora o nome de tarefa_t. Por último, para encurtar, as estruturas *sockaddr_un* e *sockaddr* foram renomeadas (mantendo o mesmo nome) para que possa ser omitido a palavra-chave *struct.*

2.1.2. Funções auxiliares gerais

Como funções auxiliares que não estão em nenhuma categoria específica, temos as funções strequals, verificarErro, randWithProb, escreverNoLog.

Em relação à primeira, esta serve para igualar duas *strings* retornando o valor de 1 (*true*) no caso em que sejam iguais e 0 (*false*) caso contrário.

Do mesmo modo, a seguinte é usada para comparar resultados de determinadas funções críticas (como funções de comunicação) com o valor de -1, valor que simboliza a ocorrência de um erro numa dada função.

Da mesma forma, a função randWithProb, de uma forma genérica, trata de gerar números aleatórios consoante a probabilidade (entre 0 e 1) indicada no *input*.

Por último, a função escreverNoLog, como o nome indica, consiste em escrever

uma dada mensagem no ficheiro *log*, onde serão registadas as ocorrências dos eventos no decorrer da simulação.

2.1.3. Funções auxiliares de comunicação

Como já se encontra abordado anteriormente, as funções que se encontram disponíveis para esta parte da implementação são as seguintes: criarSocket, criarLigacaoSocket, esperarPorCliente e criarSocketCliente.

No que toca às duas primeiras, estas tratam de criar uma *socket* e uma estrutura onde consta o endereço do espaço a aceder pela *socket*. Recorrendo a estas duas, a função esperarPorCliente trata de efetuar as etapas para a comunicação por parte do servidor – *bind*, *listen* e *accept*. De forma semelhante, a função criarSocketCliente, através das duas primeiras, consiste em efetuar o procedimento para o cliente se comunicar com o servidor – *connect*.

2.1.4. Funções auxiliares de sincronização

Em relação à sincronização, como nas renomeações acima, as funções presentes nas bibliotecas semaphore.h e pthread.h para gerir o funcionamento de *mutexes* e semáforos foram renomeadas, de forma a encurtar e ficar com nomes mais intuitivos – Fechar, Abrir, Esperar e Assinalar – para cada tipo de objeto de sincronização, respetivamente. Para além destas operações, também se incluem as renomeações das operações de inicialização e destruição de um *mutex* e de um semáforo.

2.2. Estrutura de dados do utilizador

A estrutura de dados do utilizador contem os principais dados relacionados com o utilizador, que são o id, estado, prioritários, emViagem, tempoEspGuiche, tempoEspCarros,tempoEspDev. O id é o identificador de cada utilizador. A variável Prioritários indica se o utilizador é prioritário ou não. O inteiro estado é responsável por conter a informação sobre a situação do utilizador em relação a montanha-russa,ou seja, indica se o utilizador está na fila de comparar bilhetes ou

na filha de espera dos carrinhos ou se já desistiu ou se ainda está a andar no carrinho. O valor do atributo emViagem indica se o utilizador se encontra numa viagem. Os atributos tempoEspGuiche, tempoEspCarros,tempoEspDev são responsável por guardar o tempo de espera do utilizador nas filas EspGuiche , tempoEspCarros e tempoEspDev, respetivamente.

2.3. Ficheiros de configuração

Nesta implementação, tanto o Monitor como o Simulador terão o seu ficheiro de configuração, onde poderão ser feitos *tweaks* no funcionamento de uma das componentes, alterando certos parâmetros presentes na implementação.

2.3.1. Configuração do Simulador

No ficheiro de configuração do Simulador, o que consta são os seguintes parâmetros: taxa_população (frequência com que vão aparecendo pessoas); t_viagem (duração de uma viagem de montanha-russa); taxa_atendimento_compra (rapidez no atendimento no guiché de compra); taxa_atendimento_dev (rapidez no atendimento no guiché de devoluções); taxa_atendimento_carros (rapidez no atendimento na fila para os carros da montanha-russa); max_pessoas_dev (limite de pessoas no guiché de devoluções); max_pessoas (lotação total do parque de montanha-russa); taxa_desistencia (frequência com que há desistências por parte das pessoas); lotacao_carro (lotação de cada carro da montanha-russa).

2.3.2. Leitura dos ficheiros de configuração

A leitura dos ficheiros de configuração ,tanto do simulador como do monitor, seguem a mesma estrutura lógica, ou seja, um ciclo que copia para o *buffer* o valor de cada linha do ficheiro. Cada iteração do ciclo pode ser separado em duas partes. A primeira é encarregue por encontrar qual é o nome do parâmetro que esta a ser lido, como está representado na figura abaixo.

```
for(i = 0 ; buffer[i] != '=' ; i++);

/* Copia o nome do parâmetro */
strncpy(param,buffer,i);

/* Copia o valor do parâmetro */
strncpy(value,buffer+i+1,strlen(buffer)-i-1);
```

Figura 7: Leitura do nome do parâmetro

Depois numa segunda verificasse qual é o nome desse parâmetro e é copiado o valor do parâmetro que foi lido para o respetivo campo na respetiva estrutura de dados.

```
if(strequals(param,"t_viagem"))
{
  conf->t_viagem = atoi(value);
}else{

if(strequals(param,"max_pessoas"))
{
  conf->max_pessoas = atoi(value);
```

Figura 8: Atribuição dos valores dos parâmetros

2.4. Codificação e descodificação dos eventos

O monitor tem um ciclo que fica a espera que chegue algum código do simulador, quando chega o monitor interpreta o valo. Os valores possíveis do código são **0,1,2,3.**

Caso chegue o **valor** do código é **0** isto implica que a **simulação acabou** ,o monitor fica a espera que o simulador envie o número total de clientes para poder calcular as últimas estatísticas.

No caso que o **valor** enviado pelo simulador for **1** o monitor vai interpretar isto como a **compra** de um **bilhete** por um cliente, nesta situação o monitor precisa de receber do simulador o tempo de espera do cliente na fila para poder calcular as estáticas relacionas com o tempo médio de espera.

O monitor ainda pode receber o código enviado pelo simulador com o **valor 2** isto implica que houve algum **cliente** que **desistiu** ,o monitor para poder calcular as estatísticas relacionadas com a desistências fica a espera do valor de espera do cliente até este desistir.

Por fim o código ainda pode tomar mais um **valor** de **3** neste caso o monitor vai interpretar como que tenha acabado que **chegar** um **cliente** a fila de **compra** dos **clientes**.

Evento	Número do evento
FIM_SIMULACAO	0
COMPRA_BILHETE	1
DESISTENCIA	2
CHEGAR_GUICHE_COMPRA	3

2.5. Modelos de sincronização

Neste projeto optamos por realizar 3 modelos de sincronização, um deles basicamente para cada parte fundamental do ambiente envolvente do projeto.

Um modelo para sincronizar a fila do guiché, ou seja, a compra de bilhetes, outro para a fila das devoluções, ou seja para quando um cliente desiste de esperar e finalmente o último que controla a fila de espera para entrar na montanha russa e o acesso aos carrinhos da mesma. A forma como estes modelos implementam a sincronização é explicada com mais detalhe abaixo.

2.5.1. Guiché de compra de bilhetes

Em relação ao guiché de compra de bilhetes, temos uma bilheteira apenas com um funcionário e o acesso é feito apartir de duas filas, uma delas para clientes prioritários e outra para clientes não prioritários (na criação do cliente é feito um random para determinar se o cliente é prioritário ou não).

O modelo consiste em, temos dois semáforos um para a fila de prioritários **(sGCP)** e outro para a fila de não prioritários **(sGCNP)**, ambos inicializados a zero, quando chega um cliente a uma destas filas faz assinalar sobre o seu respetivo semáforo à espera que chegue a sua vez.

Existe uma função **guicheCompraFunc()**, que faz a verificação de se existe clientes prioritários ou não prioritários, enquanto houver prioritários esta função vai assinalando estes **(Assinalar(&sGCP))**, quando já não existir esta passa a

assinalar os não prioritários **(Assinalar(&sGCNP))**, este procedimento só acontece quando um random o permitir, ou seja, não são constantemente despachados clientes para permitir a existência de um tempo de espera que depois será usado para criar um tempo médio de espera no acesso ao guiché.

2.5.2. Guiché de devolução de bilhetes

Em relação ao modelo utilizado na fila para as devoluções, consiste numa simples fila sem prioridades em que o acesso à devolução é feito de um em um.

Este modelo é semelhante ao exemplo do barbeiro dado nas aulas teóricas, são usados dois semáforos, um para os clientes que chegam **(sDev)** inicializado a zero e outro para o funcionário da bancada de devolução **(sClienteDev)** também inicializado a zero.

O modelo consiste no seguinte, quando um cliente chega à fila de devolução faz assinalar sobre o semáforo do funcionário **(Assinalar(&sClienteDev))** para indicar que existe clientes à espera e de seguida faz esperar sobre o semáforo da fila dos clientes **(Esperar(&sDev))** para esperar pela sua vez. O funcionário conforme vai despachando um cliente assinala outro (enquando estiverem clientes à espera) e assim sucessivamente **(Assinalar(&sDev))**.

2.5.3. Fila da montanha-russa

Nós optamos por ter dois carros na montanha russa ambos com 5 lugares, para o modelo de sincronização foi usado um semáforo para cada carro, **sCarro1** e **sCarro2** respetivamente ambos inicializados a 5 (valor correspondente à lotação máxima de cada carro). É utilizado duas variáveis para tornar o acesso alternado a ambos os carros, isto é, se um cliente chega e é enviado para o carro1 o próximo a chegar será enviado para o carro2 obrigatóriamente.

É utilizado outro semáforo **(sFilaCarros)** como o semáforo principal que permite o acesso propriamente dito à montanha russa.

É usada também uma variável para indicar se existe uma viagem em curso ou não **(viagemEmCurso)** existe outra variável também que serve para indicar o

tempo da viagem que no nosso caso é fixo e tem o valor de 20s **(t_viagem)**.

Na prática o que acontece é o seguinte, quando um cliente chega à fila dos carros o que este faz é esperar pelo semáforo do carro1 ou carro2 consoante o valor da variável que controla o acesso alternado aos carros. Após isto quando forem assinalados neste semáforo vão ficar à espera no semáforo **sFilaCarros**, depois através de um random a função irá **filaEsperaCarros()** começar a viagem assinalando o semáforo **sFilaCarros** dez vezes (só são assinalados se já estiverem 10 clientes à espera na fila). Depois dentro desta mesma função é contabilizado o tempo de viagem, quando este tempo acabar é chamada a função **fimDaViagem()** que assinala os semáforos **sCarro1** e **sCarro2** colocando os clientes à espera no semáforo **sFilaCarros** que depois iram ser assinalados novamente pela função **filaEsperaCarros()** e o processo repete-se.