Projeto 2 de Sistemas Operativos

Restaurante: Mecanismos associados à execução e sincronização de processos e threads

Trabalho realizado por:

Ricardo Martins,nº112876

Rodrigo Jesus, nº113526

Ano letivo 2023/24

ÍNDICE

Introdução

O objetivo deste trabalho é compreender os mecanismos associados à execução e sincronização de processos e threads (ou simplesmente aprender a utilizar/manusear processos e threads). Assim foi-nos dado este trabalho, com o intuito de simular o funcionamento de um restaurante com as seguintes entidades: *chefe*, *grupo*, *rececionista* e *empregado de mesa*, através do uso de processos independentes que representam estas entidades e de semáforos que garantem a sua correta execução.

Metodologia

Para a resolução deste trabalho é necessário entender a função que cada entidade terá, bem como os semáforos utilizados e a sua relação com as entidades (exemplo: quando é que uma entidade faz "up" ou "down" a um semáforo).

Semáforos

Ao todo, foi necessário utilizarmos dez semáforos (presentes no ficheiro de código *sharedDataSync.h*): *receptionistReq, receptionistRequestPossible, waiterRequest, waiterRequestPossible, waitOrder, orderReceived, waitForTable["id do grupo"], requestReceived["número da mesa"], foodArrived["número da mesa"], tableDone["número da mesa"].* Repara-se que existem "pares" de semáforos, ou seja, um certo semáforo que tenha uma entidade X a fazer up e Y a fazer down, para outro semáforo teria Y a fazer up e X a fazer down, isto acontece porque estes semáforos estão intrinsecamente ligados, como por exemplo, *receptionistReq* e *receptionistRequestPossible*, onde o primeiro é o pedido ao rececionista e o segundo é a disponibilidade do rececionista para receber esse pedido, que, como vamos ver na secção das funções das entidades, o grupo chega, faz down no segundo semáforo e caso o rececionista esteja disponível, entra na região crítica e por fim faz up no primeiro, visto que fez de facto um pedido.

Entidades responsáveis por fazer "up" e "down" de cada semáforo:

- **receptionistReq**: Up: *grupo*; Down: *rececionista*.
- **receptionistRequestPossible**: Up: *rececionista*; Down: *grupo*.
- **waiterRequest**: Up: *grupo, chefe*; Down: *empregado de mesa*.
- **waiterRequestPossible**: Up: *empregado de mesa*; Down: *grupo, chefe*.

- **waitOrder**: Up: *empregado de mesa*; Down: *chefe*.
- **orderReceived**: Up: *chefe*; Down: *empregado de mesa*.
- **waitForTable[“id do grupo”]**: Up: *rececionista*; Down: *grupo* (com o respetivo id).
- **requestReceived[“número da mesa”]**: Up: *empregado de mesa*; Down: *grupo* (que está na mesa com o respetivo número).
- **foodArrived[“número da mesa”]**: Up: *empregado de mesa*; Down: *grupo* (com o respetivo id).
- **tableDone[“número da mesa”]**: Up: *rececionista*; Down: *grupo* (com o respetivo id).

Funções das entidades

Chefe (um):

- Recebe pedidos do empregado de mesa e prepara a comida de cada grupo.
- Função ***waitForOrder***: Como o próprio nome indica é nesta função que o chefe aguarda o pedido feito pelo empregado de mesa, fazendo “down” no semáforo *waitOrder*. De seguida entra na região crítica (assim que o semáforo o permite) e a primeira coisa que faz é atualizar o estado do *chefe* para *COOK*, de seguida salva o último grupo que fez o pedido, guarda o estado geral e sinaliza ao empregado, fazendo “up” ao semáforo *orderReceived*, que o pedido foi recebido saindo assim da região crítica.
- Função ***processOrder***: Nesta função a primeira coisa que é feita é uma simulação do processo de preparação do pedido (com apoio da função *usleep*) e é feito “down” ao semáforo *waiterRequestPossible* para sinalizar que o chefe aguarda a oportunidade de chamar o empregado para informar que o pedido está pronto. De seguida, dentro da região crítica, o estado do chefe é atualizado para *WAIT_FOR_ORDER*, o tipo de pedido feito ao empregado é atualizado para *FOODREADY* e mais uma vez é salvo o grupo que fez o pedido e o estado geral. No final é feito um “up” ao semáforo *waiterRequest* que sinaliza que o empregado já foi informado que o pedido está pronto.

Grupo (vários):

- Vai ao restaurante, pede uma mesa, de seguida a refeição e por fim paga a conta.
- Função ***checkInAtReception***: É a primeira função a ser executada por cada *grupo* assim que chegam ao restaurante, começa por fazer “down” ao semáforo *receptionistRequestPossible*. Após o semáforo o permitir, entra na região crítica

e atualiza-se os estados do *grupo* (para *ATRECEPTION*), do tipo de pedido ao *rececionista* (para *TABLEREQ*, visto que o *grupo* pediu uma mesa para se sentar) e é guardado o *id* do *grupo*, bem como o estado geral. De seguida é sinalizado que o pedido por uma mesa foi efetuado fazendo "up" no *receptionistReq* e fazendo "down" no *waitForTable[id]* com o id do grupo, onde o mesmo espera caso não exista mesa livre.

- Função ***orderFood***: É nesta função que é feito o pedido ao empregado de mesa. Segue o mesmo padrão da função anterior (visto também se tratar de um pedido a um funcionário) começando com um "down" ao *waiterRequestPossible* e sem seguida, já na região crítica, atualiza os estados do grupo, do tipo de pedido, do id do grupo que fez o pedido, salva o estado geral e por fim faz "up" ao *waiterRequest*, sinalizando que o empregado já recebeu o pedido.

- Função ***waitFood***: Começa por guardar o id da mesa (*tableId*) em que o grupo se encontra (é opcional, foi apenas feito por conveniência e maior clareza no código), depois entra na região crítica apenas para atualizar o estado do grupo para *WAIT_FOR_FOOD*, guardar o estado e sai logo em seguida. Faz "down" no *foodArrived[tableId]* e assim que a comida tiver chegado, entra novamente na região crítica e atualiza e o estado do grupo para *EAT*.

- Função ***checkOutAtReception***: É guardado outra vez o id da mesa (tableId) e, na primeira metade desta função, segue o mesmo padrão da função *checkInAtReception*, faz "down" no *receptionistRequestPossible*, assim que estiver disponível, entra na região crítica, atualiza os estados do grupo (para *CHECKOUT*), do tipo de pedido (para *BILLREQ*, que é o pedido da conta) e guarda o id do grupo que pediu a conta. Depois faz "up" no *receptionistReq* para avisar que foi efetuado um novo pedido. De seguida faz "down" do *requestReceived[tableId]*, e assim que o rececionista "entregar" a conta, entra novamente na região crítica para atualizar o estado do grupo para *LEAVING* e por fim é feito um "down" no *tableDone[tableId]* para sinalizar que aquele grupo foi embora e aquela mesa fica novamente disponível.

Rececionista (um):

- Atribui uma mesa a cada grupo e recebe os pagamentos dos pedidos.

- Funções ***decideTableOrWait*** e ***decideNextGroup***: A primeira serve o propósito de percorrer a estrutura onde estão as mesas ocupadas e assim decidir se o grupo pode ocupar a mesa ou não. A segunda decide o próximo grupo que vai atender assim que exista uma mesa disponível.

- Função ***waitForGroup***: Entra na região crítica para salvar o estado do rececionista como 0. De seguida faz "down" no *receptionistReq* para indicar que já recebeu o pedido, entra novamente na região crítica e atribui o pedido à variável *ret* do tipo *request* e termina com um "up" ao *receptionistRequestPossible* para sinalizar que terminou de processar o pedido

e que volta a estar disponível para atender o próximo.

- Função ***provideTableOrWaitingRoom***: Entra na região crítica e atualiza o estado do grupo que está a ser atendido para *ATRECEPTION* e chama a função *decideTableOrWait* para obter o id da mesa que foi atribuída ao grupo, caso essa mesa não exista, significa que o grupo tem de esperar, atualizando, numa estrutura auxiliar que contém os grupos e se eles têm mesa, o estado para *WAIT* e somando um ao número de grupos que estão à espera, caso contrário, atualiza o estado do rececionista para *ASSIGNTABLE* e, na estrutura auxiliar, para *ATTABLE*.

- Função ***receivePayment***: Na região crítica, o estado do rececionista é atualizado para *RECVPAY*, na estrutura auxiliar dos grupos, o estado do grupo com o respetivo id é atualizado para *DONE* e a mesa em que se encontravam é "libertada", passando assim a estar disponível. Em seguida há uma chamada à função *decideNextGroup* e, dependendo se existe ou não algum grupo à espera, é atribuída uma mesa ao próximo grupo que está à espera e no fim faz "up" ao *tableDone[tableId]* para indicar que a mesa passa a estar vaga.

Empregado de mesa (um):

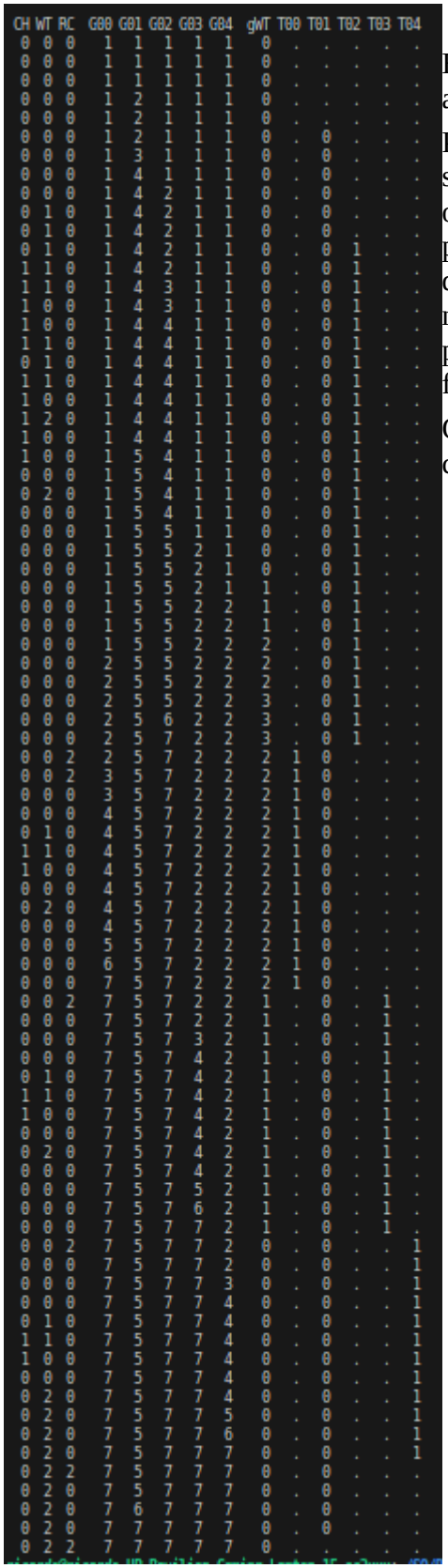
- Recebe os pedidos de cada grupo e entrega as respetivas refeições.

- Função ***waitForClientOrChef***: Entra na região crítica apenas para atualizar o estado do empregado para *WAIT_FOR_REQUEST*, depois faz "down" no *waiterRequest* para indicar que está à espera de um pedido, seja de um grupo ou do chefe. Novamente na região crítica (após chegar um pedido) uma variável to tipo *request* assume o pedido que foi feito, de acordo com que o fez. Por fim, faz "up" ao *waiterRequestPossible* para indicar que está outra vez disponível para receber novos pedidos.

- Função ***informChef***: Guarda o id da mesa que o grupo que fez o pedido está a ocupar (*tableId*), entra na região crítica e atualiza, de novo, o seu estado para *INFORM_CHEF* e o id do grupo que pediu a comida. Por fim faz "up" no *waitOrder* para liberar o chefe, assim permitindo que este receba novos pedidos, e no *requestReceived[tableId]* para informar o grupo de que o pedido foi recebido, e "down" no *orderReceived* para esperar que o cozinheiro receba o pedido.

- Função ***takeFoodToTable***: Guarda o id da mesa que o grupo está a ocupar, entra na região crítica para atualizar o seu estado para *TAKE_TO_TABLE* e termina com um "up" ao *foodArrived[tableId]*, que faz com que o grupo saiba que a comida está pronta.

Testes realizados



CH	WT	RC	G00	G01	G02	G03	G04	gWT	T00	T01	T02	T03	T04
0	0	0	1	1	1	1	1	0	.	.	.	.	.
0	0	0	1	1	1	1	1	0	.	.	.	.	.
0	0	0	1	1	1	1	1	0	.	.	.	.	.
0	0	0	1	2	1	1	1	0	.	.	.	.	.
0	0	0	1	2	1	1	1	0	.	.	.	.	.
0	0	0	1	2	1	1	1	0	.	0	.	.	.
0	0	0	1	3	1	1	1	0	.	0	.	.	.
0	0	0	1	4	1	1	1	0	.	0	.	.	.
0	0	0	1	4	2	1	1	0	.	0	.	.	.
0	1	0	1	4	2	1	1	0	.	0	.	.	.
0	1	0	1	4	2	1	1	0	.	0	.	.	.
0	1	0	1	4	2	1	1	0	.	0	1	.	.
1	1	0	1	4	2	1	1	0	.	0	1	.	.
1	1	0	1	4	3	1	1	0	.	0	1	.	.
1	0	0	1	4	3	1	1	0	.	0	1	.	.
1	0	0	1	4	4	1	1	0	.	0	1	.	.
1	1	0	1	4	4	1	1	0	.	0	1	.	.
0	1	0	1	4	4	1	1	0	.	0	1	.	.
1	1	0	1	4	4	1	1	0	.	0	1	.	.
1	0	0	1	4	4	1	1	0	.	0	1	.	.
1	2	0	1	4	4	1	1	0	.	0	1	.	.
1	0	0	1	4	4	1	1	0	.	0	1	.	.
1	0	0	1	5	4	1	1	0	.	0	1	.	.
0	0	0	1	5	4	1	1	0	.	0	1	.	.
0	2	0	1	5	4	1	1	0	.	0	1	.	.
0	0	0	1	5	4	1	1	0	.	0	1	.	.
0	0	0	1	5	5	1	1	0	.	0	1	.	.
0	0	0	1	5	5	2	1	0	.	0	1	.	.
0	0	0	1	5	5	2	1	0	.	0	1	.	.
0	0	0	1	5	5	2	1	1	.	0	1	.	.
0	0	0	1	5	5	2	2	1	.	0	1	.	.
0	0	0	1	5	5	2	2	1	.	0	1	.	.
0	0	0	1	5	5	2	2	2	.	0	1	.	.
0	0	0	2	5	5	2	2	2	.	0	1	.	.
0	0	0	2	5	5	2	2	2	.	0	1	.	.
0	0	0	2	5	5	2	2	3	.	0	1	.	.
0	0	0	2	5	6	2	2	3	.	0	1	.	.
0	0	0	2	5	7	2	2	3	.	0	1	.	.
0	0	2	2	5	7	2	2	2	1	0	.	.	.
0	0	2	3	5	7	2	2	2	1	0	.	.	.
0	0	0	3	5	7	2	2	2	1	0	.	.	.
0	0	0	4	5	7	2	2	2	1	0	.	.	.
0	1	0	4	5	7	2	2	2	1	0	.	.	.
1	1	0	4	5	7	2	2	2	1	0	.	.	.
1	0	0	4	5	7	2	2	2	1	0	.	.	.
0	0	0	4	5	7	2	2	2	1	0	.	.	.
0	2	0	4	5	7	2	2	2	1	0	.	.	.
0	0	0	4	5	7	2	2	2	1	0	.	.	.
0	0	0	5	5	7	2	2	2	1	0	.	.	.
0	0	0	6	5	7	2	2	2	1	0	.	.	.
0	0	0	7	5	7	2	2	2	1	0	.	.	.
0	0	2	7	5	7	2	2	1	.	0	.	1	.
0	0	0	7	5	7	2	2	1	.	0	.	1	.
0	0	0	7	5	7	3	2	1	.	0	.	1	.
0	0	0	7	5	7	4	2	1	.	0	.	1	.
0	1	0	7	5	7	4	2	1	.	0	.	1	.
1	1	0	7	5	7	4	2	1	.	0	.	1	.
1	0	0	7	5	7	4	2	1	.	0	.	1	.
0	0	0	7	5	7	4	2	1	.	0	.	1	.
0	2	0	7	5	7	4	2	1	.	0	.	1	.
0	0	0	7	5	7	4	2	1	.	0	.	1	.
0	0	0	7	5	7	5	2	1	.	0	.	1	.
0	0	0	7	5	7	6	2	1	.	0	.	1	.
0	0	0	7	5	7	7	2	1	.	0	.	1	.
0	0	2	7	5	7	7	2	0	.	0	.	.	1
0	0	0	7	5	7	7	2	0	.	0	.	.	1
0	0	0	7	5	7	7	3	0	.	0	.	.	1
0	0	0	7	5	7	7	4	0	.	0	.	.	1
0	1	0	7	5	7	7	4	0	.	0	.	.	1
1	1	0	7	5	7	7	4	0	.	0	.	.	1
1	0	0	7	5	7	7	4	0	.	0	.	.	1
0	0	0	7	5	7	7	4	0	.	0	.	.	1
0	2	0	7	5	7	7	4	0	.	0	.	.	1
0	2	0	7	5	7	7	5	0	.	0	.	.	1
0	2	0	7	5	7	7	6	0	.	0	.	.	1
0	2	0	7	5	7	7	7	0	.	0	.	.	1
0	2	2	7	5	7	7	7	0	.	0	.	.	.
0	2	0	7	5	7	7	7	0	.	0	.	.	.
0	2	0	7	6	7	7	7	0	.	0	.	.	.
0	2	0	7	7	7	7	7	0	.	0	.	.	.
0	2	2	7	7	7	7	7	0	.	.	.	.	.

Esta figura é um dos muitos resultados que obtivemos ao efetuar os nossos diferentes testes ao código.

Para testar o funcionamento correto do programa, já na sua fase final, decidimos compilar o nosso projeto com o comando make all e executar o comando run 1000, para os vários números de grupos e diferentes tempos dentro do ficheiro config.txt , sem nunca entramos numa situação de deadlock e tendo o output do nosso programa sempre convergido para o da solução fornecida pelos professores.

Concluímos assim que a nossa solução do projeto foi corretamente implementada e testada.

Conclusão

Em síntese, a elaboração deste relatório evidencia a aplicação prática e aprofundada dos conceitos de processos, threads e semáforos no contexto da programação concorrente, o que nos ajudou bastante a reforçar a nossa compreensão sobre esses temas.
A simulação do funcionamento de um restaurante, com as suas diferentes entidades interagindo de forma coordenada através do uso de processos e semáforos, realça assim a relevância desses mecanismos para modelar sistemas complexos.