Universidade Federal de Santa Catarina
Relatório do Projeto Prático 1

Participantes:
Thiago Comelli(19100546)
André Luiz Souza Santos(19150871)

Novembro de 2020

1. Distribuição do Trabalho

A execução do trabalho ficou distribuída entre ambos os alunos, de forma que cada um propor mudanças de código, visando modelar o código, inseriu mudanças neste e discutiram formas de criar soluções para problemas identificados. Como o trabalho não apresentava um grau de complexidade muito elevado e era relativamente bastante simples, não vimos como necessário fazer separações mais exatas do que cada um faz, fomos pensando juntos e vendo quais soluções eram melhor e indo implementando elas.

2. Estruturas de Sincronização utilizadas

Ao total foram utilizadas 7 estruturas de sincronização, 3 mutexes e 4 semáforos, como mostrado abaixo:

- Semáforo “semClerk”: Utilizado para fazer a liberação das threads dos Clerks, assim, um funcionário só começa a trabalhar e chamar uma senha, após pelo menos um cliente coletar uma senha.
- Semáforo “semClient”: Utilizado para os clientes esperarem o Cooker atualizar a lista de pedidos prontos, e após a liberação fazer a checagem de pedidos prontos.
- Semáforo “semClientWaitToCall”: Utilizado para os clientes esperarem até serem chamados/liberados pelo funcionário após o mesmo efetuar a chamada “get_retrieved_ticket()”.
- Semáforo “semClerkWaitToClient”: Utilizado para o Clerk esperar o cliente terminar de fazer o pedido.
- Mutex “clientMutex”: Utilizado para coletar uma senha e impedir que mais de um cliente possua a mesma senha.
- Mutex “clerkMutex”: Utilizado para os funcionários coletarem uma senha para efetuar a chamada e impedir que mais de um funcionário colete a mesma senha.
- Mutex “clientWaitMutex”: utilizado para dar update na quantidade de clientes esperando pelo pedido ficar pronto, essa quantidade serve para o cozinheiro liberar o semáforo “semClient” posteriormente.

```
76  void* client(void *args) {
77      client_t* cl = malloc(sizeof(int));
78      cl->id = (long)args;
79
80      int lastCheck = -1;
81
82      // MUTEX PARA GARANTIR QUE VALORES NAO IRAO
83      pthread_mutex_lock(&clientMutex);
84
85      int pw = get_unique_ticket(pc);
86
87      // GARANTE QUE O FUNCIONARIO SÓ COMECE A TR
88      sem_post(&semClerk);
```

```
121 void* clerk(void *args) {
122     clerk_t* ck = malloc(sizeof(int));
123     ck->id = (long)args;
124
125     sem_wait(&semClerk);
126
```

Funcionamento do “semClerk”, libera a thread do funcionário para começar a trabalhar.

```
        sem_wait(&semClient);
        for (int i = 0; i < balcao.actualSize; i++)
        {
            if(balcao.data[i].password_num == od->password_num){
                pthread_mutex_lock(&clientWaitMutex);
                pc->clientsWaitingOrder--;
                pthread_mutex_unlock(&clientWaitMutex);

                pthread_exit(NULL);
            }
        }
    }
```

```
        for (int i = 0; i < pc->clientsWaitingOrder; i++)
        {
            sem_post(&semClient);
        }
```

O "semClient" na função "client_wait_order()", aguarda sua liberação para fazer a verificação dos pedidos prontos, a liberação é feita pela função cooker após ela sofrer alguma inserção.

```
        while(pc->clerks_order_spot[ck->id] == 0){
            // SEMAFORO QUE AGUARDA SUA LIBERACAO QU
            sem_wait(&semClerkWaitToClient);
        }
```

```
void client_inform_order(order_t* od, int clerk_id) {
    pc->clerks_order_spot[clerk_id] = od;
    for (int i = 0; i < numClerks; i++)
    {
        sem_post(&semClerkWaitToClient);
    }

}
```

O "semClerkWaitToClient" dentro da função clerk, aguarda a liberação após o cliente efetuar o pedido.

```
    int pw = get_unique_ticket(pc);

    // GARANTE QUE O FUNCIONARIO SÓ COMECE A TRABALH
    sem_post(&semClerk);

    pthread_mutex_unlock(&clientMutex);

    // AGUARDA A LIBERACAO DO FUNCIONARIO APOS A CHA
    sem_wait(&semClientWaitToCall);

    while (true)
    {
        // CONFICIONAL NECESSARIO PARA QUE A VERIFIC
        if(lastCheck < pc->lastCheckPwd){
```

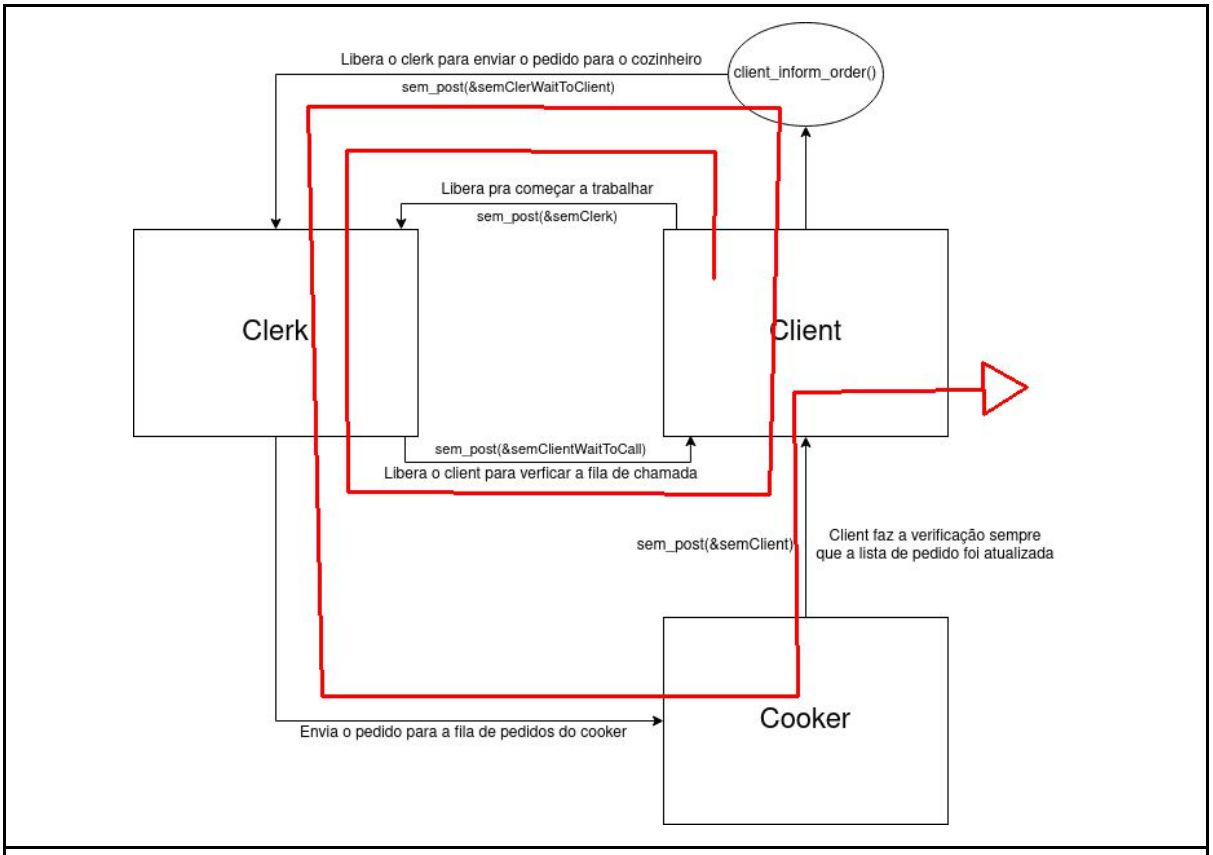
```
        int val = get_retrieved_ticket(pc);

        // LIBERA O CLIENTE PARA FAZER A VERI
        sem_post(&semClientWaitToCall);

        pthread_mutex_unlock(&clerkMutex);

        if(val == -1){
            break;
```

O "semClientWaitToCall" libera o cliente para começar a fazer verificação na lista de senhas chamadas, somente após o funcionário efetuar a chamada.



Representação de como o ciclo de um atendimento funciona com os semáforos.

3. Possibilidade de não uso de estruturas de sincronização

Após, subsequente análise das estruturas de sincronização, observa-se que estas são de suma importância para o funcionamento do codigo de forma satisfatória, tendo em vista a retirada de pedidos, fila destes por parte dos atendentes e realização do pedido pelo cozinheiro, caracterizam operações que necessitam de sincronismo, afim de impedir problemas de condição de corrida e exclusão mútua.

Contudo, ainda é possível realizar o trabalho sem o uso de estruturas de sincronização como os semáforos nas regiões onde não é necessário leitura e escritura de dados. Um exemplo disso no nosso trabalho, é na região onde o cliente fica aguardando o pedido ficar pronto. Poderíamos ter usado um laço de repetição que ficasse verificando se a lista onde está armazenado os pedidos foi alterada ou não para assim fazer a verificação, contudo acabamos deixando essa parte mais otimizada, e a verificação só ocorre com a liberação dos semáforos por parte do Cooker.

Mas de fato, realizar o trabalho inteiro acreditamos ser inviável, e se isso for possível, seria uma solução muito mal otimizada e demandaria muito processamento uma vez que precisaria de muitos laços de repetição para fazer verificações.